DISTRIBUTED ANALYSIS OF MARKOV CHAINS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Mark Mestern
April 1998

Supervised by
Prof P.S. Kritzinger

The University of Cape Town has been given
the right to reproduce this thesis in whole
or in part. Copyright is held by the author.

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

Acknowledgements

I would like to express my thanks and appreciation to:

- My supervisor, Professor Pieter S. Kritzinger, for his guidance and supervision.
- My family for their support, and love for this perpetual student.
- William Knottenbelt, for suggesting the topic, and whose excellent MSc. thesis was an inspiration.
- My fellow DNA lab colleagues and the other post-graduate students I met at the university who created a friendly and interesting place to work.
- Sandi Donno, Aleks Strez and Gerald Parker, our system administrators, for being ready to fix the many problems I found (and often caused) with the computer networks.
- Eve Gill and Mary Wood for shielding me from the paperwork.
- The South African Foundation for Research and Development for providing funding.

Abstract

This thesis examines how parallel and distributed algorithms can increase the power of techniques for correctness and performance analysis of concurrent systems. The systems in question are state transition systems from which Markov chains can be derived. Both phases of the analysis pipeline are considered: state space generation from a state transition model to form the Markov chain and finding performance information by solving the steady state equations of the Markov Chain.

The state transition models are specified in a general interface language which can describe any Markovian process. The models are not tied to a specific modelling formalism, but common formal description techniques such as generalised stochastic Petri nets and queuing networks can generate these models.

Tools for Markov chain analysis face the problem of state spaces that are so large that they exceed the memory and processing power of a single workstation. This problem is attacked with methods to reduce memory usage, and by dividing the problem between several workstations. A distributed state space generation algorithm was designed and implemented for a local area network of workstations. The state space generation algorithm also includes a probabilistic dynamic hash compaction technique for storing state hash tables, which dramatically reduces memory consumption.

Numerical solution methods for Markov chains are surveyed and two iterative methods, BiCG and BiCGSTAB, were chosen for a parallel implementation to show that this stage of analysis also benefits from a distributed approach.

The results from the distributed generation algorithm show a good speed up of the state space generation phase and that the method makes the generation of larger state spaces possible. The distributed methods for the steady state solution also allow larger models to be analysed, but the heavy communications load on the network prevents improved execution time.

Contents

Abstract ii

Acknowledgements iii

1 Introduction 1

1.1 Communicating Concurrent Systems 1

1.2 State Transition Systems 2

1.3 Performance Analysis 2

1.4 Analysis Tools 3

1.4.1 DNAmaca 3

1.4.2 Paramaca 5

1.5 Other Markov Chain Tools 5

1.5.1 USENUM 5

1.5.2 MARCA 7

1.5.3 SPNP 9

1.6 Dissertation Roadmap 11

2 Background Theory 12

2.1 Introduction 12

2.1.1 Notation 12

2.2 Stochastic Processes 13

2.2.1 Homogeneity 13
2.2.2 The Markov Property 13
2.2.3 Discrete Time Markov Chains 14
2.2.4 Irreducibility 14
2.2.5 Steady State Solution 16
2.2.6 Continuous Time Markov Chains 17
2.3 State Transition Formalisms 18
2.3.1 Petri Nets 18
2.3.2 Queuing Networks 20
2.3.3 Direct Modelling 22

3 Message Passing Environments **24**
3.1 Approaches to Parallelism 24
3.1.1 Specialist Parallelism 24
3.1.2 Agenda Parallelism 25
3.1.3 Result Parallelism 25
3.2 Message Passing Libraries 26
3.3 The PVM system 26
3.4 The MPI system 27
3.5 Paramaca Requirements 28

4 State Space Generation **29**
4.1 Introduction 29
4.2 Serial State Space Generation Algorithm 31
4.3 Storage Techniques 32
4.3.1 Memory Allocation Strategy and Reliability 32
4.3.2 Exhaustive Dynamic Methods 33
4.3.3 Probabilistic Static Methods 33

4.3.4 Probabilistic Dynamic Methods 34
4.3.5 Reliability of the probabilistic dynamic state hash table 36
4.4 DNAmaca 37
4.5 Other Work on Parallel State Space Exploration 38
4.6 Parallel State Space Generation 39
4.6.1 The Parallel Algorithm 39
4.6.2 State Communication 41
4.6.3 Detecting Termination 42
4.6.4 Partitioning Function 45
4.6.5 Testing Methods 47
4.6.6 Time Results 49
4.6.7 Space Results 51

5 The Steady State Solution of Markov Chains **54**
5.1 Introduction 54
5.2 Functional Analysis 54
5.3 Steady State Solution 56
5.4 Direct Methods 57
5.4.1 Gaussian Elimination 58
5.4.2 LU Decomposition 60
5.5 The parallel operations/primitives 60
5.5.1 Matrix Distribution 61
5.5.2 Parallel Matrix Vector Multiplication 62
5.6 Classic Iterative methods 66
5.6.1 Scaling and Convergence 67
5.7 Jacobi's Method 67
5.7.1 Gauss-Seidel 69

5.7.2 Successive Overrelaxation 70
5.8 Conjugate Gradient Techniques 71
5.8.1 Conjugate Gradients 71
5.8.2 Bi-Conjugate Gradient (BiCG) 73
5.8.3 Conjugate Gradient Squared (CGS) and Parallelisation 73
5.8.4 Bi-conjugate Gradient Stabilised (BiCGSTAB) and Parallelisation . 79

6 Conclusion **85**
6.1 Introduction 85
6.2 Distributed Generation and Solution 85
6.3 Future Work 86

Bibliography **88**

Chapter 1

Introduction

1.1 Communicating Concurrent Systems

The first automated systems were typically controlled by one program running on a single computer and the interaction between multiple systems was simple. Most modern systems have a very different character in that they either have many cooperating subsystems working independently or are designed to form part of a larger structure. Examples of this type of design are found from telephony and data networks to controllers in vehicles or machinery and these systems are termed communicating concurrent systems. Although this trend has encouraged many novel applications of technology, communicating concurrent systems bring a new breed of problems.

No matter how carefully designed a system may be, when it must interact with other systems it is often difficult to foresee the behaviour that might result from its communications. These interactions between systems bring a qualitative increase in complexity and can cause unexpected and often subtle problems. In most cases these problems result in increased costs due to incorrect information, loss of service and maintenance costs. If the problems affect safety critical systems then results could even be disastrous.

Clearly verifying the behaviour of communicating concurrent systems during the system's design is far preferable to coping with failures after they have occurred. The design of communicating concurrent systems must therefore give special consideration to the problems of correctness and performance. Formal methods for evaluating system behaviour are used to this end. These methods all construct a model of the system to which automated analysis may be applied.

1.2 State Transition Systems

The modelling method used by this dissertation is that of state transition systems. This methodology assumes that the system may be described by a finite number of discrete states and that by examining the current state one can say which states may succeed it.

Markovian models describe a system by explicitly enumerating every state in which the system can be and the ways in which the system can move from one state to another. This is the *state space* of the system and is represented by a graph with each state as a vertex and the transitions between states as arcs. The properties of this graph can then be used to deduce results about the system. For example, a state which has no arcs leading from it indicates a deadlock. If the designer wants to check more specific properties of the system this can also be done provided he can express the property as a condition on the state descriptor. Each state can then be automatically checked for specified condition.

When systems are modelled in this manner their state space can quickly become unmanageably large even for fairly simple systems. Systems can easily reach many millions of states. This is known as the state space explosion problem and because computers have limited time and memory this restricts the size of the systems that can be analysed.

The person creating the model usually cannot list the states of the system, indeed finding the state space is half the work. The model must rather be specified as an initial state from which the system starts and a set of rules describing the conditions under which the system can make a transition from one state to a new state. The state space is generated from the initial state and the from rules for finding the states which result from the transitions. The process of finding all these states is called state space exploration or generation.

The state transition model may be written manually or produced by formal modelling techniques. Petri nets and queuing networks are techniques for describing systems which naturally lead to expression in terms of states and transitions. However as long as the model is expressed in this form it does not depend on any particular modelling formalism.

1.3 Performance Analysis

If the system is one where the future states depend only on the current state of the system, i.e. not on any previous state or on how long the system has been in that state, then the system is called *memoryless* and the states and the transitions between them are called a

Markov chain. Each transition between two states is labelled with a value indicating the rate at which the system moves between those states.

If the process can change state at any time then the process is a continuous time Markov chain (CTMC) and each transition is given a *rate.* If the process can only change state at discrete intervals then the process is a discrete time Markov chain (DTMC) and each transition has a value which gives the probability of the system making that transition. These rates are usually represented by a square matrix Q (P in DTMC case) of simultaneous linear equations reflecting the probability of moving between the states.

The solution of these equations gives the proportion of time that the system is in any particular state and this result is called the *steady state solution* of the Markov Chain. This information can be used to provide performance information about the system.

For some states the model has an instantaneous transition which occur without any time passing which the system in that state. These states have a probability of 0 in the steady state solution and are called *vanishing.* These can be removed by manipulation of the matrix after state space exploration or by on-the-fly elimination which removes them as they are produced during state space exploration.

1.4 Analysis Tools

An analysis tool for such state transition systems uses a model description to produce the reachability graph through the state space exploration process. If the solution of steady state equations is desired then the resulting matrix is used by a steady state solver to find the proportion of time that the system is in each state.

1.4.1 DNAmaca

DNAmaca is a tool written by William Knottenbelt of the Data Network Architectures Laboratory of the University of Cape Town. [Kno96]

DNAmaca accepts models described in a high-level description of the model which is then converted to C++ code, compiled and linked with library code to form a state space generator.

The interface consists of a model file containing:

- A state description vector of integer values where each configuration of this vector represents a different state.
- A state descriptor for the initial state.
- Transition declarations. These specify the conditions for the transition to occur as well as the effect on the state descriptor. The transitions are labeled as timed or instantaneous and also have a rate at which they occur. There is also a provision for a priority scheme. The transition's effects and conditions are specified using ordinary expressions in C and therefore can be as general as the user desires.
- A set of invariants for which each state should be checked.

Constants and functions can be inserted as C code and there are options for control of the state space generation and steady state solution process and for customising result reporting. A parser generates C code from the model description file and links it with the rest of the libraries to form the state space generator.

DNAmaca's state space generator uses a new probabilistic dynamic state management technique to achieve considerable savings in memory use over the conventional exhaustive and static techniques. The state space generator also incorporates on-the-fly vanishing state elimination to reduce the size of the state space.

A functional analyser examines the state transition matrix Q to check if the Markov Chain is irreducible by checking if there is a single strongly connected component.

DNAmaca implements a variety of steady state solvers including direct methods, classical iterative techniques and Krylov subspace techniques and decompositional techniques.

DNAmaca implements the steady state solvers described below:

- **Direct Methods**: Sparse Gaussian Elimination and Grassmann's method. These methods are very accurate but are only suitable for the solution of small models.
- **Classical Iterative Methods**: Gauss-Seidel, fixed SOR and dynamic SOR. Dynamic SOR is the most effective of these methods.
- **Krylov subspace techniques**: BiCG, CGNR, CGS, BiCGSTAB, BiCGSTAB2 and TFQMR. CGS exhibits rapid convergence and has the lowest memory requirements of the methods in this class.

- **Decompositional techniques**: AI and AIR. AIR with table driven relaxation has very low memory requirements and achieves rapid and smooth convergence once its initialisation phase has completed.

State Space Exploration is limited to about 4 000 000 states on a Sun Sparc classic with 64MB of memory. The steady state solver is limited to generation of about 500 000 states.

1.4.2 Paramaca

Paramaca, which was developed in the course of this thesis, is an extension to DNAmaca which explores how the parallelism of a network of workstations can extend the power of the tool to analyse systems with larger state spaces.

The memory of a computer is the main factor limiting the size of the model that can be analysed. If the state space can be partitioned between several workstations, the limitations of a single workstation can be overcome. Paramaca includes parallel algorithms for both the state space generation and the steady state solver..

1.5 Other Markov Chain Tools

1.5.1 USENUM

USENUM [Scz87] is a Markov chain analyser developed at the University of Dortmund by Michael Szittnick in 1987. The analyser was originally implemented on a BS2000 system using SIMULA, then rewritten in C for use on UNIX systems. With USENUM, it is possible to analyse models with up to about 100 000 states (both vanishing and tangible) on a machine with 64 Mb RAM.

State and Transition representation: In USENUM, a state is represented as an integer-valued row state descriptor vector. Transitions from the current state to the next state are specified by:

- an **enabling condition** expressed as a conditional C expression on elements of the current state vector.
- a **transition effect** expressed as C assignments to elements of the next state vector based on operations on the elements of the current state vector.

and a list of transitions which can occur between buckets. The data file may also contain information such as the solution method for computing the steady-state distribution; alternatively, this information may be entered interactively during the solution process.

The subroutine *RATE* must be written to return the rate at which transitions occur between each possible pair of source and destination buckets, as well as the destination states that result from these transitions. The *RATE* subroutine is not restricted to changing the source and destination buckets, but may define the destination state completely. This implies that the total number of balls in a state descriptor need not be conserved but can be created and destroyed as needed.

The subroutine *INSTANT* must examine destination states and determine whether or not they are vanishing. If so, the subroutine must return the set of tangible destination states and associate with each the probability that it is the result of the timed transition from the source state.

State Space exploration algorithm: MARCA implements an exhaustive breadth-first search state space exploration algorithm which stores states in a list. On-the-fly vanishing state elimination is performed during state space generation. There is no timeless trap detection.

Numerical solution techniques: MARCA implements four classes of steady-state solution methods:

- **Direct methods**: Sparse Gaussian elimination.
- **Classical Iterative methods**: SOR, symmetric SOR (SSOR), power method, fixed-point iterations with preconditioning.
- **Krylov Subspace methods**: Arnoldi method (2 variants), GMRES (3 variants).
- **Decomposition-based techniques**: An iterative aggregation/disaggregation solver for nearly completely decomposable (NCD) chains. MARCA includes a unique facility for detecting near-decomposable components of the transition matrix.

After the steady-state vector has been computed, MARCA determines the distribution of the balls in each of the buckets and the mean and standard deviation of that distribution. For more complex performance measures, MARCA allows the user access to the list of states and the steady-state probability vector.

In addition, MARCA supports transient analysis through randomisation, Runge-Kutta, Adams ODE solver and matrix powering techniques.

Applications: A graphical front-end application for MARCA, known as XMARCA [KS95], has been developed. XMARCA is a sophisticated queueing network analyser which runs under the X-window system on UNIX systems. XMARCA allows users to build queueing networks from components such as stations, queues, servers and connectors and then analyse them using MARCA. XMARCA automatically generates the relevant MARCA files, including the *RATE* and *INSTANT* subroutines, and there is no need for user intervention.

1.5.3 SPNP

The Stochastic Petri Net Package (SPNP) [CMT89], developed at Duke University, models system behaviours with Petri nets models.

SPNP's Petri net models are specified in a language called "C-based Stochastic Petri Net" language (CSPN). Because CSPN can include C functions, the user can write models using very powerful features. The CSPN language has certain predefined functions intended for writing Petri net models, but has also the following extra functionality :

- Much of the behaviour of the model, such as transition rates, can be marking dependent and the enabling functions for transitions can be specified as complex boolean functions.
- The designer can give *assertions* which are checked for every state of the model. This helps the designers identify unexpected states as errors.
- The designer can set parameters of the model at run time. This allows variations of the model to be tested without recompiling.
- CSPN can define subnets with as C functions. These subnets can be used many times in the model with different parameters.

SPNP eliminates vanishing states from the reachability graph after it has been generated and uses the Gauss-Seidel and SOR methods (described in Section 5.6) to solve the steady state equations of the associated Markov chain.

SPNP also produces a wide variety of output measures as a result of its analysis. Tools often produce the steady state solution of the Markov chain and associated measures such

as the mean number of token on a place, but SPNP also allows for transient analysis and for cumulative measures. As an example of a cumulative measure, SPNP can calculate the sum of the steady state solution vector weighted by a reward vector. SPNP can also perform sensitivity analysis, by automatically testing the system with various to determines how some measure changes with respect to that parameter.

A drawback of the package is that the model is specified in the textual language and there is no graphical display for a modelling formalism which is naturally graphical.

1.6 Dissertation Roadmap

A guide to the subject matter of each chapter:

Chapter 2 gives the relevant definitions and mathematical theory underlying Markov chain analysis and the formalisms which may produce these models.

Chapter 3 describes the message passing libraries that give programmers the primitives they need to write parallel programs on workstations.

Chapter 4 presents the problem of state space exploration and methods for overcoming problems of large state spaces, including storage techniques and parallel algorithms. The results of the Paramaca parallel state space generator are given.

Chapter 5 discusses methods for solving the steady state equations and how they may be adapted to find parallel algorithms. The results of the parallel steady state solver are presented here.

Chapter 6 gives the conclusions and discusses future work.

Chapter 2

Background Theory

2.1 Introduction

This thesis concerns the generation and solution of Markov chains on a network of computers. A basic understanding of Markov theory is therefore required and presented in this chapter. As the theory is well covered in the literature, only the relevant major theorems will be given. A good reference for both Markov theory and numerical solution techniques for Markov chains is [Ste91].

The Paramaca state space generator takes a model file specifying a state transition system as input and produces the Markov chain describing the system. The state transition model is not tied to any particular modelling formalism, however formal description techniques can be a basis for state transition models. We give two formalisms for describing systems, Petri nets and queuing networks, and show how to derive a state transition model for the Paramaca analyser from each.

2.1.1 Notation

Throughout this thesis the following notation is used:

$x^{(k)}$ is the kth iteration of the value x or the value x at time k.

x^k is x to the power of k.

x_i is the ith element of x.

$P\{E\}$ is the probability of event E occurring.

$P, Q, A \cdots$ are matrices.

$\alpha, \beta \cdots$ are scalars.

2.2 Stochastic Processes

A stochastic process is a set of random variables that are defined on the same probability space, in other words they may take on only the same set of values. Let $\{X(t), t \in T\}$ be the set of states. T is usually thought of as the set of possible times and $X(t)$ as the state of the process at time t.

2.2.1 Homogeneity

A *stationary* or *homogeneous* process is one whose probabilities do not change with a shift of the time axis. The process must satisfy the following condition:

$$P\{X(t) \leq x | X(t_n) = x^{(n)}\} = P\{X(t - t_n) \leq x | X(0) = x^{(n)}\} \quad (1)$$

2.2.2 The Markov Property

A Markov process is a stochastic process which has the Markov property. In Markov processes the future states of the process depend only on the current state of the process and not on the previous states that it encountered. Hereafter it is always assumed that the Markov process is homogeneous.

Formally:

$$P\{X(t) \leq x | X(t_0) = x^{(0)}, \ldots X(t_n) = x^{(n)}\} = P\{X(t) \leq x | X(t_n) = x^{(n)}\} \quad (2)$$

The *sojourn time* is the time the process spends in a state. The distribution of the sojourn time must be *memoryless*. This means the time that the process spends in a state is independent of the time it has already spent there. If τ is the sojourn time and the process has already spent time s in the state then:

$$P\{\tau > s + t | \tau > s\} = P\{\tau > t\} \quad (3)$$

2.2.3 Discrete Time Markov Chains

If we consider the state of the process only at discrete time intervals then we have a Discrete Time Markov Chain (DTMC). The set of times, T, is usually taken to be the set $\{0, 1, 2, 3, \cdots\}$. In a DTMC the sojourn must be randomly distributed according to the geometric distribution.

Chapman-Kolmogorov Equations

Let us define a matrix P where the element $p_{i,j}$ is the probability of moving from state i to state j in a single step.

The matrix $P^{(n)}$ contains the probabilities of the process making n step transitions between states.

Then as the Markov chain is homogeneous

$$p_{i,j}^{(n)} = P\{X(m+n) = j | X(m) = i\} (\forall m, n \in T) \tag{4}$$

From the Markov property we obtain the following formula for the elements in the multi-step transition matrix by finding the sum of the probabilities of the process passing through the possible intermediate states.

$$p_{i,j}^{(n)} = \sum_{\forall k} p_{ik}^{(l)} p_{kj}^{(n-l)} \quad \forall (l)(0 < l < n) \tag{5}$$

These are the Chapman-Kolmogorov equations and they can be used to derive the multi-step transition probability matrix. In matrix form the matrix $P^{(n)}$ is determined from the single step matrix P and letting $\pi = (\pi_0, ...\pi_n)$ in matrix form :

Theorem 2.1 $\pi^{(n)} = \pi^{(0)} P^{(n)} = \pi^{(0)} P^n$

2.2.4 Irreducibility

Two states i and j *communicate* if there is a nonzero probability of the system moving from i to j and back again. In the graph of a Markov chain this is represented by the existence of a path in each directions between the states. A Markov chain is *irreducible* if

$$\forall(i,j)\exists(m)(p_{i,j}^{(m)} > 0) \tag{6}$$

Let $f_j^{(m)}$ be the probability of leaving state j and returning to state j after m steps.

Therefore the probability of ever returning to state j is

$$f_j = \sum_{m=1}^{\infty} f_j^{(m)} \tag{7}$$

The states of a Markov chain can be separated into those which the system is guaranteed to revisit, the *recurrent states*, and those which have a non-zero probability of never occurring again, the *transient states*. The system can pass through transient states repeatedly, but there is a chance that the system will never return to them. *Recurrent* and *transient* are defined by f_j

Definition 2.1 *For any state j:*

- *if $f_j = 1$ then state j is recurrent.*
- *if $f_j < 1$ then state j is transient.*

Definition 2.2 *If the Markov chain returns to a state j only at the regular time intervals $k\eta$ for all $k \geq 1$ and for some $\eta \geq 2$ then it is* periodic *with period η, otherwise it is* aperiodic.

Definition 2.3 *If state j is recurrent then the mean recurrence time is*

$$M_j = \sum_{m=1}^{\infty} m f_j^{(m)} \tag{8}$$

If $M_j = \infty$ then the state is recurrent null *otherwise the state is* recurrent non-null.

Theorem 2.2 *If a Markov chain is irreducible then the states are either*

- *all transient or*
- *all recurrent null or*
- *all recurrent nonnull*

2.2.5 Steady State Solution

Let π be a vector whose elements π_j give the probability of the process being in state j.

Definition 2.4 *π is the* stationary probability distribution *of a DTMC if $\pi = P\pi$.*

The limiting probability distribution of the Markov chain is

$$\pi = lim_{n\to\infty}\pi^{(n)} \tag{9}$$

if that limit exists. It is important to note that the limiting distribution may not always exist even if a stationary distribution does.

The next theorem shows the conditions for the existence of a limiting distribution:

Theorem 2.3 *For an irreducible, aperiodic and homogeneous DTMC a limiting distribution exists and is independent of the initial distribution. Also one of the following is true:*

- *All the states are transient or all states are recurrent null. $\pi = 0\ (\forall j)$ and there is no steady state distribution. In this case the state space is infinite.*
- *All states are recurrent nonnull and $\pi_j > 0\ (\forall\pi)$ and*

$$\pi_j = \frac{1}{M_j} \tag{10}$$

and the values in π can be found from the equations

$$\sum_i \pi_i = 1 \tag{11}$$

$$\pi_j = \sum_i \pi_i p_{i,j} \tag{12}$$

In matrix form Equation 12 can be written as:

$$\pi = \pi P \tag{13}$$

The solution π of this equation is the steady state solution of the Markov Chain.

2.2.6 Continuous Time Markov Chains

If the state of the process may change at any time then we have a Continuous Time Markov chain (CTMC). The set of times, T, is usually $\{t \in \mathbb{R}, t >= 0\}$. In a CTMC the sojourn time must be distributed according to the negative exponential distribution.

A homogeneous CTMC has its transitions represented by the infinitesimal generator matrix Q, where $Q_{i,j}$ represents the rate of transition between i and j. The sojourn time in state i is negative exponentially distributed with the parameter $-Q_{i,i} = \sum_{j \neq i} Q_{i,j}$.

This implies:

$$\forall i \quad \sum_i Q_{i,j} = 0 \tag{14}$$

Definition 2.5 *π is the* stationary probability distribution *of a CTMC if $\pi Q = 0$.*

Theorem 2.4 *For an irreducible, aperiodic and homogeneous CTMC a limiting distribution π exists and is independent of the initial distribution. Also :*

- *The limiting probability distribution is also the stationary probability distribution and can be determined from the following equations:*

$$\sum_i \pi_i = 1 \tag{15}$$

$$Q_{i,j} = \sum_{i \neq j} Q_{i,j} \pi_i \tag{16}$$

Equations 15 and 16 are called the *global balance equations* and may be written in matrix form as

$$\pi Q = 0 \tag{17}$$

which gives π as the steady state solution of the CTMC.

2.3 State Transition Formalisms

The previous section presented an introduction to Markov chains which can be used for performance analysis. Although the interface language used by DNAmaca and Paramaca is very general and can be used for any state transition system with the Markov property, it is useful to consider some formalisms which can produce state transition models for analysis.

2.3.1 Petri Nets

Petri nets were introduced by Carl Petri as a formal method for describing systems with synchronism and concurrency. There are many varieties of Petri nets, each with their own particular properties and uses. We will present place transition nets and generalised stochastic Petri nets (GSPN).

Place Transition nets

A place transition net consists of :

Places represented by circles.

Token represented by dots. Tokens are contained within Places. Tokens on a place represent certain conditions or quantities on the places.

Transitions drawn by rectangles. They act to remove tokens from places and place tokens according to a fixed rule.

Arcs are lines drawn between transitions and places to identify the places that a transition acts on.

When drawn with these elements, a place transition net is a bipartite graph. In other words no place may have an arc to another place and no transition may have an arc to another transition.

Definition 2.6 *A Place transition network is a 5-tuple* (P, T, I^-, I^+, M_0) *where*

- P *is a finite set of places.*
- $T : T \cap P = \emptyset$ *is a finite set of transitions.*

- $I^- : P \times T \rightarrow \mathbb{N}_0$ *is a backward incidence function. If* $I^-(p,t) > 0$ *an arc leads from place* p *to transition* t.
- $I^- : P \times T \rightarrow \mathbb{N}_0$ *is a forward incidence function. If* $I^+(p,t) > 0$ *an arc leads from transition* t *to place* p.
- $M_0 : P \rightarrow \mathbb{N}_0$ *defines the number of tokens on each place,* $M_0(p)$ *gives the number of tokens on place* p. *Every possible function from* P *to* N *is a* marking, *and* M_0 *is known as the* initial marking.

A transition t is *enabled* iff $M^k(p) \geq I^-(p,t)$ $(\forall p \in P)$. An enabled transition t may *fire* to define further markings $M^{(k+1)}(p) = M^{(k)}(p) - I^-(p,t) + I^+(p,t)$ $(\forall p \in P)$.

The above definition shows that the incidence function I^- determines which transitions may fire and the functions I^- and I^+ may be viewed as weights on the arcs which determine the number of tokens added and removed from the places.

Generalised Stochastic Petri nets

Place Transitions networks can specify states (markings) and transitions to form a state transition model for analysis. However they do not allow one to specify probabilities for the transitions or to give time distributions to transitions. This restriction is overcome by Generalised Stochastic Petri nets (GSPNs). [AMCB84]

A GSPN is a 4-tuple $PN, T1, T2, W)$ where

- $PN = (P, T, I^-, I^+, M_0)$ is a place transition net
- $T_1 \subseteq T$ is a set of timed transitions, $T_1 \neq 0$
- $T_2 \subset T$ is a set of immediate transitions, $T_1 \cap T_2 = \emptyset, T = T_1 \cup T_2$
- $W = (w_1, ...w_{|T|})$ is an array where w_i is either
  - a rate $\in \mathbb{R}^+$ of an exponential distribution specifying the firing delay, when a transition t_i is a timed transition.
  - is a weight $\in \mathbb{R}^+$ specifying the relative firing frequency, when transition t_i is an immediate transition.

GSPNs have two types of transitions, immediate and timed. If an immediate transition is enabled at a marking then all timed transitions are disabled. An immediate transition fires without any delay and the new marking is reached with no change in time.

Markings which enable immediate transitions are known as *vanishing* markings and those which enable timed transitions are known as *tangible* markings. Vanishing markings have no effect on the time spent in other markings and are often eliminated from the set of reachable markings.

Deriving state transition models from GSPNs

Each marking is a different state of the system and the future of the system depends only on the current marking. The time spent in each marking is exponentially distributed, which makes the system memoryless. With these conditions a GSPN is a Markov process and the set of reachable markings is the state space of the Markov Chain. Therefore GSPN models are Markov processes and can be written as state transition models for analysis by Paramaca.

The state descriptor is a vector $\{p_1, p_2, \cdots p_n\}$ of integers, where n is the number of places in the Petri net and p_i is the number of tokens on place i.

A transition t adds and subtracts from the elements of the vector as defined by the incidence matrix: $p_i^{(k+1)} = p_i^{(k)} - I^-(i,t) + I^+(i,t)$ $(\forall i \in P)$. The condition for transition t to fire is also defined by the incidence matrix, t may fire if : $p_i^{(k)} \geq I^-(i,t)$ $(\forall i \in P)$.

The initial state of the system is easily deduced from the initial marking. The conditions and effects of each transition can be written in the C-like interface language of model file from the forward and backward incidence matrices.

2.3.2 Queuing Networks

Many communicating systems lend themselves to a queuing network model. Queuing networks [BCMP75] have the following components :

Service Center A service center has a queue and one or more servers. The queue is used to hold customers who cannot be immediately served. The queue may be of infinite or limited capacity. If a customer arrives at an empty queue it will be immediately passed on to a server, otherwise it must wait until a server is free and removes it from the queue.

Customers Are objects which require processing by servers. Groups of customers may be given different behaviours by assigning them to different customer classes. Each class may queue according to its own discipline.

Customer Routing Matrices Once a customer leaves a server it may proceed to another queue. A routing probability matrix gives the probability of a customer moving to each service center after being served. Each customer class may have a different routing matrix. Some systems may allow customers to exit or enter the system. Such systems are called *open*, whereas systems that conserve the number of customers are *closed*.

Arrival and Service Discipline

The *input distribution* specifies how customers arrive in the system for the open customer classes. Customer arrival rates are given by a probabilistic distribution. The *service time distribution* specifies the time the servers take to serve a customer. The service time is also given by a probability distribution.

The *service discipline* specifies how customers arriving at service centers are assigned to queues and how servers select customers from the queues to serve. Four strategies are:

First-Come First-Served The customers are served in the order in which they arrive.

Round Robin The customers are served in rotation.

Pure Delay Also called *infinite server*, the customers are served immediately and experience only the delay of the service.

Last-Come First-Served, Preempt Resume The latest arrival is served first. Earlier customers being served are interrupted and resume service after the later ones have been served and left the service center.

Deriving state transition models from queuing networks

Classes of queuing networks with the product form property can be analysed with Mean Value Analysis or the Convolution algorithm. Sometimes the model contains elements which violate the product form property and Markovian analysis can be used as an alternative to Mean Value Analysis.

A queuing network may be cast as a Markov process by finding a representation for all possible states of the queuing system and ensuring that service and movements of customers are memoryless transitions.

The state descriptor must represent the number of customers present in each queue. For a multiclass queuing network this value must be recorded for each class. In this case the descriptor will be a vector listing the number of customers of each class in each queue.

Such a vector is sufficient when the service time distributions are exponentially distributed. As the exponential distribution is memoryless, no record of how long the system has been in the current state is needed. However if a Coxian distribution is used then the descriptor must also include the current phase of the service distribution. The full state descriptor will be the concatenation of all the descriptors for the individual queues and every possible configuration of the state descriptor represents a distinct state.

All possible transitions between the states must also be given. Each transition will remove a customer of a certain class from the queue and may place it in the queue of another service center. The routing matrix specifies which service centers may receive the customer. The transition is reflected in the state descriptor by changing the number of customers in descriptor elements corresponding to the relevant queues. The condition for a transition to occur is that the customer must be in the source queue in a position that may be served.

2.3.3 Direct Modelling

State transition models can also be created directly from a description of the system without being described by formal method. As an example consider a multimedia traffic switch in Figure 1 designed to handle two classes of traffic, delay-sensitive voice and delay-insensitive data [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.

The state descriptor for this system is a vector of 3 integers, $\{Data, Buffer, Voice\}$, representing the number of data calls, buffered data calls and voice calls in the system respectively. The state descriptor is initially $\{0, 0, 0\}$.

The following transitions can occur in the system

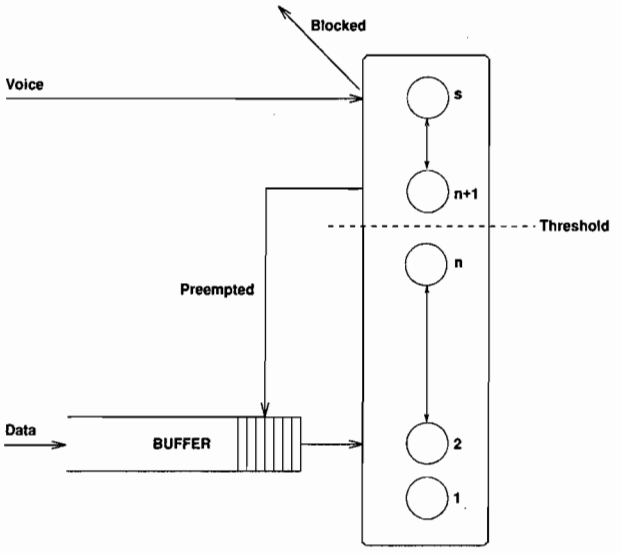

Figure 1: A multimedia switch for handling voice and data traffic

Data Service Subtract 1 from *Data*.

Buffer Dequeue Subtract 1 from $Buffer$ and add 1 to *Data*.

Voice Arrival

if $(voice + data \geq s)$ and $(Data > n)$ then{
    if $(buffer < b)$ then add 1 to $Buffer$
    Subtract 1 from *data*
    Add 1 to *voice*}
else if $(Voice + Data \geq s)$ and $(Data \leq n)$ then {/*discard*/}
else if $(Voice + Data < s)$ then {Add 1 to $Voice$}

Voice Service Subtract 1 from $Voice$.

Data Arrival Add 1 to $Buffer$.

Each of these transitions conditions can be expressed in C code together with the state descriptor and the initial state to form the model file for input to Paramaca.

Chapter 3

Message Passing Environments

3.1 Approaches to Parallelism

Parallel programs can be divided into three classes (sometimes called paradigms) based on their structure. Each of these paradigms is associated with a parallel programming method and we will briefly discuss and illustrate each method with an example of a program which produces a list of prime numbers. [CG90]

3.1.1 Specialist Parallelism

Programs written with the specialist paradigm are usually divided between processors so that each processor focuses on a specific task. The whole program is viewed as a network of cooperating tasks which work relatively autonomously, but provide each other with data messages when needed. Processor communication is by explicit *message passing.*

We can use the Sieve of Eratosthenes as a structure for a message passing program for finding prime numbers. Several processes are started and joined in a pipeline. Each process is associated with one prime number and is charged with the task of discarding which numbers from an incoming stream of data are divisible by a certain prime number. If numbers received by the process are not divisible by its number then they are passed on to the next process in the pipeline. If a number is not divisible by the prime held by the last process then it is a new prime. For each new prime found, a new process is created and added to the end of the process chain. This new process discards all numbers which are multiples of the new prime. One process generates a stream of numbers at the start of the pipeline. Figure 2 shows this method.

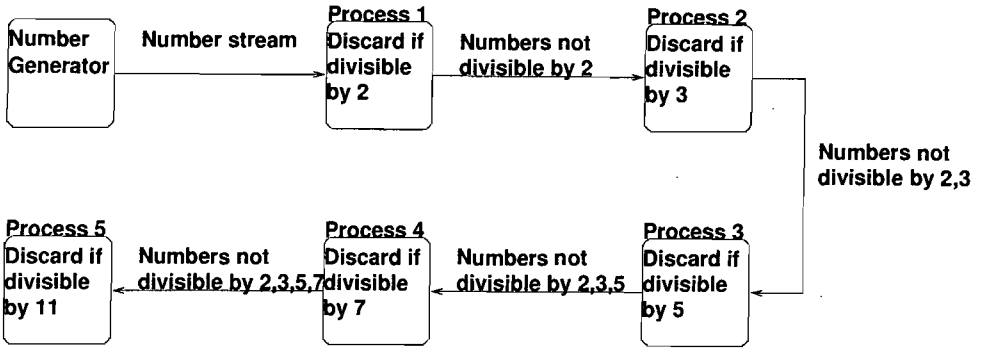

Figure 2: An example of specialist paradigm used to find prime numbers

3.1.2 Agenda Parallelism

A problem that can be partitioned into a number of smaller autonomous tasks can be structured as agenda parallelism. Each processor can perform any of the tasks and selects more work from the queue of tasks as soon as it is free. The processors communicate using a *distributed data structure*. Although the name, distributed data structure, suggests that this is a data structure divided between several computers, it is viewed as one data structure available to all the processors. There is no need for programs to explicitly pass messages to each other as all data is shared by all processors. This method clearly depends on having an underlying system to ensure the distributed data structure is up to date and consistent across the processors.

An agenda parallel version of a program to find prime numbers would assign a block of numbers (as an example the numbers from 1000 to 2000) to each process. Each process tests all the numbers in the block for primeness and reports any prime numbers it finds in the block. After searching the block a process would then start work on a new block.

3.1.3 Result Parallelism

Result parallelism is useful when the result of the program can be seen as a multi-element data structure. Each process is responsible for computing one of these elements and the process terminates when it has computed its element. This method is called a *live data structure* and also has no explicit message passing, the processes merely refer to other elements of the data structure to obtain the results of the previously terminated processes.

Prime numbers can be found using a live data structure by viewing the solution as a vector of bits where each bit indicates if the number of that bit position is prime. A process is

created for each bit position which tests the primality of just one number. When the process terminates, its answer is entered into that bit position. The results of the previous processes are used to find prime numbers for use in the computation.

The algorithm for each process considering number n and resulting in the vector *prime* is

Result Parallelism Prime Finding

```
lim = sqrt(n)+1;
for (i=2; i¡lim; ++i) {
    if (prime[i] && (n mod i) == 0) return 0;
}
```

3.2 Message Passing Libraries

The programs in this project all use message passing as the programming method and so fall into the specialist paradigm.

A message passing library is used to augment conventional sequential programming languages by providing the functions necessary to pass messages between different processes.

A typical message passing library provides :

- A means of starting processes on different machines.
- Point to point communication between the processes.
- Group communication including broadcasts and reduction operations.
- Higher level operations and other utilities. These vary from library to library but can include methods for profiling and debugging.

3.3 The PVM system

The Parallel Virtual Machine (PVM) [G+94] [GKP96] is a message passing library written for networks of computers with versions for a wide variety of architectures and operating

systems including Sun and Silicon Graphics (SGI) workstations and Cray computers. PVM is distributed with interfaces for C and Fortran.

PVM was developed in Oak Ridge National Laboratory for research on heterogeneous computing. It was made freely available and as researchers at various universities realised its utility it become popular for scientific computation.

As the the name implies, PVM presents to the developer a virtual machine where simple commands can send data between processes on different machines and even to processes on different architectures and operating systems. The PVM system is designed with an emphasis on portability, which allows applications to be written for different architectures, and interoperability, which allows applications on different systems to communicate.

A major feature of PVM is in the control of processes and hosts [GKP96]. Host machines can be added or removed and similarly, processes started or killed under the control of the application. This allows fault tolerant applications to be built and gives great flexibility in load balancing.

3.4 The MPI system

PVM was not the only message passing library developed. Several other groups developed libraries for different machines. Clearly there would be benefits in portability and compatibility of programs if a standard was made of the core functions that a message passing library should support and the interface for the library.

Vendors and researchers decided to draw up a standard for message passing libraries in the form of the Message Passing Interface (MPI) standard.[For94] The MPI standard was produced by the Message Passing Interface Forum and released in May 1994. Although MPI was created by a wide body of users, developers and researchers, no official standards body recognises MPI. MPI-2 is an update of the MPI standard that has been under development since the release of MPI-1.

The Message Passing Interface is a definition of an interface for message passing libraries. It lists the functions that should be supported and describes the interface to languages such as C and FORTRAN, but in contrast to PVM does not provide an implementation. The implementation is supplied by various vendors for their specific platforms, some free and others commercial.

Because each implementation can be written with a specific architecture in mind and the

MPI standard does not require interoperability, therefore implementors can make non-portable optimisations, and Massively Parallel Processing computer implementations are expected to be faster than PVM.

Free implementations of MPI include MPICH from Argonne National Laboratory and Mississippi State University, LAM from Ohio Supercomputer Center and CHIMP from Edinburgh Parallel Computing Centre. There are also commercial vendors of MPI implementations including Hewlett Packard, Silicon Graphics and Alpha Data.

3.5 Paramaca Requirements

Paramaca does not require the advanced features of either system. The ability to send and receive point-to-point and broadcast messages and to perform reduction operations is sufficient for the algorithms. Both environments support these features, therefore both systems are appropriate for Paramaca.

Paramaca was written in MPI, but as Paramaca uses only the basic features of each message passing system it was easy to write a set of macros to translate the syntax of the necessary functions of MPI to PVM. This system lets Paramaca be compiled with either library. The MPICH [GL] implementation of MPI and PVM version 3.3.10 were used as the libraries for Paramaca.

We chose to use PVM because the process startup system was more flexible and the Solaris implementation was more reliable. The complete version however can still run with either library.

Chapter 4

State Space Generation

4.1 Introduction

This chapter presents the serial state space generation algorithm, a parallel version which uses several workstations for the same task and the results obtained by the parallel method.

The function of the State Space Generator is to find and enumerate all the states that the system can reach. This set is called the *state space*. The generator also records which states can be reached from any particular state.

This information can be used to check that certain properties hold for the system

- That conditions always hold on the system.
- Structural analysis, for example testing for liveness.
- Performance testing.

The state space generator needs a model of the system as input to perform this task. This model consists of an initial state and a series of rules which specify how the system can move from one state to another

These rules must:

- determine what changes are possible for the system in this state (the transitions).
- how each transition changes the state of the system.

- be applicable to any state that may be generated.

The state space generator applies the rules to the initial state s_0. This reveals a set of transitions $t_1..t_n$ and the set of states $s_1..s_n$ which result from the transitions. These transitions are recorded in the form of a graph with arcs from each state to its successor states. It is likely that some states will have a transition leading to one of its ancestor states. The process of generating the successor states from a single state (s_i) is known as exploring state S_i.

The newly generated states are added to the table of known but unexplored states. Each new state that is found in this way is then subjected to the process of exploration itself until all states of the system have been explored.

As each state is explored, the information for that state (transitions and successors) is written to disk. The graph which is generated in this way can also be represented as an n by n matrix with each row having an entry in the column of its successor state. This state transition matrix will later be used for performance analysis by the steady state solver.

A problem with the state space exploration process is that there is usually no *a priori* guarantee that the particular model's state space is finite. It is quite possible that the state space will be unbounded and the state space generator will not terminate. Often it is only when the state space exploration is complete that can one can say the state space is bounded. If the state space is unbounded then state space exploration and the associated method of analysis is clearly inapplicable.

The other major problem in the state space exploration process is the state space explosion problem. Even simple models can generate many millions of states and their state spaces can easily exceed the memory of a workstation. This is called the state space explosion problem and means that the size of the system to be analysed is usually limited by storage requirements.

Two methods of resolving this problem are improved storage methods and increasing the memory available to the algorithm through parallelism. This chapter presents the state space exploration algorithm, the relevant state table storage techniques and reviews other work on parallel algorithms for this problem before describing the algorithm used in Paramaca.

4.2 Serial State Space Generation Algorithm

The state space exploration algorithm is similar to the traversal of a directed graph in a classical depth first or breadth first search starting from the initial state s_0. However the graph does not already exist in the computer's memory but is created by the exploration algorithm.

State Space Generator starts with a queue of unexplored states F, which contains only the starting state. A table of visited states E is kept to identify which states have been encountered previously before. The function succ(s) returns the set of successor states to state s. The state space exploration algorithm given in Figure 3 constructs the state transition graph A.

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

Figure 3: Sequential state space generation algorithm

The model may generate an unbounded state space, in which case the algorithm will not terminate, otherwise it will terminate when the queue F is empty, meaning all states have been explored.

The actual state space exploration algorithm as originally implemented by DNAmaca differs in some practical aspects. The state transition graph is written to disk in the form of a matrix file where the element $A_{i,j}$ is the rate at which the process moves from state i to state j. The output process can be simplified if the states are numbered and explored in a breadth first order, so DNAmaca uses a queue to store the unexplored states rather than

the stack which state space exploration algorithms traditionally favor. The state transition graph can be written to disk as it is produced, so there is no need to store it in primary memory. It is necessary to check if a state has been previously reached by the exploration process and to identify it by number. Table E is a hash table used to identify states that have been encountered.

4.3 Storage Techniques

A state table is used to check which states have been encountered before and to identify them by a state number. This state table must not only return results quickly but must also store this information in an compact way to maximise the size of the state space that can be generated. The need to return results quickly means that a hashing technique is usually employed to speed up the searching process.

4.3.1 Memory Allocation Strategy and Reliability

There are two approaches for providing memory for the state table. One is to *statically* allocate memory for the state table (usually whatever memory can be afforded or is available). This simplifies the memory management and makes the best use of the system's capacity, but carries some disadvantages: the memory could be a severe overestimation and result in a waste of resources, especially if the memory is shared with other users, or if the memory is an underestimate then the algorithm could fail. The *dynamic* approach is to allocate memory as it is needed, thus avoiding any unnecessary memory use, but there is still the possibility that the algorithm will fail if the primary memory is exceeded.

State space exploration techniques can also be categorised by their reliability as either *exhaustive* or *probabilistic*. The exhaustive methods store every state in the state table and they are guaranteed to find all the states in the system, but the need to store every state descriptor is very limiting as the state descriptors are often very large.

Probabilistic methods use hashing techniques to reduce the amount of memory that is needed to store the state space. It is possible that two distinct states may be represented in the state hash table in the same way. Should this occur, the state hash table will incorrectly report a state as previously visited when it is actually new. This error would add incorrect transitions to the state transition graph and lead to some states being omitted from the graph. This risk is acceptable, because it can be estimated and kept low enough that we can say with a high degree of confidence that states have not been missed.

4.3.2 Exhaustive Dynamic Methods

A simple list - MARCA

Certainly the simplest method of creating a state table is to store the full state description of all the states found in an array or linked list. This is the technique used in the MARCA Markov chain analysis tool. [Ste91, KS95]

However a linked list is inefficient in both the memory use and in the long time needed to check if a state is present. The size inefficiency is because the complete state description is stored for each state. The state descriptor size depends on the model, but it can easily be hundreds of bytes long.

If n is the number of states in the table, the mean time needed to check if a state is present is $\frac{n}{2}$ comparisons if the state is present otherwise n comparisons are needed to establish that the state is not in the table. The space used to store the states is dn bytes if d is the number of bytes per state descriptor and if the list is stored as an array.

Full state description with hashing - USENUM and DNAnet

The performance of a state table with full state information can be improved by using a hashing system.

USENUM [Scz87] and DNAnet [BD95] store the state information in an array of linked lists and uses a hash value calculated from the state descriptor to choose which linked list to check for the state. If r rows are used to hold the states then the time taken to check if the state exists will be $\frac{n}{2r}$ for a successful search and $\frac{n}{r}$ for an unsuccessful search. If h is the overhead per row the memory required for n states is $dn + rh$ bytes.

4.3.3 Probabilistic Static Methods

The probabilistic techniques do not keep the full state information but instead use various hashing techniques to indicate whether a state has been encountered before.

Bit state hashing

Bit state hashing [Hol95] does not store the full state descriptor, but instead uses only one bit to indicate whether a state has been inserted into the hash table or not. This technique tries to ensure full state coverage even with inadequate memory.

The method treats all memory available for the state hash table as a bit vector and the hash function points each state to the location in that vector. Before state space exploration begins, the entire bit vector is set to 0. As each state is found, the bit corresponding to the position of the hash value is set to 1. When the algorithm checks to see if a state has been encountered previously, the result is given by returning the bit at the requested state's hash value position.

A hash collision occurs if two states' hash values refer to the same bit position. If this occurs then a state will erroneously be reported as inserted. Keeping the chance of collision low depends on keeping the size of the table t large relative to n the number of states that need to be inserted.

The probability of there being no collisions is

$$p = \frac{t(t-1)...(t-n+1)}{t^n} = \frac{t!}{(t-n)t^n}$$

Leroy and Wolper [WL93] show that this is approximated by $p \approx e^{-\frac{n^2}{t}}$. This implies that t should by larger than $\frac{-n^2}{\ln(1-p)}$ which becomes very large for realistic values of n and small values of p.

Wolper and Leroy

Wolper and Leroy [WL93] recommend a *hash compaction* scheme which calculates a hash value for a state and then stores that hash value in a standard hash table with collision resolution.

Their analysis of this method shows that the probability of having no collisions is:

$$p = e^{-\frac{n^2}{2^k}}$$

where the hashed values are stored as k bit numbers. The memory required for this technique with m slots is $(mk + m)/8$ bytes.

4.3.4 Probabilistic Dynamic Methods

DNAmaca [Kno96] introduced a probabilistic method that was also dynamic. DNAmaca does not store state identifiers, but instead stores a hash value in a state hash table to

represent that a state has already been encountered. For each state two different hash values are produced from different hash functions.

One hash value (the primary) is used to determine which bucket should store the hash value. Each bucket is a linked list of the secondary hash values. These buckets are also stored in a linked list for efficient retrieval of the secondary hash values and to reduce the chance of collision between secondary hash values by separating them into different buckets.

This system is show in Fig. 4.

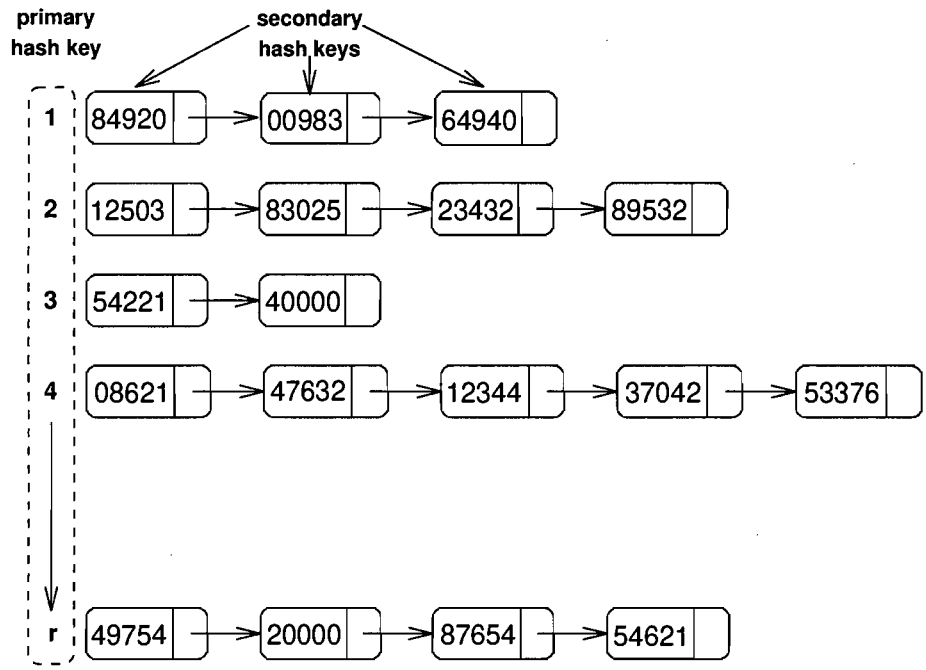

Figure 4: Hash table with compressed state information

The algorithm for using this state hash table is as follows.

- To check if a state is in the state hash table:
  - Calculate the primary and the secondary hash values $h1$ and $h2$.
  - Search bucket $h1$ for the value $h2$
  - If $h2$ is found then the state is said to be already entered in the state hash table.
- To enter a state in the state hash table:
  - Calculate the primary and the secondary hash values $h1$ and $h2$.

- Search bucket $h1$ for the value $h2$
- If $h2$ is not found then add $h2$ to the state hash table, otherwise the state is said to be already entered in the state hash table.

Thus two states s_i and s_j are treated as being the same if $h1(i) = h1(j)$ and $h2(i) = h2(j)$. If two different states happen to have identical values for $h1$ and $h2$ then a hash collision has occurred and that state will not be explored.

4.3.5 Reliability of the probabilistic dynamic state hash table

The analysis of the reliability and complexity is derived in detail in [Kno96] therefore only the final results will be presented here.

The probability of having no hash collision in any row in the state hash table is (i.e. of getting complete state coverage) :

$$p = \frac{(rt)!}{(rt-n)!(rt)^n} \tag{18}$$

where there are :

- r rows in the hash table.
- $t = 2^b$ where b is the number of bits in the secondary key.
- There are n unique states to be inserted in the state hash table.
- Each of the 2 hash functions distribute states to a uniform distribution and independently of each other.

We can use Stirling's approximation for $n!$ to write Equation 18 as

$$p \approx e^{-\frac{n^2}{rt}} \tag{19}$$

when $n^2 \ll rt$ (which will be the case in practical schemes with p close to 1), we can use the fact that $e^x \approx (1+x)$ for $|x| \ll 1$ to obtain the following approximation:

$$p \approx 1 - \frac{n^2}{rt} \tag{20}$$

The probability that a hash collision does occur is of course

$$1 - p \approx \frac{n^2}{rt} \tag{21}$$

Space complexity

The memory needed by this scheme is $M = hr + nb/8$ where h is the overhead bytes per row.

[Kno96] gives the following formulae for selecting optimal values for b and r to minimise the memory use with acceptable values of p.

$$b \approx \log_2 \left(\frac{h(n-1)\ln 2}{q} \right) + 2 \tag{22}$$

$$r \approx \frac{n(n-1)}{2^{b+1}q} \tag{23}$$

For an implementation it is impractical to design a scheme which changes these values according to the model under analysis. The number of states which the state space exploration will produce is usually unknown and in addition the size of the secondary hash values should be a whole number of bytes (preferably corresponding to the size of a standard data type) to avoid complex addressing schemes. The analysis in [Kno96] suggests that four or five byte secondary hash values are suitable for a range of state space sizes.

4.4 DNAmaca

DNAmaca [Kno96] is a serial state space explorer and general Markov chain analysis tool. The state space exploration module introduced the probabilistic dynamic storage of the state hash table as discussed in Section 4.3.4, and a steady state solver module implemented a several iterative methods for finding the steady state solution of the resultant Markov Chain.

DNAmaca performs state space generation and steady state solution of general state transition models on a single UNIX workstation with impressive performance and speed. The space saving techniques allowed it to generate state spaces of up to 1.27×10^7 states inside 64MB of memory while keeping the chance of omitting a state to less than 0.1%. The Paramaca tool is an extension to DNAmaca which provides parallel and distributed algorithms for state space exploration and for solving the steady state equations.

4.5 Other Work on Parallel State Space Exploration

S Allmaier, and G Horton present a shared memory parallel algorithm for state space exploration in [AH97]. They note that the "... memory and performance of a typical monoprocessor workstation ... imposes severe practical restrictions on the set of tractable models."

They propose various methods for dealing with the restrictions brought about by the state space explosion problem. These are:

- state space reduction techniques
- reducing model complexity
- sparse storage based on model invariants
- parallelisation

Parallelisation is preferred as it does not require the model to have any special properties nor does it restrict the power of the analysis. The paper presents a parallel shared memory algorithm for Generalised Stochastic Petri Net (GSPN) analysis by state space exploration.

The shared memory approach means that there is no need to partition the state space as must be done in the distributed memory version. This also brings the advantage of simplifying the load balancing problem. However, it does introduce synchronisation problems between the processors. Their state space exploration program was tested on a Convex SPP 1600 shared memory multiprocessor with 4GB of main memory. The state space as generated by this method is stored in a balanced search tree for rapid searching.

GSPNs introduce the problem of vanishing state elimination. Vanishing state elimination is the removal of states with instantaneous transitions in order to generate the continuous time Markov Chain. They implement both on-the-fly and post elimination of vanishing states. The shared memory parallel algorithm produces speedups for a range of numbers of processes employed and they report that the system can handle 4 000 000 states with 2 GB of memory.

S. Caselli, G Conte, and P. Marenzoni [CCM95] have also seen parallelisation as a way to overcome the problems of large state spaces. They offer two ways to parallelise the state space generation for massively parallel machines.

In the Data Parallel method a marking with t transitions is assigned to t processors. Each processor handles the firing one transition only and is responsible for determining the resulting state. This method was tested on a Connection Machine CM-5. This results in computation times linear in relation to the number of states and an increase in the size of the state space that can be analysed.

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. The algorithm as implemented did not include a distributed termination technique and this omission meant that excessive barrier synchronisation was needed to ensure proper termination. This synchronisation caused nodes to be idle even when they could be processing.

G. Ciardo, J. Gluckman, and D. Nicol's paper "Distributed State Space Generation of Discrete-State Stochastic Models" [CGN98] presents 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 between the processors and uses message passing to distribute the newly found states to their "owner processes". The algorithm also eliminates vanishing states. Unfortunately little information is given on the storage techniques used. The paper emphasised the importance of a hashing function which evenly distributes the states across the processors, but they try and achieve some degree of locality too. The method was tested on a network of SPARC workstations on an Ethernet network and an IBM SP-2 processor. In both cases a good reduction in the processing time was reported although with larger number of processes diminishing returns occurred.

There are also approaches that attack the state space explosion problem by exploiting symmetries, hierarchies or other special properties of the underlying model. These are useful, but the focus of this paper is using parallelisation as a technique for expanding the power of analysis without introducing restrictions. *Therefore the emphasis of this thesis is on the solution of general models.*

4.6 Parallel State Space Generation

4.6.1 The Parallel Algorithm

In the parallel algorithm each worker performs the state space generation algorithm as in Section 4.2. The processes on each of the participating machines still have the same data

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

Figure 5: Parallel state space generation algorithm for node i

structures, the unexplored state queue F_i and the state hash table E_i. However each worker is only responsible for a portion of the state space because the state space is partitioned between the workers. The new algorithm is given in Figure 5.

The state space is partitioned by a simple partitioning function $part(s)$. This function is given a state descriptor and returns the number of one of the processes. This state is then assigned to that process. Worker i may only have the states $\{s : part(s) = i\}$ in its queue of unexplored states and in its state hash table.

4.6.2 State Communication

When the successor function in a process i returns a state s' that is in the state partition of another process k, the process i will be unable to check if the state has been encountered before. Furthermore the unique number identifying the state is unknown which prevents the matrix row from being written to disk, this information can only be found by checking the information in the state hash table of process k.

If the worker finds states that belong to other processes it sends them to the relevant processes as determined by the partitioning function. Conversely each process will be receiving states belonging to it from the other processes. Each of those states must be checked against the state hash table and, if not present, added to the state hash table and the queue of unexplored states.

As our algorithm is written for a distributed memory message passing architecture it is similar to that of [CGN98] which is also designed for a network of workstations. However there are important differences in the communication between the processes. In the above algorithm states belonging to other processes are sent to their owners as they are discovered and each state is written out to the matrix file with its children's state identity numbers. This means that the matrix is written out in the format of ordered rows. However the local process does not know the state identity numbers of the states of other processes.

There are two ways around this problem.

- Continue to write out the matrix in row format. When a process gets a state from another process it should identify it in the state hash table and then return the state identity number to the process that originated the sending message. Each process then receives the identity of the states it sends to other processors in this manner and can fill in the entries in the rows to be written to the matrix file. This requires that

the matrix rows must wait until they are complete before they can be written out of memory. [CGN98] identifies this as "a more complex protocol".

- Decide that the matrix will have the entries identified by column. This removes the need for state identifiers to be sent back to the process that finds them. Each process can write out the entries of the state transition matrix listing only the transitions which terminate at states owned by the local process.

Although [CGN98] chose the latter solution, we chose the first method for the following reasons:

- It ensures the rows can be written out as they are found and in order. The column method requires the columns either to be stored in memory or to be written out of order to the matrix file.
- Consistency with the input formats of the steady state solver and functional analyser in DNAmaca.

A graphical description of the communication involved in the state identification protocol is in Figure 6.

4.6.3 Detecting Termination

For a process in the distributed algorithm to terminate, it is not sufficient merely to check if the queue of unexplored states is empty as other processes can send messages to the local process which would restart processing. A more complex method is needed to decide if the state space exploration is complete.

The exploration is only complete if all processes empty their queues of unexplored states and become idle. This problem is tricky for if the master process polls all the nodes to check if they are idle, the test still might fail. After the master has polled one node, say i, and found it to be idle, another node $j > i$ could send a message to i which prompts it to do more processing. Process j might then be idle when polled and all processes could report themselves as idle. This naive algorithm can easily reach the incorrect conclusion that the program is complete while process i is activated by receiving a message.

One solution to this is given by Dijkstra, the Circulating Probe Algorithm. [DFG83] The Circulating Probe Algorithm works by having each processor assigned a number from 0

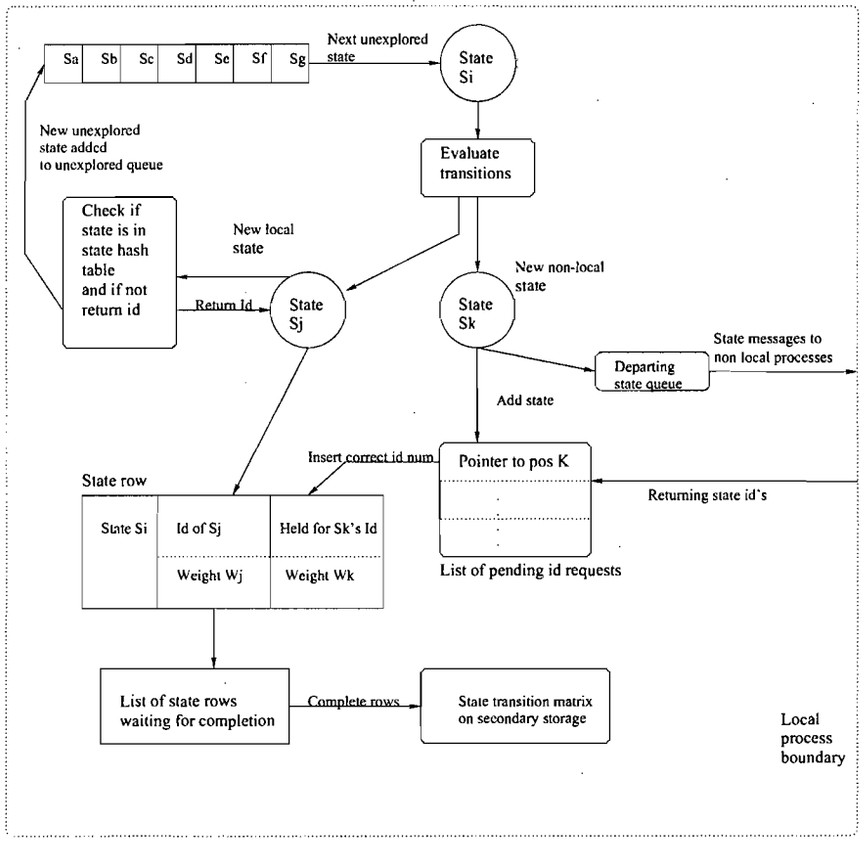

Figure 6: Information flow and structure of the Parallel State Space Exploration Algorithm

to $n-1$ and process 0 is viewed as the master which initiates the probe. The probe is performed by sending a token message to the processes in the form of a ring. Process 0 sends the token to process $n-1$ and as any other process say i passes the token to process $i-1$. Additionally all the processes and the token are regarded to be in one of two states labeled *black* and *white*.

The token is passed according to the following rules, but the token signalling is not counted as a message for the purposes of these rules:

1. The master process starts a probe by declaring itself *white* and sending a *white* token to process $n-1$.
2. A process i receiving the token will hold the token until it is idle, at which point it will send the token to the process $i-1$.
3. Any process which sends a message turns *black*.
4. A *white* process will pass on the token with the token colour unchanged, but a *black* process will turn the token *black*.
5. After a process has sent on the token, the process is bleached.
6. When the token is returned to the master process the following two cases may occur:
   - The master process is *white* and receives a *white* token in which case it may conclude the program is terminated.
   - The master process is *black* or the returned token is *black* in which case the probe is inconclusive and must be restarted.

Assumptions of the algorithm:

- The processes are either in an idle or active state.
- A process which is idle may only become active upon receiving a message.
- A process which is active becomes idle when it has completed its processing.
- No idle machine may send a message.
- Message passing is instantaneous.

In the state space exploration program all the assumptions hold with the exception of instantaneous message passing because the message passing libraries obviously cannot guarantee that the messages will arrive instantaneously. However if a slight delay is introduced between each probe by the master it is unlikely that a message from process i to process j will be delayed long enough for all the probes to complete and then reappear to restart processing on j.

For the algorithm to fail in this way would require that a message take longer than k token hops plus any delay induced by the master. The value of k depends on the rank of the sending process and the position of the token, but even in the worst case it is guaranteed that $k \geq 2$.

Therefore, with the proviso about delayed messages, the Circulating Probe Algorithm is suitable for detecting the termination of the parallel state space exploration algorithm.

4.6.4 Partitioning Function

During the state space exploration process each processor must avoid duplicating works by exploring states that other processes have explored but still ensure that no states are neglected. This demands an unambiguous way of partitioning the state space between the processors. The solution is to have a simple function that would determine the which partition and therefore which process owns the state. This decision should depend solely on the information stored in the state descriptor to avoid the impracticality of passing queries between processes.

A sound partition function should give the processes an even balance of states as any imbalance would leave all the machines waiting for the most heavily loaded process which would finish last. We should have a quantitative measures of partition function performance. For spatial balance we can measure imbalance by the value $imbalance = \frac{nS_{max}}{S_{total}}$, where S_{max} is the number of states allocated to the process with the largest number of states, and n is the number of processes. That is the maximum partition size divided by the ideal partition size. The ideal partition would give a spatial imbalance measure of 1 with imbalanced partitions giving larger values.

For a measure of spatial imbalance it would be tempting to use

$$imbalance = S_{total} - \sum_{i=0}^{n-1} S_i$$

or the standard deviation of the partition sizes. However, with all other factors being equal, the critical memory consumption and total run time are determined by the machine with the largest state partition. The run time would exceed that of the balanced distribution in proportion to the excess states on the process with the largest state load, and therefore the given measure of spatial imbalance gives a good indication of the effects of partition imbalance.

[CGN98] has investigated the question of partitioning functions for state space exploration and tried to find functions that provide locality as well as balance. By locality we mean that states should be generally assigned to the same partition as their children. This reduces the number of messages and state identity replies that need to be sent between processes. The locality can be measured by the number of states sent from one process to another, in other words the number of *crossarcs* between the state partitions.

By using a hash function on only a subset of the state descriptor, called the control set, many of the transitions do not alter the control set and do not add crossarcs from one partition. By appropriate selection of the control set, [CGN98] it is possible to reduce the number of crossarcs between the processes and lower the communication costs of the algorithm. Unfortunately this may come at the expense of imbalance in the state distribution and at present there is no way to automatically select a control set which will reduce cross arcs without partition imbalance.

A hash function is a good start for a partition function as a good hash function results in a uniformly random distribution which automatically brings spatial balance. The state space generation algorithm already computes hash values for the state hash table for every state so using either the primary or secondary hash values for the partition function would be computation saving.

Our measure of spatial imbalance might be criticised as overly stringent. If one uses a hash function it is unreasonable to expect a perfect balance as this would be expecting better performance than from a purely random function. However we should leave open the possibility that for certain models with special properties, a hash function can be written which would perform better than a random function.

The spatial imbalance of the partition function (using the secondary hash function to partition the state space between five processes) is shown below for various numbers of states.

States	Spatial Imbalance
172	1.33721
2 359	1.09580
11 386	1.07808
32 653	1.03957
71 560	1.03207
13 3507	1.01725
223 894	1.01807
348 121	1.00939
511 588	1.00951

Using the primary hash function as the partition function results in some rows of the hash table being empty in each process, while using the secondary hash value for partitioning results in the secondary hash key entries of the state hash table being restricted to a certain range. In both schemes the state hash table has the same reliability as the state hash table would have when run on a single process. If a third independent hash function is used for partitioning then the full range of rows and secondary hash keys are used in the state hash tables and collision on all three hash values is needed for a state to be missed. This would result in increased reliability, but requires computation of an extra function.

4.6.5 Testing Methods

The parallel state space generator was tested on a local area network of Sun workstations linked by 10Mb/s Ethernet. Unfortunately this environment provides neither homogeneous machines, nor the ability to configure a cluster of workstations into a parallel processing cluster. This means that the effect of adding an extra processor to the system may not show a smooth trend, as each processor may have a different CPU speed, different amount of main memory and different hard disk access times.

It may also be seen as an advantage that Paramaca was tested in this environment as this kind of ad hoc network is common in both universities and businesses. These results show that the algorithm works and is beneficial even on systems which do not have dedicated parallel computers, and for people who probably had not thought that their local area network could be used as a parallel computer.

The machines in the in which order they were added are:

Model	Memory
Sun Classic	64MB
Sun Classic	40MB
Sun Sparc 10	64MB
Sun Classic	40MB
Sun Classic	40MB
Sun Classic	32MB

The state space generator algorithm was tested on two models:

Flexible Manufacturing System (FMS) A model used in the paper by Ciardo [CGN98] which is derived from a Petri-net model.

BenchProd The model used by Casselli [CCM95] which is also based on a Petri-net model.

Both of these systems have the advantage that a parameter in the initial state vector can be adjusted to readily provide models with various state space sizes. With the adjustable parameter (k) they produce state spaces of the sizes in Figure 7.

Parameter k	FMS model size (states)
1	54
2	810
3	6 520
4	35 910
5	152 712
6	537 768
7	1 639 440
8	4 459 455

Parameter k	BenchProd model size (states)
1	172
2	2 359
3	11 386
4	32 653
5	71 560
6	133 507
7	223 849
8	348 121
9	511 588
10	719 695
11	977 842
12	1 291 429

Figure 7: State Space sizes of FMS and BenchProd models for parameter k

The tests of the parallel state space generator (and later the steady state solver) use a set of processors, where each processor is a UNIX workstation on an Ethernet LAN. As exactly one process is always run on each machine, the terms "process" and "processor" and "machine" are used interchangeably. Each model was as tested with 1 to 5 processes in the process set and for various values of k. Each test was run 5 times and the mean results was plotted with 95% confidence intervals.

4.6.6 Time Results

The results reported as "CPU time" are the mean CPU seconds used by the processes for the state space generation. This has the advantage that it only includes the time spent on computation and therefore clearly shows the effects of parallelisation on the processing time required for solution.

The results reported as "elapsed time" are the real times from the start to the finish of the state space generation, and includes the communications overhead as well as the CPU time. Naturally this is what is most relevant to the user of the system, as it is the time he must wait for the solution. However it is subject to the unpredictable delays of network traffic and output devices.

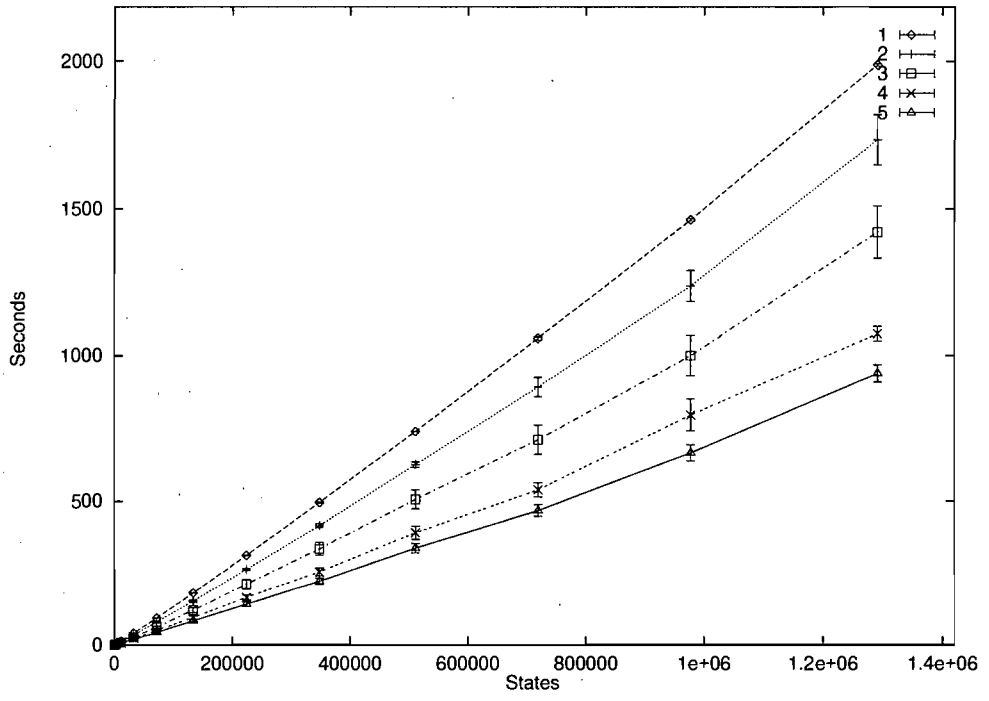

Figure 8: CPU times for the BenchProd model with 1-5 processes.

The CPU times recorded for both the BenchProd and FMS model are shown in Figures 8 and 9. Both models show that the CPU times increase smoothly and linearly with increasing model size for all process set sizes.

The parallel state space generation algorithm gives a clear reduction in the CPU times as more processors are added to the system. However as the number of processors increases,

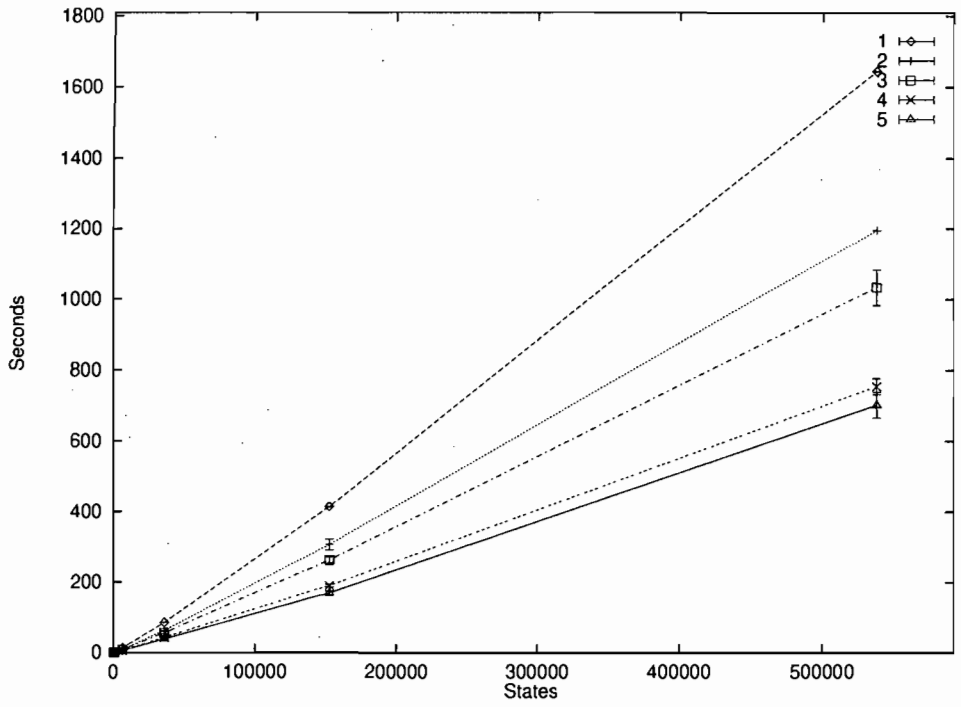

Figure 9: CPU times for the FMS model with 1-5 processes.

the efficiency of the parallel algorithm drops. The efficiency for the BenchProd model with 977 842 states is indicated below for different process set sizes.

Number of processes	CPU time (seconds)	BenchProd Efficiency
1	1462	100%
2	1237	59%
3	999	49%
4	796	46%
5	666	44%

The elapsed times for the models are shown in Figures 10 and 11. The elapsed time also shows a speedup for both models, but as this includes the time the processors spend waiting for messages the speedups are not as good as for the CPU times. The efficiency of the algorithm as measured by the elapsed times for the FMS model with 537 768 states is shown below

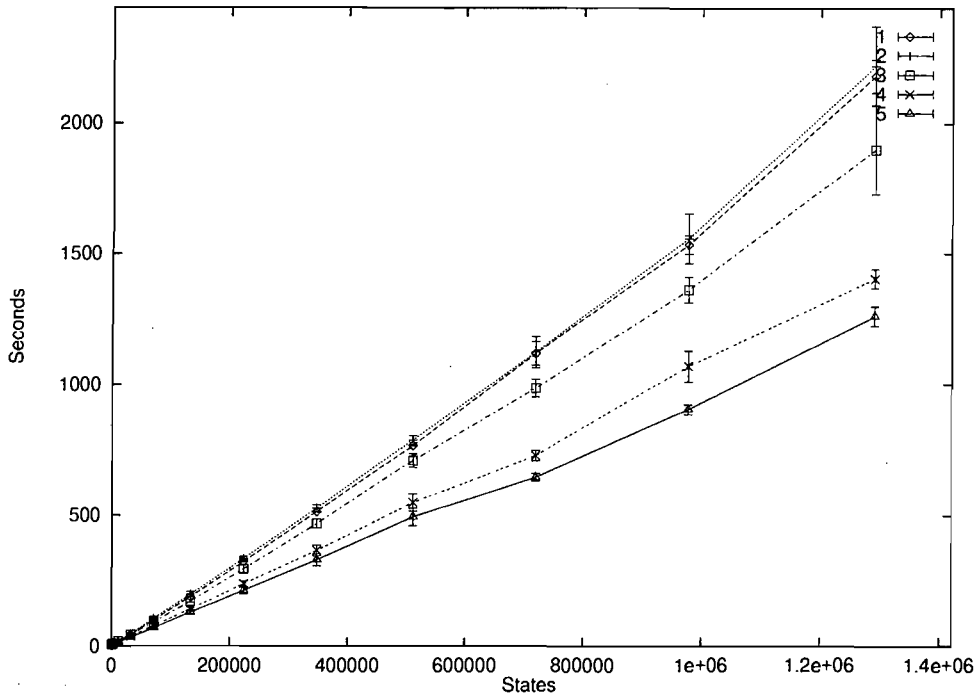

Figure 10: Elapsed times for the BenchProd model for 1-5 processes.

Number of processes	Elapsed time (seconds)	Efficiency
1	1810	100%
2	1425	63%
3	1270	47%
4	938	48%
5	890	41%

4.6.7 Space Results

The parallel algorithm divides the state hash table between the processes, which reduces the memory requirements of each machine so that state spaces can be generated using less memory on each machine. The memory usage for the two models in Figures 12 and 13 is the mean memory usage of all the processes used for the state space exploration. This memory usage consists of a base overhead which comes from running the program, memory for the state hash table which is proportional to number of states held, and memory for the unexplored queue and the state rows waiting to be written to disk. The memory requirements for the last two data structures depend on the exact timing of the messages

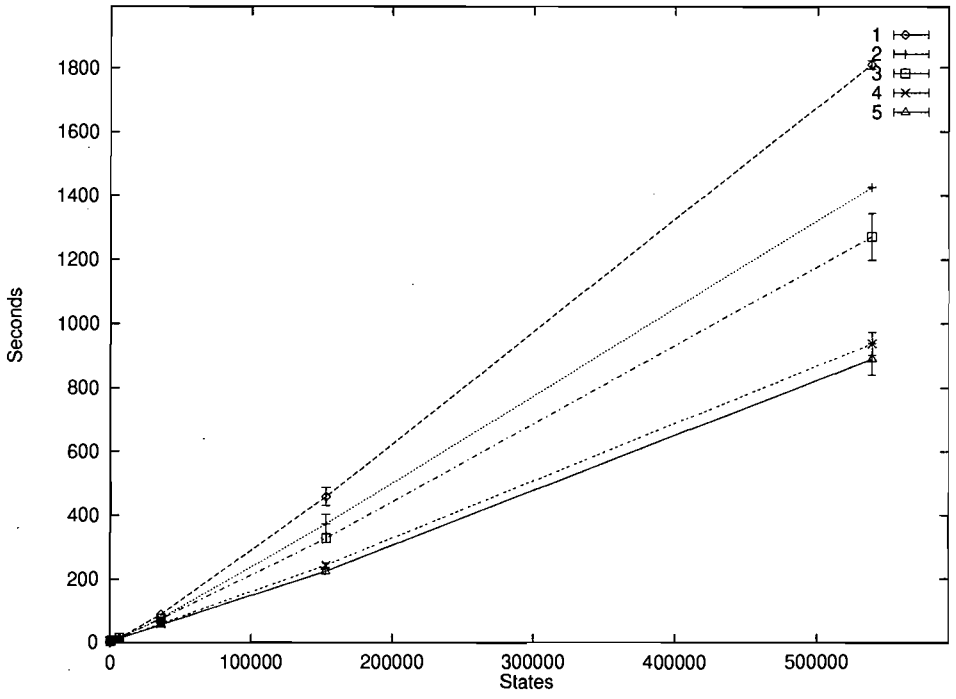

Figure 11: Elapsed times for the FMS model for 1-5 processes.

containing states and state identifiers. Therefore the memory consumption for each machine may vary even when running the same model on the identical set of workstations.

Only a portion of the state hash table is stored in each process, therefore the memory used for the state hash table increases at a slower rate than it does in the serial case. The parallel algorithm has the memory overhead of the message passing buffers, and for small state space sizes that overhead exceeds the saving of a smaller state hash table and memory consumption is increased. For models with state space sizes over approximately 250 000, memory savings occur and the memory use of the parallel version is lower than than that of the serial version.

After the submission of this thesis a conference paper was written by William Knottenbelt, Mark Mestern, Peter Harrison and Pieter Kritzinger which presented further results which show the success of parallel methods for state space generation. A parallel generation of a state space with 111 414 940 states took 55 minutes on an Fijutsu AP3000 parallel computer with 12 processing nodes. See [KMHK98] for details.

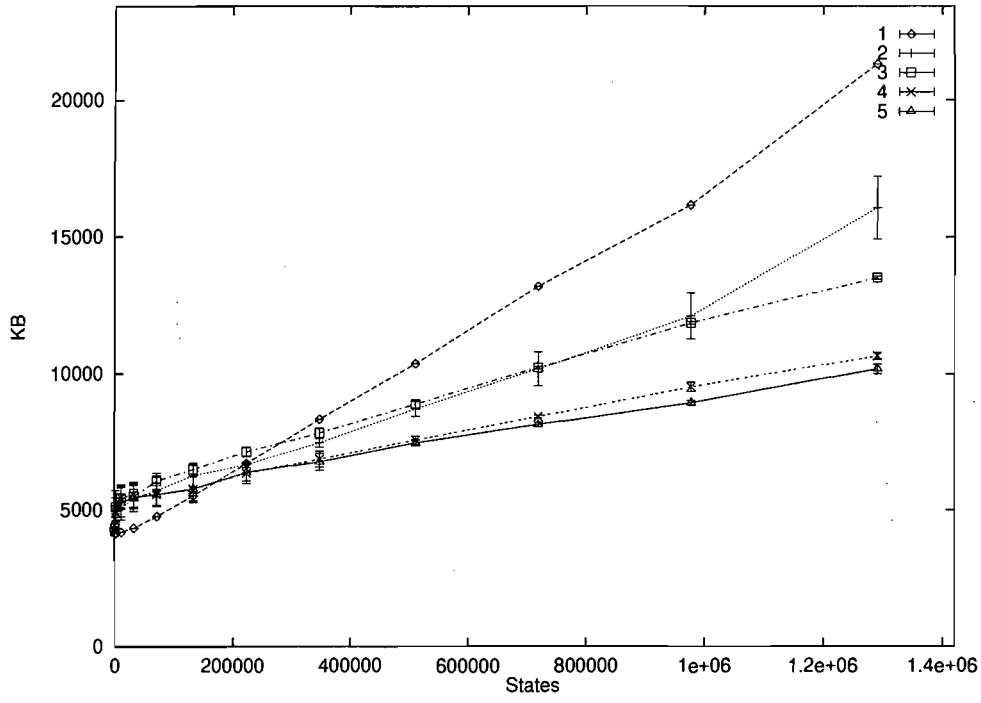

Figure 12: Memory usage for the BenchProd model for 1-5 processes.

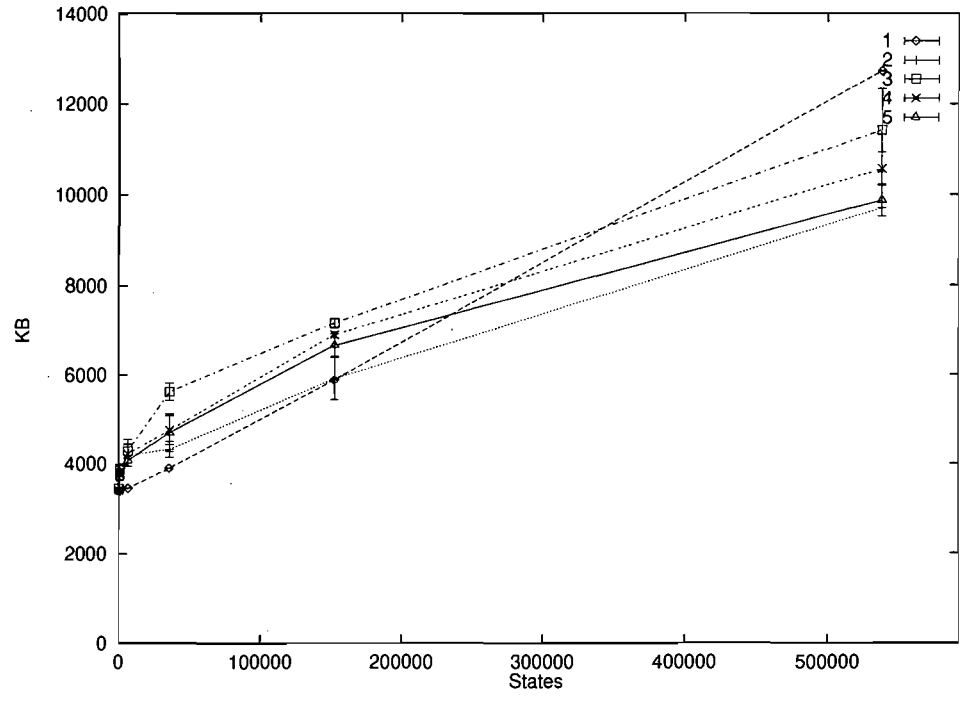

Figure 13: Memory use for the FMS model for 1-5 processes.

Chapter 5

The Steady State Solution of Markov Chains

5.1 Introduction

This chapter discusses steady state solution of the Markov chains. A steady state solution is only possible if the Markov chain is irreducible, therfore this chapter starts with a section considering the problems of functional analysis. We then discuss the classical and iterative techniques for solution and show how some of them can be rewritten as parallel algorithms. The chapter ends by giving the results obtained by the parallel methods.

5.2 Functional Analysis

As noted in Theorem 2.3, the Markov chain found from state space exploration must be irreducible and aperiodic for it to have a steady state solution. The functional analysis of the reachability graph to ensure it is irreducible should therefore be a prerequisite to solving the steady state equations. A Markov chain is irreducible if every state can be reached from every other state. This is the case if the state transition graph is a single strongly connected component.

The state transition graph was generated by exploring all states reachable from the initial state, therefore the graph already has the property that every state can be reached from the initial state. This fact can be used to simplify the process of checking that the graph is a single strongly connected component.

A strongly connected component test can be performed in parallel by the following method

- Read in the state transition graph with the arcs reversed and partition the graph between the processes.
- Perform a parallel depth first search of this reverse state transition graph, using the initial state as the root node. [KGGK94]
- All states found in this way are marked as being in the same component as the initial state. They are capable of reaching the initial state in the original state transition graph.
- If all states in the graph are marked at the end of the depth first search then they can all reach the initial state and the Markov chain is a single strongly connected component irreducible.

If some states are not marked after this search then the Markov chain contains some transient states. There is still the possibility that the Markov chain contains a single final strongly connected component that will be reached after the process passes through a set of transient states. We can check for such a case using the strongly connected components algorithm in [Baa88]. The transient states have no effect on the final solution, therefore if a single irreducible subgraph is found, then the transient states may simply be eliminated and the steady state solver can proceed.

The serial algorithm in [Baa88] for finding strongly connected components has time complexity is $O(e)$ where e is the number of edges in the graph. However the author is unaware of any parallel graph algorithms that perform the same task.

A related problem is that of finding the transitive closure of the graph of which the detection of strongly connected components is a special case. If the transitive closure contains all possible edges then the graph is a single strongly connected component. Although parallel algorithms do exist for the transitive closure algorithm they are all $O(n^3)$ in time complexity and $O(n^2)$ in space complexity which is unacceptable for the size of n that the state transition graphs have.

DNAmaca has a functional analyser which checks to see if the reachability graph has a single strongly connected component using the [Baa88] algorithm. If so, then it is irreducible or can be made irreducible by a simple removal of the transient states.

In Paramaca the function analyser will do serial functional analysis if the machine has sufficient memory. Otherwise it simply combines the state transition matrices from the different processors into a single consistent state transition matrix file.

5.3 Steady State Solution

After the functional analysis we have an irreducible state transition graph stored as a matrix. We now wish to obtain the limiting distribution for the system as defined by

$$\pi Q = 0$$

where $\pi = (\pi_1, \pi_2, ...\pi_n)$ and π_i is the proportion of time the system spends in state i for the Continuous Time Markov chain case;

or as

$$\pi P = \pi$$

in the Discrete Time Markov chain case.

The equations above are also solved subject to $\sum_{i=1}^{i=n} \pi_i = 1$ as the sum of proportions of the times must be 1.

These problems can be converted into a standard system of linear equations by rewriting as follows

$$(I - P^T)\pi^T = 0$$

and

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

in the DTMC and CTMC cases respectively.

The produces a system $Ax = 0$ where the matrix A is singular. A variety of methods are known for solving such more general systems of $Ax = b$.

These can be divided into two general approaches.

Direct methods The direct methods are a fixed sequence of manipulations of the matrix and of the solution vector to find the result and are guaranteed to finish. However the manipulations of the matrix causes large portions of the matrix that were previously zero to be filled-in. This destroys the sparsity which allows the process to store such large matrices compactly. This memory problem means the direct methods are often inappropriate for solving the steady state equations unless the system is fairly small. Nevertheless we will review them for completeness as some direct methods are included in DNAmaca. As they are unsuitable for large systems no attempt has been made to implement parallel versions of direct methods.

Iterative methods These methods start with an initial solution vector and then use the matrix to adjust the solution vector to reduce the difference between the solution vector and the actual solution. Over several iterations the solution vector be made to converge to the actual solution to within a required degree of accuracy. These methods are generally faster than the direct methods for large matrices, but they are not always guaranteed to converge or to converge within any particular time. Ill-conditioned systems may result in particularly poor performance. Iterative methods also have the advantage of never modifying the matrix A, which means there need be no accumulation of roundoff errors from the computation.

5.4 Direct Methods

Direct methods compute the solution in a fixed number of operations. The number of operations depends on the size of the matrix, but it is guaranteed to complete within a time that is known in advance.

Direct methods usually involve additions, multiplications and divisions to eliminate many nonzero entries in the matrix in a systematic way. However in the process of doing so, many zero entries may become *filled in* with nonzero values. Even although this is only temporary - if the original matrix was sparse then the fill-in can increase the memory consumption many fold and exceed the available memory. This is a major problem that prevents these methods from being used on large sparse systems.

In the direct methods many calculations are made with results which are carried forward to be used again as a basis for further calculations. Unfortunately the computer is unlikely to represent all the values exactly at every stage, therefore truncation error will occur and

the following calculations will be done with the erroneous values. After this has occurred repeatedly the error is compounded and the results become inaccurate.

5.4.1 Gaussian Elimination

Gaussian elimination is the best known direct method which solves the systems of the form $Ax = b$.

Manipulations of A must be followed by the corresponding manipulations of b for b to contain the solution vector. This may be conveniently implemented by treating b as an extra column of the matrix.

Gaussian elimination consists of two phases. The first phase is *reduction* which leaves the matrix in upper triangular form. The elements on the diagonal are called the pivot elements and each is used in turn to eliminate the nonzero elements beneath it. This is done by adding the necessary multiple of the pivot element's row to each row beneath it.

It may be necessary to exchange rows so that all the pivot elements are nonzero. For certain problems it may also be desirable to make the necessary exchanges in rows and columns so that the pivots are larger than the elements below them in order to make the algorithm stable.

This matrix shows A and b the solution vector during the reduction phase

$$\left(\begin{array}{ccccccc|c} a_{1,1} & a_{1,2} & & & \cdots & & a_{1,n} & b_1 \\ 0 & a_{2,2} & & & \cdots & & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & & & & \vdots & \vdots \\ 0 & 0 & \cdots & a_{i,i} & a_{i,i+1} & \cdots & a_{i,n} & b_i \\ 0 & 0 & \cdots & 0 & a_{i+1,i+1} & \cdots & a_{i,n} & \vdots \\ \vdots & \vdots & & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & a_{n,i+1} & \cdots & a_{n,n} & b_n \end{array}\right)$$

The second phase is *backsubstitution* which leaves the matrix as an identity matrix and the solution vector in b. The substitution process works from bottom up by using the pivot elements to remove the entries in the upper triangle of the matrix. Once the reduction and the backsubstitution of the pivots below the current row is complete, the current pivot is the only nonzero element in the row and can therefore be added to the elements above to eliminate them without fill-in.

An attempt at Gaussian elimination on the matrix Q will fail to produce a unique solution because the matrix A is singular. There are two ways to overcome this and these must be done after the reduction phase

Replacing the last equation The problem can be solved by remembering we actually have another equation, as the system must be solved subject to $\sum_{i=1}^{n} \pi_i = 1$. This equation can replace the last row of the matrix and the last element of b to produce a nonsingular system from which Gaussian elimination will produce a unique solution. Although convenient analytically, this is not recommended for numeric computation because it will carry the rounding error in x_n to all the other rows during the back-substitution.

Removing an equation Ignore the last row and declare $x_n = 1$. Any nonzero value could be used, but 1 is the most convenient for calculation. After carrying out the backsubstitution b will be the solution vector. However the vector must be normalised afterwards to make it sum to 1.

In the middle of the backsubstitution the matrix will look as follows

$$\left(\begin{array}{ccccccccc|c}
a_{1,1} & a_{1,2} & & & \cdots & & & a_{1,n} & b_1 \\
0 & a_{2,2} & & & \cdots & & & a_{2,n} & b_2 \\
\vdots & \vdots & \ddots & & & & & \vdots & \vdots \\
0 & 0 & \cdots & a_{i,i} & a_{i,i+1} & \cdots & & a_{i,n} & b_i \\
0 & 0 & \cdots & 0 & a_{i+1,i+1} & \cdots & & 0 & b_{i+1} \\
\vdots & \vdots & & \vdots & & \ddots & & \vdots & \vdots \\
0 & 0 & \cdots & 0 & \cdots & & a_{n-1,n-1} & 0 & b_{n-1} \\
0 & 0 & \cdots & 0 & \cdots & & 0 & a_{n,n} & b_n
\end{array}\right)$$

Gaussian elimination has time complexity $O(n^3)$ and space complexity $O(n^2)$.

Gaussian elimination can be done on a parallel computer [DHv93] by noting that there is parallelism in the row updates, as all elements of the row can be calculated independently of each other. Gaussian elimination is not used in Paramaca because the fill-in of the matrix makes it unsuitable for large systems.

5.4.2 LU Decomposition

LU decomposition [GW89] factors the matrix A into upper and lower triangular matrices U and L. One of these matrices has all the diagonal elements equal to 1. Having the unit diagonal in the L matrix is known as Doolittle's method, and in the U matrix is the Crout or Cholesky method.

The formulae for deriving upper and lower matrices with Crout's method are

$$\begin{array}{lcll} l_{ij} & = & a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj} & j \leq i \quad i = 1,2,...,n \\ u_{ij} & = & a_{ij} - (a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{jk})/l_{ii} & i \leq j \quad j = 2,3,...,n \end{array} \tag{24}$$

Now consider that we started with

$$Ax = b$$

and

$$LUx = b$$

which can we rewritten as

$$Ly = b \tag{25}$$

where

$$Ux = y \tag{26}$$

Therefore we can solve for x by solving Equations 25 and 26 which involves a series of substitutions.

LU decomposition has the advantage that once the decomposition of the matrix A is found then it is computationally cheap to compute the solution for further equations having the same matrix and a different right hand side. As an example once $Ax = b$ has been solved then the same LU decomposition may be used to solve $Ax = c$.

5.5 The parallel operations/primitives

Before we can consider the iterative methods and the way in which they may be rewritten as parallel algorithms, we must first describe the operations that the iterative methods use.

The iterative methods used for solving the system $Ax = b$ contain operations such as

- Matrix Vector multiplication
- Vector inner products
- Vector norms
- Vector sums
- Vector scalar multiplication

If each process holds a portion of the vector or matrix data, these operations can be performed in parallel, although some data will need to be exchanged between the processes.

5.5.1 Matrix Distribution

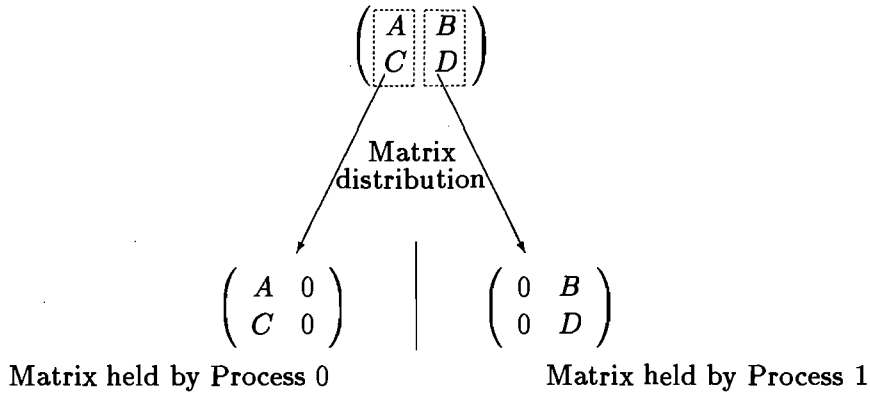

Figure 14: A Matrix distributed to 2 processors by Column-Striping

The matrix can be partitioned between the participating processors in several ways. Column-striped partitioning as in Figure 14 allocates several columns of the matrix to each processor. Row-striping is the analogous method where the matrix is split by rows. The columns of the matrix that are not assigned to a local process are filled with 0. These zero entries do not require any storage, because sparse storage techniques are used.The algorithms for the parallel operations are similar under row and column-striping.

It is also possible to divide the matrix up in a checkerboard pattern of blocks and this has advantages on parallel computers with many processing elements. However Paramaca was

written for a network of workstations with only a small number of processors available for the problem, so it would be hard to find sufficient processors for this division.

5.5.2 Parallel Matrix Vector Multiplication

When implementing a parallel matrix vector multiplication there are several factors to consider.

Pre- or post-multiplication

The matrix vector multiplications can be in the form Ax or xA. This will depend on the requirements of the iterative method. This distinction not only changes the way in which the processes perform their multiplications, but also the way they must communicate.

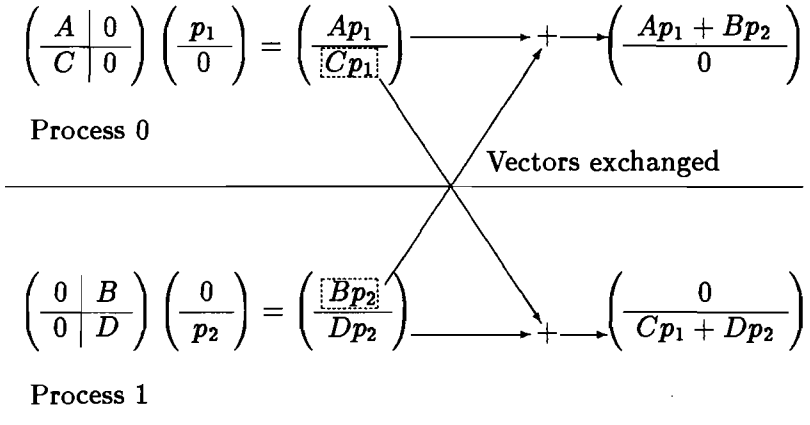

Figure 15: Matrix vector multiplication for row-striped multiplication

The choice between the master/slave and equal methods

There are two different ways to perform parallel matrix vector multiplication.

The first (the master/slave) method is for one processor, called the master, to hold the vector to be multiplied and to use the other processes, the workers, as slaves to help it with the task. The master sends a copy of the entire vector to each worker. The workers multiply the vector with the section of the matrix they hold and return the resulting partial vector to

the master. The master then combines these partial vectors to form the final result which is held only by the master process.

The second (the equal method) starts with the vector already split into sections and divided between the workers in proportions corresponding to the amount of the matrix they hold. Each process sends its portion of the vector to be multiplied to the other processes. When each process has received the full vector, it carries out the multiplication with the section of the matrix it holds. At the end of the operation each process again has only its portion of the result vector.

The vector partitioning of the equal method simplifies and reduces the communications cost of other parallel operations in the iterative methods such as inner products. For this reason equal method will be used in all the parallel iterative methods. Figure 15 shows the equal method of matrix vector multiplication with two process where A, B, C and D are block of the matrix and the vector to be multiplied is divided into two segments, p_1 and p_2.

The message communication between the processes depends both on whether a pre- or post-multiplication is being performed and on whether the matrix is distributed by row- or column-striping. The communication for the row-striped pre-multiplication is the same as the communication for column-striped post-multiplication. In this case the processes send their partial result vectors after the multiplication and a vector addition must be performed. Similarly row-striped post-multiplication and column-striped pre-multiplication have the same communication. The processes send their vectors before the multiplication and no vector additions or sends are necessary afterwards.

Traffic Analysis of Matrix Vector Multiplication

Master Sends Processes	Messages	Data	Worker Sends Processes	Messages	Data
1	0	0	1	0	0
2	1	n	2	1	$\frac{n}{2}$
3	2	$2n$	3	1	$\frac{n}{3}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
p	$(p-1)$	$(p-1)n$	p	1	$\frac{n}{p}$

Figure 16: The master/slave method with rowstriping, individual traffic.

The tables in this section show how much data is sent from each process and sent in total for each of the parallel matrix vector multiplication methods. Figures 16 and 17 show

Total Sends Processes	Messages	Data
1	0	0
2	2	$n + \frac{n}{2}$
3	4	$2n + \frac{2n}{3}$
$\vdots$	$\vdots$	$\vdots$
p	$2(p-1)$	$n(p-1)(1+\frac{1}{p})$

Figure 17: The Master/Slave method with rowstriping, Total traffic.

Individual Sends Processes	Messages	Data	Total Sends Processes	Messages	Data
1	0	0	1	1	0
2	1	$\frac{n}{2}$	2	2	$2n$
3	2	$\frac{2n}{3}$	3	6	$3n$
p	$(p-1)$	$\frac{(p-1)n}{p}$	p	$p(p-1)$	$(p-1)n$

Figure 18: The Equal method with rowstriping.

the master/slave case, and Figure 18 the equal case. In all tables, p is the number of processes involved and n is the length of the state vector. The data sent per matrix vector multiplication is $O(n)$ and $O(p)$. In other words the total amount of data sent will rise linearly with the number of processors used and the size of the vector.

Isoefficiency analysis attempts to determine how quickly the problem size must be increased with respect to the number of processors to maintain efficiency. [KGGK94]

If

- W is the problem size.
- p is the number of processes.
- T_p is the parallel run time.
- $T_o(W,p)$ is overhead of the parallel system as a function of W and p.
- S is the speedup.
- E is the efficiency.

then

$$T_p = \frac{W + T_o(W,p)}{p} \tag{27}$$

$$S = \frac{W}{T_p} \tag{28}$$

$$S = \frac{Wp}{W + T_o(W,p)} \tag{29}$$

And efficiency is

$$E = \frac{S}{p} \tag{30}$$

$$E = \frac{1}{1 + T_o(W,p)/W} \tag{31}$$

If W must be increased linearly as p is increased then the system is easily scalable. However if W must be increased at a greater rate then the system will not scale as well.

To find the isoefficiency function we can express W as a function of p by rewriting Equation 31 and allowing K to be a constant that depends on the level of efficiency to be maintained.

$$W = \frac{E}{1-E} T_o(W,p) \tag{32}$$

$$W = K T_o(W,p) \tag{33}$$

If we take the amount of data that needs to be sent to be the overhead of the parallel system then the tables would suggest that the isoefficiency function of matrix vector multiplication is $O(p)$.

This analysis assumes the full interconnection of the processes as presented to the programmer by the message passing library, but the actual traffic may vary depending on both the topology of the network and the message passing library implementation. The discussion of isoefficiency of parallel systems assumes that they are designed in a standard parallel processing topology such as a mesh or a hypercube where adding more processors will add more connections between different processors and thereby increase the network capacity.

Paramaca was designed and tested on an Ethernet local area network of workstations. In this network the available bandwidth does not increase as processors are added, conversely

congestion is made more likely and effective bandwidth will eventually decrease as more workstations are used as processors. Unfortunately as long as the network is connected in this fashion adding more processors will result in inefficiency regardless of the problem size.

Vector products and norms

A vector inner product between two vectors distributed across several computers is done by each worker calculating the inner product of the sections of the vectors that they possess. The final result is obtained by a reduction operation. The workers send all their results to the master who performs the necessary operation and then can return the final result to any workers that need it. If all workers get the final result then it is termed an All Reduce operation. The operations performed by the master on the partial results would be addition for vector inner products, and determining the maximum value for the maximum magnitude norms.

5.6 Classic Iterative methods

To solve $A\boldsymbol{x} = \boldsymbol{b}$ we can split A: $A = M - N$. Then solve for $\boldsymbol{x}$ as follows

$$(M - N)\boldsymbol{x} = \boldsymbol{b} \tag{34}$$

$$M\boldsymbol{x} = N\boldsymbol{x} + \boldsymbol{b} \tag{35}$$

and get the iteration

$$\boldsymbol{x}^{(k+1)} = M^{-1}N\boldsymbol{x}^{(k)} + M^{-1}\boldsymbol{b} = H\boldsymbol{x}^{(k)} + \boldsymbol{c} \tag{36}$$

$H = M^{-1}N$ and is the *iteration matrix* and $\boldsymbol{c} = M^{-1}\boldsymbol{b}$.

The Power, Jacobi and Gauss-Seidel methods can all be expressed in this form and the nature of H will determine the convergence properties.

If $\boldsymbol{x}$ is the solution then the error is $\boldsymbol{e}^{(k)} = \boldsymbol{x}^{(k)} - \boldsymbol{x}$.

Consider

$$H^k\boldsymbol{e}^{(0)} = H^k\boldsymbol{x}^{(0)} - H^k\boldsymbol{x} \tag{37}$$

$$H^k e^{(0)} = x^{(k)} - \sum_{i=0}^{k} H^i c - x + \sum_{i=0}^{k} H^i c \tag{38}$$

$$H^k e^{(0)} = x^{(k)} - x \tag{39}$$

$$H^k e^{(0)} = e^{(k)} \tag{40}$$

for convergence $\lim_{k\to\infty} e^{(k)} = 0$ and $\lim_{k\to\infty} H^k = 0$.

5.6.1 Scaling and Convergence

The matrix Q can be by multiplied by D^{-1} before applying the method so that the diagonals are equal to 1. Once the analyser has solved $Q^T D^{-1} x = 0$ to find x, π may be found by $\pi = (D^{-1}x)^T$.

There are several ways of testing the solution vector for convergence. Testing the largest value in the residual or the change in the solution vector are obvious approaches, although they can give incorrect results if the method converges slowly, or if all the values of the solution vector are small.

As a test for convergence, the method as recommended in [Ste91] is used. When the value

$$max_i \left(\frac{|\pi_i^{(k)} - \pi_i^{(k-1)}|}{\pi_i^{(k)}} \right)$$

is less than ϵ then we consider the method to have converged. ϵ is given a small value such as 10^{-8}.

5.7 Jacobi's Method

Jacobi's method [Var62] [GW89] is a simple iterative method based on the observation that solving $Ax = b$ is equivalent to finding the solution to the n equations

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \quad i = 1, 2, ..., n \tag{41}$$

Solving the ith equation for x_i yields

$$x_i = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x_j) \tag{42}$$

which suggests the iterative method

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}) \tag{43}$$

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

The residual $r_k = b - Ax_k$ therefore and we can express Equation 43 as

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}) \tag{44}$$

By substituting the residual term into Equation 44 we have the the following iterative form

$$x_i^{(k+1)} = \frac{r_i^k}{a_{ii}} + x_i^k \tag{45}$$

The serial algorithm for the Jacobi method [KGGK94] appears in Figure 19.

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

The Jacobi method may be derived by splitting the matrix A as in Section 5.6

$$A = Q^T = D - (L + U) = M - N$$

where $M = D$ and $N = L + U$

$$H_{Jacobi} = M^{-1}N = D^{-1}(L + U)$$

and as $b = 0$ we are solving homogeneous case

$$x^{(k+1)} = D^{-1}(D + U)x^{(k)} + 0$$

Algorithm for the Jacobi Method

```
begin
    k = 0;
    x^(0) =initial guess at solution;
    r^(0) = b − Ax^(0)
    while (not converged) do
    begin
        k = k + 1;
        for i = 0 to n − 1
            x^(k)[i] = r^(k−1)[i]/A[i, i] + x^(k−1)[i];
        r^(k) = b − Ax^(k);
        Test for Convergence
    endwhile
    x = x^(k)
end
```

Figure 19: Algorithm for the Jacobi Method

For example let

$$Q = \begin{pmatrix} -1 & 0.5 & 0.5 \\ 0.3 & -1 & 0.7 \\ 0.2 & 0.3 & -0.5 \end{pmatrix}$$

then

$$H_{Jacobi} = \begin{pmatrix} 0 & 0.3 & 0.2 \\ 0.5 & 0 & 0.3 \\ 1 & 1.4 & 0 \end{pmatrix}$$

The Jacobi method will not converge for all matrices and when the method does converge, it suffers from an extremely slow rate of convergence. For this reason Jacobi iteration is seldom used, as methods like Gauss-Seidel and SOR are more effective.

5.7.1 Gauss-Seidel

The Gauss-Seidel method [Ste91] is similar to the Jacobi method, but uses the most recent value for each solution element in the next calculation. The elements can be evaluated in ascending order (forward Gauss-Seidel) or descending (backward Gauss-Seidel). This uses the splitting $M = D - L$ and $N = U$. Then $H_{GS} = (D - L)^{-1}U$ for the forward method,

or $H_{GS} = (D - U)^{-1}L$ for the backward method. For the Jacobi method, the forward and backward methods are identical.

The forward method is also expressed as

$$x_i^{(k+1)} = \frac{1}{D_{i,i}(\sum_{j=1}^{i-1} L_{i,j}x_j^{(k+1)} + \sum_{j=i+1}^{n} U_{i,j}x_j^{(k)})} \tag{46}$$

A useful, but not universally applicable, rule of thumb is that if the majority of the elements in the matrix are below the diagonal then the forward method is recommended and conversely when most elements are above the diagonal the backward method is better.

As with the Jacobi method not all systems will converge, but the convergence properties of the Gauss-Seidel method are much better than that of the Jacobi method.

As seen from Equation 46 the computation of each element x_j depends on the previous elements x_i $(\forall i < j)$. This would make the algorithm essentially serial and prevent any parallel implementation. However the situation is not quite so severe, as the matrix is often so sparse that many of those dependencies do not occur. A parallel implementation is possible if the matrix elements are sorted and reordered in such a way that avoids these dependencies as much as possible.

One way to do this [BBC+94] is to partition the states into wavefronts. The first wavefront contains all those elements that depend on no predecessor. The second wave will be all elements that depend on no predecessor except the elements of the first wave. Once all the states have been allocated to wavefronts in this way, then the elements of each wavefront may be calculated in parallel.

In this way parallelism can be found in the Gauss-Seidel algorithm, although additional time is needed to divide the elements into wavefronts and reorder the computations.

5.7.2 Successive Overrelaxation

Successive Overrelaxation (SOR) is an extension of Gauss-Seidel which improves on the convergence rate of Gauss-Seidel.

The update to each solution vector element x_j is a weighted average of the element calculated by the Gauss-Seidel method and the previous iteration of x_j.

$$x_i^{(k+1)} = (1 - \omega)x^{(k)} + \omega\frac{1}{D_{i,i}(\sum_{j=1}^{i-1} L_{i,j}x_j^{(k+1)} + \sum_{j=i+1}^{n} U_{i,j}x_j^{(k)})} \tag{47}$$

Where ω is the relaxation parameter. If $\omega > 1$ then we are using over-relaxation and $\omega < 1$ is under-relaxation. If $\omega = 1$ then SOR is the Gauss-Seidel method.

SOR is expressed in the splitting form of Equation 36 as

$$M = (D - \omega L) \tag{48}$$

$$N = [(1 - \omega)D + \omega U] \tag{49}$$

$$H_\omega = (D - \omega L)^{-1}[(1 - \omega)D + \omega U] \tag{50}$$

SOR only converges if $0 < \omega < 2$, and there is an optimum value for ω that depends on the matrix Q. Unfortunately there is no known method of discovering the optimal value for general matrices.

The parallelisation issues for SOR are the same as those for Gauss-Seidel.

5.8 Conjugate Gradient Techniques

5.8.1 Conjugate Gradients

Matrix A is symmetric if

$$A^T = A \tag{51}$$

and positive definite if

$$x^T Ax > 0 \quad \forall x \neq 0 \tag{52}$$

The conjugate gradient method [She94][Ste91][SW95] can be used to solve the equation $Ax = b$ when A is symmetric and positive definite.

Consider the equation

$$f(x) = \frac{1}{2}x^T Ax - x^T b \tag{53}$$

If A is symmetric and positive definite then the gradient is

$$\frac{\partial f}{\partial x_k} = Ax_k - b = -r_k \tag{54}$$

When this gradient equals 0 then the function $f(x)$ is at a minimum. This implies that the problem of solving $Ax = b$ can be recast as that of minimising $f(x)$. This process of minimising f occurs by starting with an initial vector $x^{(0)}$ and computing successive values $x^{(k)} = x^{(k-1)} + \alpha^{(k)}\nu^{(k)}$ which are improvements on the previous iterates.

The selection of α and ν values is determined by the method chosen. A simple and intuitive method is that of *steepest descent* which, as the name implies, moves the solution vector in the direction of the gradient until the lowest point in that direction is reached.

For steepest descent $\alpha = \frac{r^T r}{r^T A r}$ and $\nu = b - Ax^{(k)}$.

In practice the method of steepest descent is slow to converge and it is quicker to use the conjugate gradient method which uses slightly different directions for modifying the value of x.

In the conjugate gradient method the ν vectors are A orthogonal[Wei94], which means

$$p_k^T A p_j = 0 \quad (\forall j \neq k). \tag{55}$$

The conjugate gradient method uses the following recurrences to change the solution vector x.

$$x_{k+1} = x_k + \alpha_k p_k \tag{56}$$

By multiplying Equation 56 by $-A$ and adding b we obtain

$$r_{k+1} = r_k - \alpha_k A p_k \tag{57}$$

$$p_{k+1} = r_{k+1} + \beta_k p_k \tag{58}$$

where

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

and

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

These updates result in the A-conjugacy of the vectors p and the conjugacy of the vectors r.

$$p_k^T A p_j = 0 \quad \text{for } j < k \tag{59}$$

$$r_k^T r_j = 0 \quad \text{and} \quad r_k^T p_j = 0 \quad \text{for } j < k \tag{60}$$

The conjugate gradient method (see Figure 20) is unsuitable for our purposes as we have no guarantee that the matrix Q will be positive definite or symmetrical. However there are ways to adapt the method to handle more general matrices.

Algorithm for the Conjugate Gradient Method

begin
$r = b - Ax$
$p = r$
$iterations = 0$
while (not converged)
$\rho = \tilde{r}.r$
$v = Ap$
$\alpha = \rho / p.v$
$r = r - \alpha.v$
$\beta = \tilde{r}.r / \rho$
$x = x + \alpha p$
$p = \beta p + r$
Test for convergence
$iterations + +$
end

Figure 20: Algorithm for the Conjugate Gradient Method

5.8.2 Bi-Conjugate Gradient (BiCG)

The biconjugate gradient algorithm (BiCG) [Fle76] adapts the method of conjugate gradients to nonsymmetric systems.

The BiCG method constructs a system with two residual vectors r_j and $\tilde{r}_j$ where $\tilde{r}_j{}^T r_i = 0$ if $i \neq j$. The search vector v_j similarly has a matching vector $\tilde{v}_j$ where $\tilde{v}_j{}^T A v_i = 0$ if $i \neq j$. The algorithm is given in Figure 21.

BiCG needs to perform two matrix vector multiplications in each iteration, one with A and the other with A^T. Some storage methods cannot access the transpose of the matrix and therefore cannot be used with the BiCG methods.

BiCG often has erratic convergence behaviour, and therefore faster methods with a smoother convergence, such as the conjugate gradient squared method or BiCGSTAB, are usually preferred.

5.8.3 Conjugate Gradient Squared (CGS) and Parallelisation

The conjugate gradient squared technique [Son89] is another method for using conjugate gradients for nonsymmetrical systems.

Algorithm for the BiConjugate Gradient Method

```
begin
    r̃ = b − A^T x
    p̃ = 0
    p = 0
    v = 0
    oldρ = 1
    iterations = 0
    while (not converged)
        ρ = r̃.r
        β = ρ/oldρ
        p = βp + r
        v = Ap
        p̃ = r̃ + βp̃
        α = ρ/p̃.v
        ṽ = A^T p̃
        x = x + αp
        r̃ = r̃ − αṽ
        oldρ = ρ
        Test for convergence
        iterations + +
    endwhile
end
```

Figure 21: Algorithm for the BiConjugate Gradient Method

Define

$$\alpha = \frac{\tilde{r}_0^T r_{j-1}}{\tilde{r}^T A \nu_j} \tag{61}$$

$$\nu = r_{j-1} + 2\beta_j q_{j-1} + \beta_j^2 \nu_{j-1} \tag{62}$$

$$\beta_j = \frac{\tilde{r}_0^T r_{j-1}}{\tilde{r}^T r_{j-2}} \tag{63}$$

$$q = r_{j-1} + \beta_j q_{j-1} - \alpha_j A \nu_j \tag{64}$$

$\tilde{r}_0^T$ is usually set to r_0.

then the recurrence equations for the conjugate gradient squared algorithm are

$$x_j = x_{j-1} + \alpha_j (r_{j-1} + \beta_j q_{j-1} + q_j)$$

$$r_j = r_{j-1} - \alpha_j A (r_{j-1} + \beta_j q_{j-1} + q_j)$$

Algorithm for the Conjugate Gradient Squared Method

begin
$old\rho = 0$
$r = b - Ax$
$\tilde{r} = r$
$p = 0$
$q = 0$
$old\rho = 1$
$iterations = 0$
while (not converged)
$\rho = \tilde{r}.r$
$\beta = \rho / old\rho$
$u = r + \beta q$
$p = u + \beta q + \beta\beta p$
$v = Ap$
$\alpha = \rho / \tilde{r}.v$
$q = u - \alpha * v$
$w = u + q$
$x = x + \alpha * w$
$r = b - Ax$: A parallel multiplication
Test for convergence
$iterations++$
endwhile
end

Figure 22: Serial Conjugate Gradient Squared Algorithm

The serial algorithm for conjugate gradient squared method is given in Figure 22.

The algorithm for the parallel version in Figure 23 is similar to the serial case, but with some operations replaced by their parallel versions and reduction operations inserted where necessary.

The parallel version of this algorithm uses the approach of having all the vectors distributed between the processes in the same proportions as the matrix, and uses the Equal multiplication method for matrix vector multiplications. While all the processors perform the same tasks, only the master has the extra responsibilities of deciding when to terminate the algorithm, coordinating the inner product sums and normalising and writing the final solution.

The algorithm requires communication between the workers at the following points.

- The matrix vector multiplication

Parallel Algorithm for the Conjugate Gradient Squared Method

```
begin
    r = b − Ax : A parallel multiplication
    r̃ = r
    p = 0
    q = 0
    oldρ = 0
    iterations = 0
    while (not converged)
        ρ = r̃.r
        allreduce(ρ,+)
        β = ρ/oldρ
        u = r + βq
        p = u + βq + ββp
        v = Ap : A parallel multiplication
        α = r̃.v
        allreduce(α,+)
        α = ρ/α
        q = u − α ∗ v
        w = u + q
        x = x + α ∗ w
        r = b − Ax : A parallel multiplication
        Test for convergence - Requires Allreduction
        ρ = oldρ
        iterations + +
    endwhile
end
```

Figure 23: Parallel Conjugate Gradient Squared Algorithm

- The vector norms
- Testing the solution vector for convergence.

The vector sums and scalar multiplications do not require communication if the vectors are already distributed between the processes.

The results of the Parallel CGS Method

The parallel CGS methods were tested on the BenchProd state transition model which was also used in Section 4.6.5. Each model size was tested with 1 to 5 processes in the

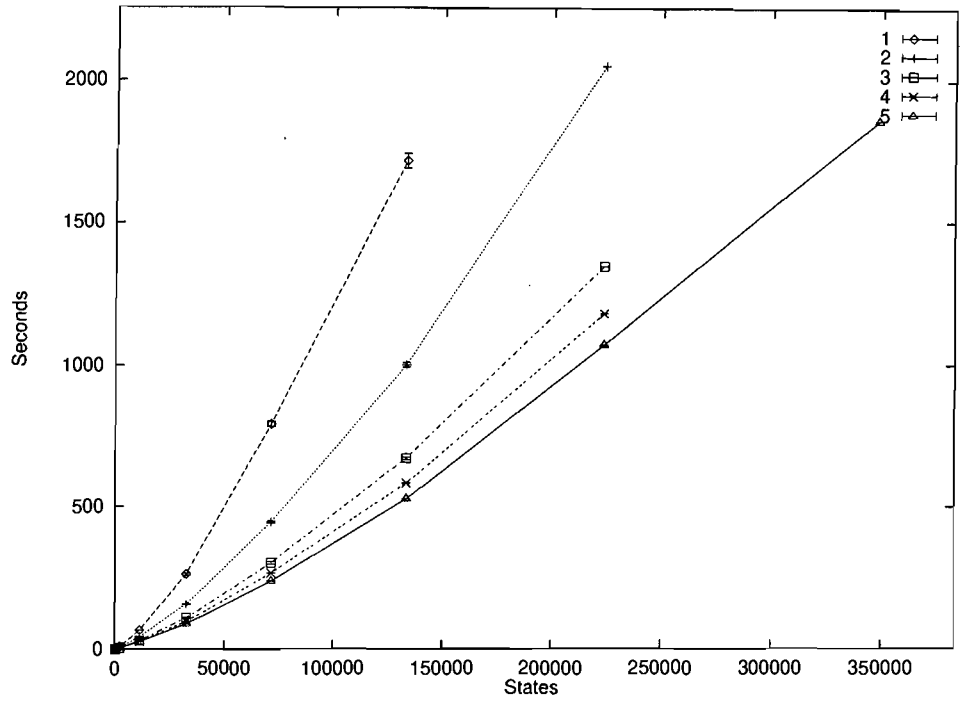

Figure 24: CPU Times for the CGS method for the BenchProd model.

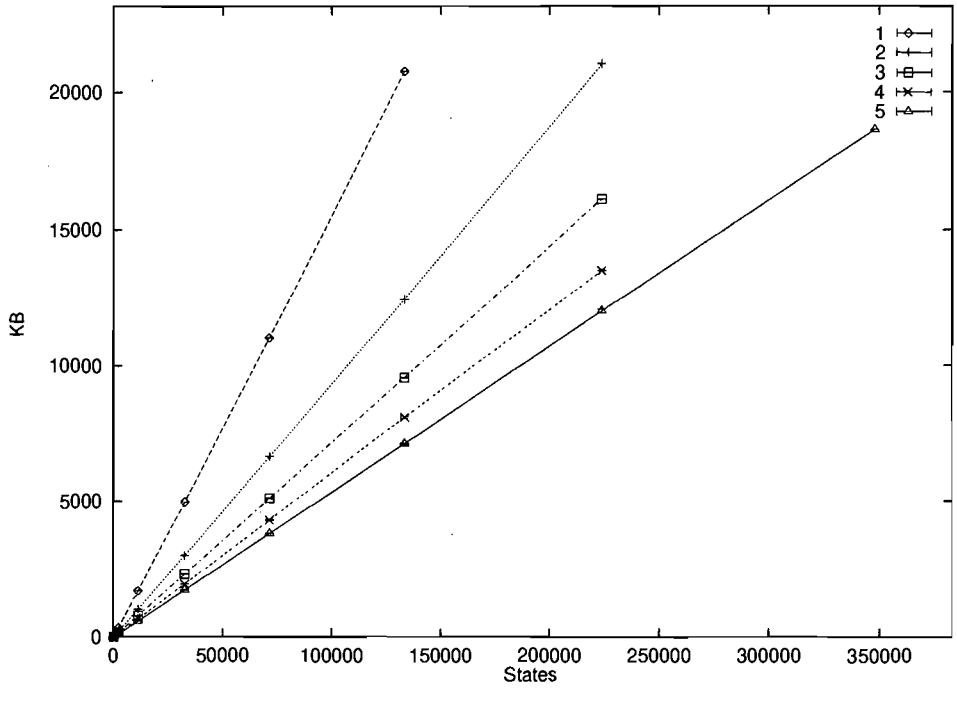

Figure 25: Memory use of the CGS method for the BenchProd model.

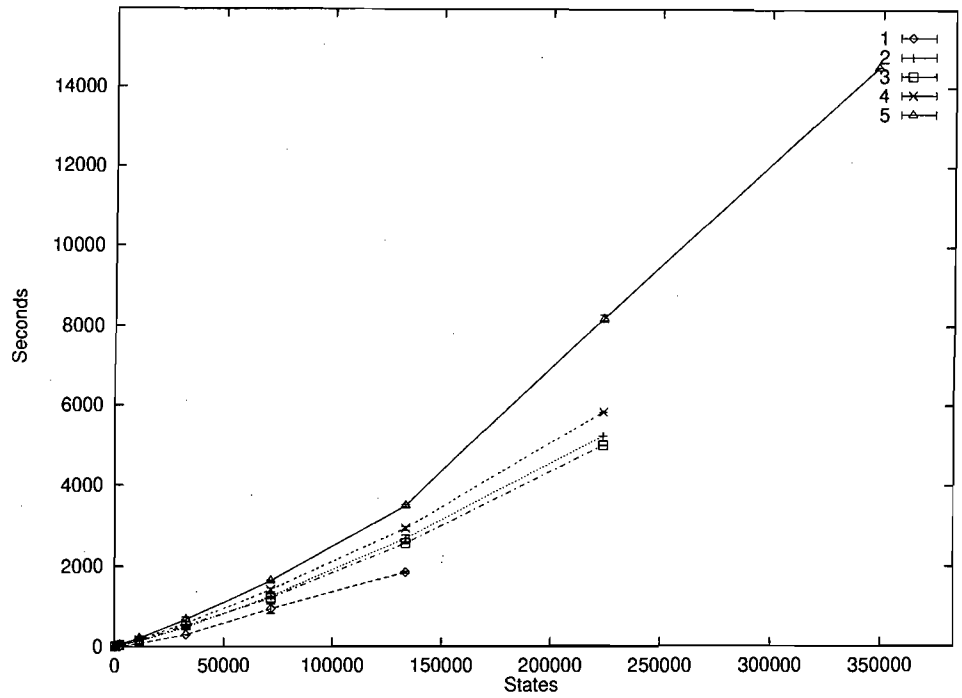

Figure 26: Elapsed Time for the CGS method for the BenchProd model.

process set. Both models have similar behaviour for the CGS and BiCGSTAB (Section 5.8.4) iterative methods, therefore the results and discussion for the CGS method will only be presented for BenchProd model and the results and discussion of the BiCGSTAB method will similarly only consider the FMS model - although results for both models are available.

Figure 24 shows that the parallel CGS algorithm achieves a reduction in mean CPU time per process and that the CPU times decrease steadily with increasing process set sizes. This trend naturally reflects the fact that each process now does less computation per iteration of the iterative method. Note that solving the 348 121 state Markov chain using 5 processors uses approximately the same CPU time per process as solving the 133 507 state chain using a single process.

Figure 25 shows that mean memory usage for the BenchProd model is reduced for models of all sizes. This reduction means that larger systems can be held in primary memory, and that page swapping will only occur at larger model sizes. These graphs also show that with more processes the solver was able to solve the larger models than it could with small numbers of processes. Models with large state spaces that were not divided between the process set caused machines to exceed their primary memory.

With 5 processes the average memory usage for the 348 121 state model is 18 641KB, which is less than the 20 745KB needed by 1 processor for a 133 507 state model.

Unfortunately the communications over the network are so slow that they overcome the advantages of the lower CPU time. The elapsed times are slightly higher for all model sizes (Figure 26).

The efficiency table below presents the efficiency of the parallel conjugate gradient squared algorithm based on the CPU times for solving the BenchProd model (133 507 states). As expected the efficiency decreases as the size of the process set increases.

Number of processes	CPU time (seconds)	CPU Time Efficiency
1	1 718	100%
2	1 001	86%
3	672	85%
4	582	74%
5	527	65%

5.8.4 Bi-conjugate Gradient Stabilised (BiCGSTAB) and Parallelisation

BiCGSTAB [Vor92][SV95] is a stabilised version of the biconjugate gradient algorithm which tries to maintain the convergence speed of CGS with a smoother convergence than CGS or BiCG.

BiCGSTAB performs two matrix vector multiplications, but both multiplications are with A so access to the transpose of the matrix is not needed. The serial version of the algorithm is given in Figure 27.

Parallel version

The same parallelisation techniques that were used for the CGS can be applied to the BiCGSTAB algorithm. Once the necessary operations have been replaced by their parallel equivalents, the new algorithm is given Figure 28.

Results of the parallel BiCGSTAB method

The BiCGStab Algorithm asks for the variable $\tilde{r}$ to be set to an arbitrary vector. We implement this by using a vector whose elements are uniformly randomly distributed between

Algorithm for the BiConjugate Gradient Stabilised Method

begin
$r = b - Ax$
$\tilde{r} = randomvector$
$p = 0$
$v = 0$
$\alpha = 1$
$\omega = 1$
$old\rho = 1$
$iterations = 0$
while (not converged)
$\rho = \tilde{r}.r$
$\beta = \rho\alpha/(old\rho\omega)$
$\gamma = \rho\alpha/(old\rho)$
$p = \beta p + \gamma v + r$
$v = Ap$
$\alpha = \rho/\tilde{r}.v$
$s = r - \alpha * v$
$t = As$
$\omega = t.s$
$\omega = \omega/t.t$
$x = x + \alpha p + \omega s$
$old\rho = \rho$
Test for convergence
$iterations++$
endwhile
end

Figure 27: Serial BiConjugate Gradient Stabilised Algorithm

0 and 1000. This randomness in the initialisation of the BiCGStab algorithm means that the number of iterations and time required to reach convergence may vary from run to run. For the purposes of obtaining timing results that do not show the wide deviations induced by the random vector, we make $\tilde{r}$ a uniform vector.

The parallel BiCGStab algorithm was tested on the FMS model (introduced in Section 4.6.5). The processors set size was varied from 1 to 5 for solving models of sizes of up to 537 768 states. Each test was run 5 times and the average CPU time, elapsed time and average memory use was recorded.

The FMS model and BiCGStab model has a very similar behaviour to the parallel CGS method solving the BenchProd model. The mean CPU times (Figure 29) are decreased by

using multiple processors. Figure 31 shows that the memory used per process is reduced by dividing the matrix between multiple processors.

As was the case with the parallel CGS method the elapsed time, in Figure 30, did not show an improvement. The time the processes spend waiting for for the messages exceeds that saved by the reduced computation load. The advantages of combining the memory of several machines come at the expense of longer run times.

Parallel Algorithm for the BiConjugate Gradient Stabilised Method

begin
$r = b - Ax$: A parallel multiplication
$\tilde{r} = randomvector$
$p = 0$
$v = 0$
$\alpha = 1$
$\omega = 1$
$old\rho = 1$
$iterations = 0$
while (not converged)
$\rho = \tilde{r}.r$
allreduce(ρ,+)
$\beta = \rho\alpha/(old\rho\omega)$
$\gamma = \rho\alpha/(old\rho)$
$p = \beta p + \gamma v + r$
$v = Ap$: A parallel multiplication
$\alpha = \tilde{r}.v$
allreduce(α,+)
$\alpha = \rho/\alpha$
$s = r - \alpha * v$
$t = As$
$\omega = t.s$
allreduce(ω,+)
$temp = t.t$
allreduce($temp$,+)
$\omega = \omega/temp$
$x = x + \alpha p + \omega s$
$old\rho = \rho$
Test for convergence - Requires Allreduction
$iterations++$
endwhile
end

Figure 28: Parallel BiConjugate Gradient Stabilised Algorithm

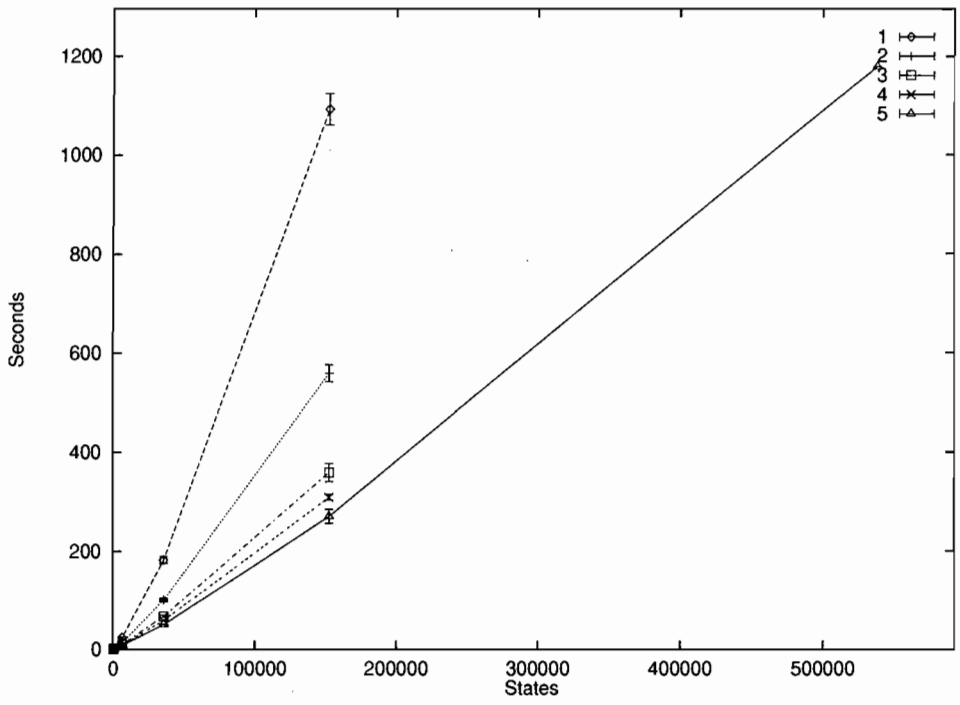

Figure 29: CPU Times for the BiCGSTAB method for the FMS model with 1-5 processes.

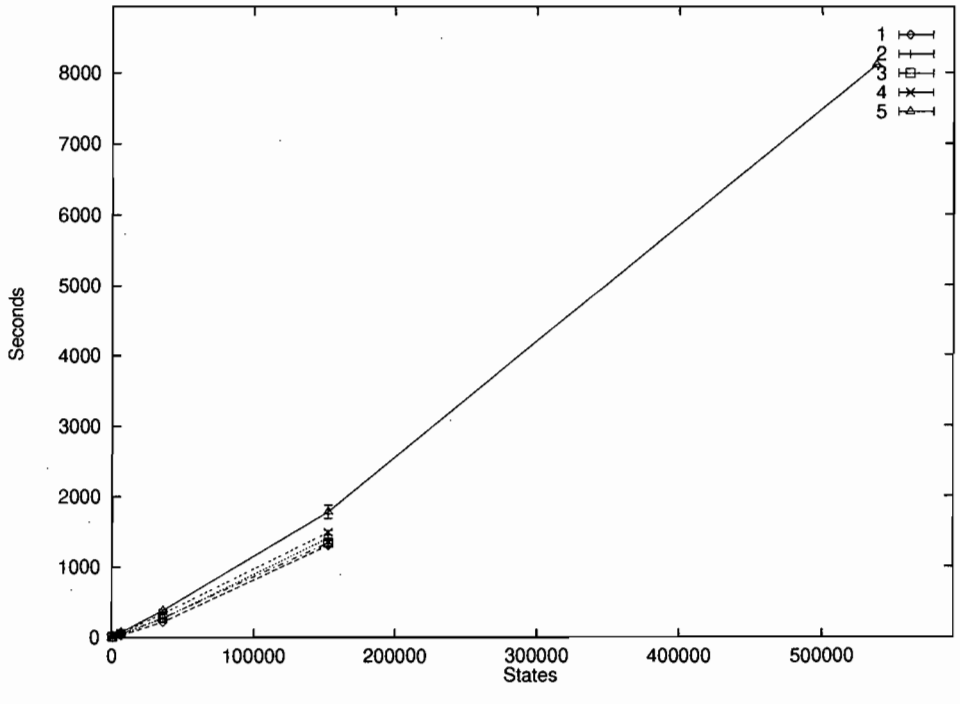

Figure 30: Elapsed times for the BiCGSTAB method for the FMS model with 1-5 processes.

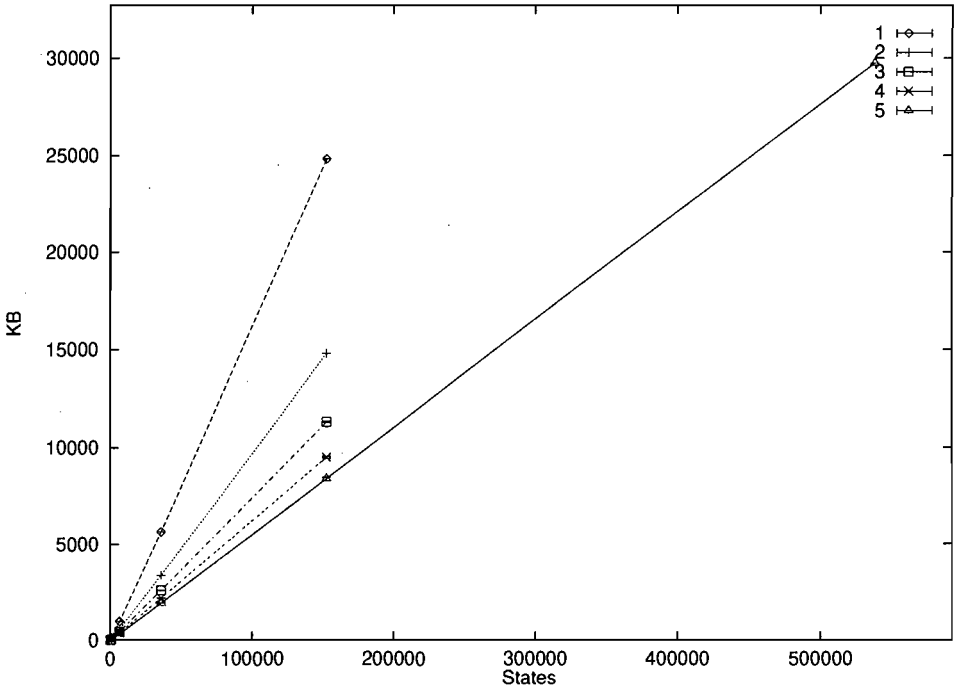

Figure 31: Memory use of BiCGSTAB for the FMS model with 1-5 processes.

Chapter 6

Conclusion

6.1 Introduction

This thesis considered the way in which parallelisation and distribution can be used to improve Markovian analysis of state transition systems. The Paramaca tool which implements these techniques is an extension of the tool DNAmaca and was tested on a local area network of Sun workstations.

The system to be analysed must be one that can be specified by an initial state with a set of rules giving conditions for transitions to occur and defining the effects of those transitions. These models are independent of any particular modelling formalism, but Markov chains can easily be derived from formal descriptions by using the interface language.

The Markovian analysis of systems suffers from the state space explosion problem, which means that even simple state transition system models often have extremely large state spaces. These Markov chains require so much memory and processing time that their generation and solution can become an intractable problem on a single workstation.

6.2 Distributed Generation and Solution

A distributed state space generator was implemented to take advantage of the parallelism in a local area network of computers. This method gives a large reduction in execution time for generating the state space and allows larger state spaces to be generated by combining the memory of several computers.

The functional analysis of the state transition graph involves finding the strongly connected components of a directed graph of states and transitions. This problem also requires large amounts of memory, therefore finding a linear time parallel algorithm would be useful. Unfortunately no linear time parallel algorithm for detecting final strongly connected components in a graph with transient states is known, although one does exist for confirming that the graph consists entirely of one strongly connected component.

The solution of the steady state equations of the Markov chain can also exceed the capacity of single computer because of the need to store a large sparse matrix. The steady state equations are solved by the use of iterative methods for large sets of linear equations.

The conjugate gradient methods for solving the Markov chain can be written as parallel algorithms by dividing the matrix among the processors and performing the operations of the iterative method in parallel. The conjugate gradient squared and bi-conjugate gradient stabilised algorithms were rewritten in this way. These parallel iterative methods resulted in lower CPU time per machine and lower maximum memory use in the participating computers. The overhead of network communication is high enough to overcome the reduction in CPU time, therefore the real time taken for finding the steady state solution is longer in the parallel algorithm. The lower maximum memory use allows larger systems to be solved even it if comes at the expense of longer execution times.

The factor most limiting to the effectiveness of the parallel Markov chain analyser is the speed of the network (Paramaca was tested on a 10Mbs Ethernet LAN). If these algorithms are used on a cluster of workstations with better interconnection, major improvements in performance, especially in the steady state solution phase can be expected.

6.3 Future Work

In both the state space exploration and steady state solving phases of analysis, the quantity of network traffic increases with the number of processors used. This results in a heavy network communication when many processors are used. This could be reduced by introducing caching techniques for state identifier communication.

The choice of partition function is another aspect that can benefit from study. Ciardo [CGN98] shows that good partition functions can reduce network communications during state space exploration. If we have no prior knowledge about the properties of the model it is difficult to see how to make an automatic selection of a function to reduce cross arcs. However a system designer will often have good insight as to what states the model

is likely to generate and this knowledge could guide the selection of a balanced function with locality. Distributed state space generators should therefore provide the user with the option of specifying his own partition function. The partition function also changes the order in which the states are found, which affects the convergence of some solution methods, but again these effects are difficult to predict before the state transition graph has been generated.

Bibliography

[AH97] S. C. Allmaier and G. Horton. Parallel shared-memory state-space exploration in stochastic modeling. *Lecture Notes in Computer Science*, 1253:207, 1997.

[AK93] Haruo Akimaru and Konosuke Kawashima. *Teletraffic: theory and applications.* Springer-Verlag, 1993.

[AMCB84] 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.

[Baa88] Sara Baase. *Computer Algorithms.* Addison-Wesley, 1988.

[BBC+94] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the solution of linear systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, 1994.

[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.

[BD95] Jonathan Billington and Michel Diaz, editors. *Petri nets applied to Protocols: Proceedings of a Workshop of the 16th International Conference on Application and Theory of Petri nets*, Torino, Italy, June 1995.

[BK94] F. Bause and P. Kemper. QPN-Tool for the qualitative and quantitative analysis of Queueing Petri Nets. In G. Haring and G. Kotsis, editors, *Lecture notes in Computer Science 794: Proceedings of the 7th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools, Vienna, Austria*, pages 321–334. Springer-Verlag, 1994.

[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.

[CG90] N. Carriero and D. Gelernter. *How to Write Parallel Programs.* MIT Press, 1990.

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

[CMT89] G. Ciardo, K. Muppala, and K.S. Trivedi. SPNP stochastic Petri net package. In *Proceedings of the Third International Workshop on Petri Nets and Performance Models*, Kyoto, Japan, 1989.

[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.

[DHv93] J. Demmel, M. Heath, and H. van der Vorst. Parallel numerical linear algebra. In *Acta Numerica 1993*, pages 111–198. Cambridge University Press, Cambridge, UK, 1993.

[Fle76] R. Fletcher. Conjugate gradient methods for indefinite systems. In *Lecture Notes in Mathematics*, volume 506, pages 73–89. Springer-Verlag, 1976.

[For94] Message Passing Interface Forum. MPI: A message passing interface standard, 1994. Online Document .

[G+94] Al Geist et al. *PVM 3 Users' Guide and Reference Manual.* Oak Ridge National Laboratory, September 1994.

[GKP96] G A Geist, J A Kohl, and P M Papadopoulous. PVM and MPI: A comparison of features. *Calculateurs Paralleles*, 8, 1996.

[GL] William Gropp and Ewing Lusk. *Installation Guide to MPICH, a Portable Implementation of MPI.* Argonne National Laboratory, University of Chicago.

[GW89] C. F. Gerald and P. O. Wheatley. *Applied Numerical Analysis.* Addison-Wesley, 4th edition, 1989.

[Hol95] Gerard 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.

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

[KMCS90] Udo R. Krieger, Bruno Müller-Clostermann, and Michael Sczittnick. Modelling and analysis of communication systems based on computational methods for Markov chains. *IEEE Journal on Selected Areas in Communications*, 8(9):1630–1648, December 1990.

[KMHK98] W. Knottenbelt, M. Mestern, P. Harrison, and P. Kritzinger. Probability, parallelism and the state space exploration problem. *Lecture notes in Computer Science 1469: Proceedings of 10th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 165–179, September 1998.

[Kno96] William Knottenbelt. Generalised Markovian analysis of timed transition systems. Master's thesis, University of Cape Town, 1996.

[KS95] R.L. Klevans and W.J. Stewart. From queueing networks to Markov chains: the XMARCA interface. *Performance Evaluation*, 24:23–45, 1995.

[Scz87] Michael Sczittnick. Technicken zur funktionalen und quantitativen Analyse von Markoffschen Rechensystemmodellen. Diplomarbeit, Universität Dortmund, October 1987.

[She94] Jonathan R Shewchuck. *An Introduction to the Conjugate Gradient Method without the Agonizing Pain.* School of Computer Science, Carnegie Mellon University, 1994.

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

[Ste74] William J. Stewart. *Markov Analysis of Operating System Techniques.* PhD thesis, Queen's University of Belfast, N. Ireland, 1974.

[Ste91] W.J. Stewart. MARCA: Markov chain analyser. a software package for Markov modelling. In W.J. Stewart, editor, *Numerical Solution of Markov Chains*, pages 37–62. Marcel Dekker Inc., New York, 1991.

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

[SV95] Gerard L. Sleijpen and Henk van der Vorst. An overview of approaches for the stable computation of hybrid bicg methods. Preprint 908, University of Utrecht, Department of Mathematics, March 1995. To appear in Appl. Numer. Math.

[SW95] W. Schönauer and R. Weiss. An engineering approach to generalized conjugate gradient methods and beyond. *Appl. Numer. Math.*, 19:175–206, 1995. Special Issue on Iterative Methods for Linear Equations.

[Var62] R. S. Varga. *Matrix Iterative Analysis.* Prentice-Hall, 1962.

[Vor92] Henk van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of BiCG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, March 1992.

[Wei94] Rüdiger Weiss. Orthogonalization methods. Interner Bericht 52, Universität Karlsruhe, 1994.

[WL93] Pierre Wolper and Denis Leroy. Reliable hashing without collision detection. In *Lecture Notes in Computer Science 697*, pages 59–70. Springer-Verlag, 1993.