

**Parallel iterative solution methods
for
Markov decision processes**

Thomas Welsh Archibald

**Thesis presented for the degree of
Doctor of Philosophy
University of Edinburgh
1992**



This work is dedicated to the memory of Cathie Archibald (1909–1992).

What though on hamely fare we dine,
Wear hoddin grey, an' a' that?
Gie fools their silks, and knaves their wine,
A man's a man for a' that.
For a' that, an' a' that,
Their tinsel show, an' a' that,
The honest man, tho' e'er sae poor,
Is king o' men for a' that.

Robert Burns

Contents

Declaration	vii
Acknowledgments	viii
Abstract	ix
Publications arising from this work	xi
List of abbreviations	xii
Notation used in the result tables	xiv
1 Introduction	1
1.1 Dynamic programming	1
1.2 Markov decision processes	2
1.3 Parallel processing	5
1.4 Parallel processing and dynamic programming	7
1.4.1 Introduction	7
1.4.2 Shortest path problems	8
1.4.3 Optimal splitting problems	14
1.4.4 Parallel processing and Markov decision processes	18
1.5 Overview of the remaining chapters	20
2 Value iteration methods	23
2.1 Basic Algorithms	23
2.2 Stopping rules for value iteration	30
2.3 Elimination of non-optimal actions	33
2.4 Reordering the state space	38
3 Properties of Markov decision processes	41
3.1 Properties which affect the performance of iterative solution methods	41
3.2 Test problems based on real models	43
3.2.1 Howard's car replacement problem (HOWARD)	43
3.2.2 Single item inventory problem (INVENT1)	44
3.2.3 Multiple item inventory problems	44

3.3	Generation of Markov decision processes	46
3.3.1	Motivation for problem generation	46
3.3.2	Existing generation techniques	46
3.3.3	PROCON — a new problem generator	47
3.4	Generated test problems	54
3.5	Generated problems with special transition structures	54
3.6	Summary of the properties of the test problems	55
4	Serial value iteration	58
4.1	Introduction	58
4.2	Results	59
4.3	Analysis	62
4.4	Conclusions	72
5	Parallel value iteration	74
5.1	Introduction	74
5.2	Synchronous non-overlapped parallel value iteration	75
5.3	Efficient communication procedures	79
5.3.1	Modelling communication between transputers	80
5.3.2	Procedures for multinode broadcasts	84
5.3.3	Procedures for global maximisations and minimisations	108
5.3.4	Conclusions	115
5.4	Implementing synchronous non-overlapped parallel value iteration	116
5.4.1	Introduction	116
5.4.2	Partitioning the state space	116
5.4.3	Results	118
5.4.4	Analysis	122
5.4.5	Conclusions	129
5.5	Asynchronous parallel value iteration	129
5.6	The phased pipeline algorithm	135
5.7	Modelling overlapped calculation and communication on a transputer	140
5.8	Implementing the phased pipeline algorithm	144
6	Reward Revision	150
6.1	Introduction	150
6.2	Serial reward revision algorithms	151
6.3	Computational aspects of serial reward revision	154

6.4 Implementing serial reward revision	156
6.5 Parallel reward revision algorithms	163
6.6 Implementing synchronised non-overlapping parallel reward revision	169
7 Summary of the main conclusions	175
Bibliography	177

Declaration

Except where specific reference is made to other sources, the work presented in this thesis is the original work of the author. The work has not been submitted, in whole or in part, for any other degree. Certain parts of the thesis have been published.

Thomas W. Archibald

Acknowledgments

Firstly I would like to express my gratitude to my supervisors, Dr. Ken McKinnon and Prof. Lyn Thomas, whose advice and enthusiasm made this work possible.

I also owe a great debt to my parents for their support and encouragement over the years and for their tolerance and patience over recent months.

I would like to thank Kenny Shaw and Iain Murray who have been a great help to me throughout my academic career.

Thanks are also due to the staff and students of the departments of Mathematics and Business Studies for providing a stimulating and friendly working environment.

Finally I would like to thank the University of Edinburgh and the Edinburgh Parallel Computing Centre for the use of their resources.

Abstract

Markov decision processes form an important class of dynamic programming problems because they are widely applicable. However solving real applications of Markov decision processes on serial computers is often impractical due to constraints on memory and processing time. Parallel processing has long been considered a potential solution to the computational intractability of these problems on serial machines, but prior to this work no detailed theoretical or practical studies in this area had been carried out.

This thesis examines several successful serial iterative solution methods for infinite horizon, time invariant, discounted Markov decision processes and develops efficient analogous parallel algorithms. Particular consideration is given to the two classes of iterative solution methods known as value iteration methods and reward revision, but the techniques developed and the conclusions drawn are applicable to other iterative methods for Markov decision processes (for example policy iteration methods) and also to iterative methods in general. Iterative methods are applied to many other problem areas including dynamic programming and the solution of linear and differential equations.

The main thrust of this thesis is concerned with the optimisation of the performance of the parallel algorithms developed. A detailed analysis of the implementation of several parallel iterative solution methods on a distributed memory, multiple instruction, multiple data, parallel processor reveals the key issues involved in optimising performance. Timing models are developed for processor communication time, processor calculation time and overall run time. These models guide the choice of the connection topology, the communication protocols and the degree of overlapping

of communication and calculation. This leads to the development of a phased pipeline algorithm which yields 60 fold speed-ups when a ring of 121 transputers is used to solve problems with 60,000 states and sparse transition structures.

The results include a comparison of the computational performance of both the serial and the parallel algorithms considered. The set of test problems used consists of problems with distinct identifiable properties. This enables the performance of the algorithms to be related to the properties of the problem being solved, so that criteria for selecting an efficient algorithm for a particular problem can be determined. An important factor in the success of this aspect of the study has been the development of a new method of generating Markov decision processes which allows the key properties of the problems to be controlled.

Publications arising from this work

There have been a total of five publications arising from this work, see items 3 to 7 in the bibliography. Most of chapter 1 appears in [3]. [5] is based on the results of Chapter 3. The analysis in [4] is similar to that in chapter 4 and section 5.4, although chapter 4 and section 5.4 consider some additional algorithms. The results of chapter 5 are reported in [6] and [7]. Reprints of these publications are not available.

In addition the main results of chapter 5 have been presented at the following conferences.

Parallel Numerical Analysis Workshop, Edinburgh, June 1991.

14th International Symposium on Mathematical Programming, Amsterdam, July 1991.

IMA Conference on Parallel Computation, Oxford, September 1991.

Edinburgh Parallel Computing Centre Annual Seminar, Edinburgh, September 1991.

ORSA/TIMS 32nd Joint National Meeting, Anaheim, November 1991.

EURO XII/TIMS XXXI Joint International Meeting, Helsinki, July 1992.

List of abbreviations

Value iteration algorithms

PJ — the pre-Jacobi algorithm, see page 23

J — the Jacobi algorithm, see page 24

GS — the Gauss-Seidel algorithm, see page 24

SOR — the successive over-relaxation algorithm, see page 25

Stopping rules

PB — a stopping rule based on the Porteus bounds, see page 30

$L_\infty B$ — a stopping rule based on the L_∞ norm bounds, see page 30

Action elimination tests

MAE — an action elimination test based on the MacQueen bounds, see page 33

PAE1 — an action elimination test based on the Porteus bounds, see page 34

PAE2 — an action elimination test based on the Porteus bounds, see page 35

$L_\infty AE1$ — an action elimination test based on the L_∞ norm bounds, see page 36

$L_\infty AE2$ — an action elimination test based on the L_∞ norm bounds, see page 36

As a prefix to GS or SOR

L — with a WLT ordering of the state space, see page 39

U — with the reverse of a WLT ordering of the state space, see page 39

R — with a random ordering of the state space, see page 39

A — with a particular ordering and its reverse alternately, see page 40

Names for particular value iteration algorithms

GS1 — a particular instance of generalised Gauss-Seidel, see page 76

GS2 — a particular instance of generalised Gauss-Seidel, see page 76

GS3 — a particular instance of variable generalised Gauss-Seidel, see page 137

SOR1 — a particular instance of generalised successive over-relaxation, see page 78

Communication protocols

BROADC — a multinode broadcast procedure for a chain topology, see page 86

BROADR — a multinode broadcast procedure for a ring topology, see page 93

BROADT — a multinode broadcast procedure for a torus topology, see page 98

GMAX — a global maximisation procedure for a chain or ring topology, see page 110

Other

SIMD — single instruction streams, multiple data streams, see page 6

MIMD — multiple instruction streams, multiple data streams, see page 6

CREW — concurrent reads, exclusive writes, see page 15

CRCW — concurrent reads, concurrent writes, see page 15

P-RAM — parallel random access memory, see page 15

WLT — weighted lower triangular, see page 38

cpu — central processing unit, see page 140

Notation used in the result tables

N_{iter} — the number of iterations required

P — the number of processors used

T_{soln} — the solution time

t_{aver} — the average time per iteration = $\frac{T_{soln}}{N_{iter}}$

T_{stan} — the standardised time = $\frac{T_{soln}}{\text{best serial solution time for that problem}}$

T_{comm} — the communication time ($T_{comm} = 0$ for serial cases)

ξ_{comm} — the communication overheads = $\frac{T_{comm}}{T_{soln}}$

speed-up — the speed-up = $100 \times \frac{\text{serial solution time}}{T_{soln}}$

ξ — the efficiency = $100 \times \frac{\text{speed-up}}{P}$

ω^* — the best choice for the relaxation parameter

F_{iter} — the increase in the number of iterations = $\frac{N_{iter}}{\text{number of iterations on 1 processor}}$

N_{upd} — the number of complete updates performed in asynchronous algorithms

F_{upd} — the increase in the number of updates = $\frac{N_{upd}}{\text{number of iterations on 1 processor}}$

D — the number of minor iterations performed during each major iteration of reward revision or asynchronous algorithms

λ — the threshold used to modify the transition structure in reward revision algorithms

F_{imp} — the ratio of the work per iteration for the revised problems to that for the original problem in reward revision algorithms = $\frac{A+2t_p-1}{A+2(t_p+t_z)-1}$, see section 6.3

$\bar{\beta}$ — the discount factor for the revised problems in reward revision algorithms

Chapter 1

Introduction

1.1 Dynamic programming

Dynamic programming was first proposed by Bellman in [8] as a technique for solving sequential decision processes. In these problems a number of decisions have to be made in sequence. The decision taken is based only on the state of the process at the time. Each decision involves a reward and affects the future state. To maximise total reward, a balance must be struck between receiving a high immediate reward and keeping out of states in which low rewards are unavoidable. All dynamic programming formulations are based on the principle of optimality which states that “an optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision” (Bellman [8] p83). In a deterministic formulation the decision taken dictates the next state or states. Two deterministic formulations are discussed in detail in sections 1.4.2 and 1.4.3. In other cases the decision taken may only influence the possible outcomes of a chance event. Such formulations are called stochastic and the objective is to maximise the expected reward. This study concentrates on stochastic formulations. The problem of minimising total or expected cost, as in a shortest path problem for example (see section 1.4.2), can be dealt with in a similar manner.

A very general dynamic programming formulation is given in (1.1), where m is the number of states, v_i is the maximum reward when the initial state is i , K_i is the set of all decisions permissible in state i , $f_i^k : \mathfrak{R}^m \rightarrow \mathfrak{R}$ is some function and v is the vector with components v_j for $j = 1, 2, \dots, m$.

INTRODUCTION

$$v_i = \max_{k \in K_i} f_i^k(v) \text{ for } i = 1, 2, \dots, m \quad (1.1)$$

The applications of sequential decision processes are extremely diverse and include allocation and scheduling problems, optimal search techniques and maintenance and replacement problems. Details of a large number of formulations can be found in Bellman and Dreyfus [9] and Howard [28]. While there are many practical problems which can be easily solved using dynamic programming (for instance `TeX`, the widely used text processing package, uses a finite stage deterministic dynamic programming formulation to decide where to place line breaks in order to achieve the most aesthetically pleasing output [29]), real life problems often have such large data sets that the computation of their solution is beyond the scope of serial machines due to memory and processing time constraints. This has restricted the extent to which dynamic programming has influenced decision making.

As a result the efficient implementation of dynamic programming algorithms is an area which has been at the centre of a considerable research effort for the last three decades. For example Bellman and Dreyfus [9] suggest many techniques which simplify the computational requirements of the dynamic programming algorithm for some applications and in the early 1980s a number of comparisons of the computational performance of solution algorithms for the important class of problems known as Markov decision processes were carried out, for example Porteus [42], Thomas *et al* [49] and Hendrikx *et al* [25].

1.2 Markov decision processes

Markov decision processes comprise an important class of dynamic programming problems. Applications arise in maintenance and replacement, inventory control, queuing models, harvesting policies, resource allocation and many other areas [55]. This study is concerned with infinite horizon time invariant discounted Markov decision processes which can be defined as follows.

The state space (or the set of all possible states) S is finite and consists of the values

INTRODUCTION

$1, 2, \dots, m$. Actions are chosen at regular time intervals or stages. For each state $i \in S$, an action is chosen from the action space K_i which is finite.

When action k is chosen in state i , an immediate reward r_i^k is received and the probability that the system will be in state j at the next stage is given by the transition probability $p_{i,j}^k$. Transition probabilities satisfy

$$\sum_{j \in S} p_{i,j}^k = 1 \quad \forall i \in S, \forall k \in K_i$$

$$\text{and } 0 \leq p_{i,j}^k \leq 1 \quad \forall i, j \in S, \forall k \in K_i.$$

The immediate rewards and the transition probabilities do not depend on the stage.

Future returns are discounted by a factor β , where $0 \leq \beta < 1$.

The single item inventory problem described below is a simple example which helps one to understand the various components of a Markov decision process.

Stock levels are reviewed on a regular basis (for example daily or weekly) and the frequency of review determines the stage. The state of the system is the current stock level and the state space is limited by the storage capacity.

The action is the number of units to be ordered. Any units ordered will arrive in stock after a delay referred to as the lead time. The action space in state i is finite and depends upon i since $i + k$ cannot exceed the storage capacity.

The immediate reward will typically consist of a fixed cost for placing an order, a cost per unit ordered, a cost per unit of stock held and a cost per unit of demand which cannot be met. Therefore in this example all rewards will be negative. The transition probabilities are determined by the demand for the item. For example if the lead time is one stage, then

$$p_{i,j}^k = \begin{cases} Pr(\text{demand in 1 stage} \geq i) & \text{if } j = k \\ Pr(\text{demand in 1 stage} = i - j + k) & \text{if } k < j \leq i + k \\ 0 & \text{otherwise} \end{cases}$$

A problem which is of much more practical interest is the multiple item inventory problem. In this case the state and action are vectors. Each component of the state vector is equal to the current stock level of one of the items and each component of the action vector is equal to the order quantity for one of the items. The transition probabilities have a more complex form as the demand for one item will not necessarily

INTRODUCTION

be independent of the demand for the other items. This problem (which will be discussed more fully in section 3.2.3) aptly illustrates the curse of dimensionality which plagues many real applications. Even a modest example with ten items and a storage capacity for six of each has a million states.

A policy is a rule which assigns a feasible action to each state at each stage. A policy which is independent of stage is called time invariant or stationary and can be denoted by $\delta : S \rightarrow \bigcup_{i \in S} K_i$ where $\delta(i) \in K_i$. Δ will be used to denote the set of all possible stationary policies.

The maximum expected reward over an infinite time horizon for a time invariant discounted Markov decision process with initial state i will be denoted by v_i^* . If the immediate rewards are bounded then v_i^* is bounded and is the unique solution of (1.2) (which is a form of Bellman's principle of optimality).

$$v_i = \max_{k \in K_i} \left\{ r_i^k + \beta \sum_{j \in S} p_{i,j}^k v_j \right\} \quad \forall i \in S \quad (1.2)$$

Further the stationary policy which assigns to state i the action which maximises the right hand side of (1.2) for state i is an optimal policy. For this reason when dealing with time invariant discounted Markov decision processes one need only consider stationary policies. For more details see [11]. From now on the term "policy" will be reserved for stationary policies.

The value of a policy δ will be denoted by v^δ and is the unique solution of (1.2) with K_i restricted to $\delta(i)$. For an optimal policy $v^* = v^\delta$. If $\|v^* - v^\delta\|_\infty < \epsilon$, where $\|u\|_\infty$ denotes the L_∞ norm¹ of u , δ is said to be ϵ -optimal. The transition matrix for a policy δ will be denoted by P^δ and is equal to $(p_{i,j}^{\delta(i)})_{1 \leq i,j \leq m}$.

Unlike in deterministic formulations where the methods used find the exact solution in a finite number of steps (see for example sections 1.4.2 and 1.4.3), methods for solving (1.2) only find a good approximation to the optimal solution. Given a small tolerance ϵ , the aim is to find an ϵ -optimal policy and vectors of bounds, $\underline{v}$ and $\bar{v}$, such that

$$\underline{v} \leq v^* \leq \bar{v} \text{ and } \|\bar{v} - \underline{v}\|_\infty < \epsilon. \quad (1.3)$$

Using an infinite time horizon and a time invariant model is useful when the problem

INTRODUCTION

has a large number of stages and the parameters governing the system are thought to vary slowly with time. Problems which are not time invariant can be reformulated as time invariant problems. This approach can be useful in the case of periodic problems, but the size of problem which can be solved in this way is severely restricted.

1.3 Parallel processing

There have always been many interesting problems which have been just beyond the scope of the computing technology of the time. Consequently larger and faster computers have always been sought after. One method of obtaining higher performance from existing technology is by parallel processing. The concept of parallel processing is simple: rather than increasing the speed of a single processor, one can reduce the solution time for a problem by the coordinated use of a number of processors linked by a communication network. The success of parallel processing depends critically on the nature of the problem under consideration, as idling, communication and repeated calculation are all wasteful. For example if the problem cannot be split into tasks requiring similar processing time (poor load balance), some processors will spend time idling. Also if the processors' data sets are highly dependent on each other, then a lot of communication or repeated calculation will be necessary.

Parallel processing is not a new idea (the earliest implementation dates back to the 1950s), but a number of factors have precipitated rapid development in this area over the last 15 years. Perhaps the most important factor has been the realisation that the desired improvements in performance cannot be achieved by increasing the speed of a single processor alone. The advances in technology which have made the production of very large arrays of simple processors a possibility have also been significant. Another factor has been the increase in availability of computers generally.

A number of companies specialising in the production of parallel computers with many processors have become established (for example Thinking Machines Corporation and Meiko), while the manufacturers of high speed vector computers, aware of the potential of parallel processing, now produce parallel computers with a modest number

INTRODUCTION

of high performance processors (for example the CRAY Y-MP). In this climate it is important to develop algorithms which are suitable for parallel computers and to understand the factors which affect the performance of these algorithms.

Parallel computers can be crudely classified according to whether they have multiple instruction streams, multiple data streams (MIMD) or single instruction streams, multiple data streams (SIMD) and shared or distributed memory. The efficiency of communication between processors depends upon the connection topology (which describes the manner in which the processors are connected). Some parallel computers have a fixed connection topology, while others allow the user some control over the choice of connection topology.

Hockney and Jesshope [27] provide more background information on parallel processing.

This research uses the Meiko computing surface at the University of Edinburgh. This machine consists of more than 400 T800-20MHz transputers with as many as 131 available for a single application and this makes it one of the largest transputer based computers in existence. The Meiko computing surface is a MIMD distributed memory computer. Each transputer has 4Mbytes of memory and four bidirectional links which are used to exchange data with other transputers. The links operate at 20Mbits/s. The connection topology is completely reconfigurable enabling experiments with different connection topologies to be conducted. Trew and Wilson [51] supply further details on Meiko and transputers. All code written in the course of this research uses OCCAM-2 [53].

Three measures are used in this study to quantify the success of a parallel algorithm, namely speed-up, efficiency and communication overheads. All three are defined on page xiv. The speed-up is a simple indication of the improvement obtained, while the efficiency relates the improvement obtained to the number of processors used. In this research the speed-up and efficiency for a parallel algorithm are relative to the solution time for the serial analogue of that algorithm. Using the solution time for the best serial algorithm in the calculation of speed-up and efficiency can hide some of the features

INTRODUCTION

of the performance of the parallel algorithm, for example it is difficult to compare how well two different algorithms parallelise. Bertsekas and Tsitsiklis [12] discuss further the calculation of speed-up and efficiency. Estimates of communication overheads enable information regarding the load imbalance to be deduced.

1.4 Parallel processing and dynamic programming

1.4.1 Introduction

The application of parallel processing has long been advocated as a potential solution to the computational intractability of the dynamic programming algorithm on serial machines and between 1960 and 1980 a number of articles were published on the subject. Bellman and Dreyfus [9] describe a parallel method for the solution of one-dimensional allocation processes. Tabak [46] identifies dynamic programming as suitable for parallel processing and assesses the likely impact of the evolving technology on this area. Casti *et al* [15] describe a number of algorithms which exploit the parallelism inherent in the dynamic programming formulation, but since these algorithms require a vast number of processors they were never implemented. Al-Dabass [1] extends these ideas, performs extensive theoretical comparisons and implements the algorithms developed on a two processor parallel computer.

Of course since then parallel computers have grown in size and become more widely available. As a result the number of articles on the subject of parallel dynamic programming has increased dramatically.

In section 1.4.2, the parallel solution of shortest path problems by dynamic programming is considered. It is interesting to note that one application of this class of problems is in the design of optimal routing algorithms for computer networks. Section 1.4.3 deals with parallel methods for optimal problem subdivision. Recently this problem has received more attention from parallel processing practitioners than any other dynamic programming formulation. Surprisingly the solution of Markov decision processes has received little attention and what little work there is in this area is reviewed in section 1.4.4.

INTRODUCTION

1.4.2 Shortest path problems

The problem of finding the shortest path from one vertex (vertex 1) to all other vertices in a digraph² can be formulated as a deterministic dynamic programming problem as follows

$$\begin{aligned} v_1 &= 0 \\ v_i &= \min_{k \in E_i} \{c_{k,i} + v_k\} \text{ for } i = 2, \dots, m \end{aligned} \tag{1.4}$$

where m is the number of vertices in the digraph, v_i is the length of the shortest path from 1 to i , E_i is the set of vertices which lie at the root of an edge into i and $c_{i,j}$ is the length of the edge from i to j . The cardinality of E_i is equal to the in-degree of i .

If every cycle has positive length and there exists a path from 1 to all other vertices, then (1.4) has a unique solution.

Bertsekas [11] gives more details of this formulation as well as a number of applications. An example with relevance to parallel processing is the problem of routing data in a computer network. The time required to send a message of length ℓ along a link joining node i to node j in a computer network can be modelled as $\sigma_{i,j} + \frac{\ell}{\rho_{i,j}}$, where $\sigma_{i,j}$ is the set-up time for the link and $\rho_{i,j}$ is the rate of transmission for the link. With $c_{i,j} = \sigma_{i,j} + \frac{\ell}{\rho_{i,j}}$ in (1.4), v_j can be interpreted as the time required to send a message of length ℓ from node 1 to node j using optimal routing.

One solution method for shortest path problems is Ford's algorithm [19]. In this algorithm each value is initialised so that v_j is an upper bound on the length of the shortest path from 1 to j and $v_1 = 0$. At an iteration each edge in the digraph is considered in turn. When an edge from i to j is encountered, v_j is updated according to $v_j = \min\{v_j, c_{i,j} + v_i\}$. The algorithm terminates when none of the values change during an iteration and at this point v_j is equal to the length of the shortest path from 1 to j .

If D is the maximum number of edges in a shortest path between 1 and any other vertex, convergence will be detected in at most $D+1$ iterations. However there are many instances of Ford's algorithm, each one corresponding to a different order of the edges, and some require far fewer iterations than others. Most other shortest path methods

INTRODUCTION

can be viewed as a version of Ford's algorithm with a specific order for updating the edges.

Bertsekas and Tsitsiklis [12] suggest using a parallel iterative solution method to solve (1.4). One such method, a distributed version of Ford's algorithm, involves partitioning the vertex set of the digraph into P sets denoted by $S_1, \dots, S_P$, where P is the number of processors used, and using the following recurrence relation.

$$\begin{aligned} v_j^0 &= \infty \text{ for } j = 2, \dots, m \\ v_1^r &= 0 \text{ for } r \geq 0 \\ v_j^r &= \min_{i \in E_j} \left\{ c_{i,j} + \begin{cases} v_i^r & \text{if } i, j \in S_t \text{ for any } t \text{ and } i < j \\ v_i^{r-1} & \text{otherwise} \end{cases} \right\} \text{ for } r \geq 1 \end{aligned}$$

At the r^{th} iteration processor t can calculate the values $\{v_j^r : j \in S_t\}$ independently of all other processors. Synchronisation is required at the end of each iteration, at which point a check for convergence is performed.

The method terminates when the values at two successive iterations are identical. Although bounded above by the maximum number of edges in any shortest path, the number of iterations required generally increases with the number of processors used. Therefore in an efficient parallel implementation a balance must be struck between reduced calculation time per iteration, due to the parallel execution, and increased number of iterations.

The method is analogous to a Gauss-Seidel method for solving linear equations and as the number of processors increases the convergence of such a method approaches that of a Jacobi method. This type of behaviour is observed when parallel value iteration algorithms are applied to Markov decision processes, see chapter 5. Bertsekas and Tsitsiklis [12] also consider a Jacobi method and discuss in more depth the importance of the choice of starting values.

If the digraph is acyclic, the vertices can be ordered so that an edge from i to j satisfies $i < j$ and the solution of the problem is greatly simplified. With such an order E_j is a subset of $\{1, 2, \dots, j-1\}$.

In some cases an order which achieves this is apparent from the application. For example consider a replacement problem in which the system is to be in operation for

INTRODUCTION

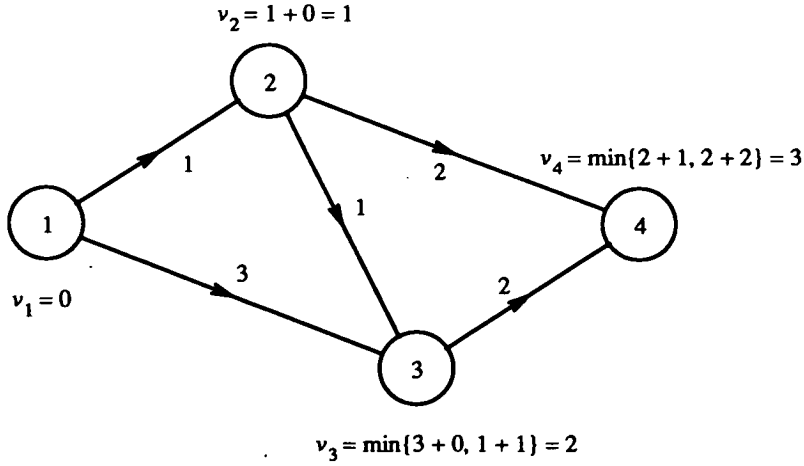


Figure 1.1: A simple shortest path problem in an acyclic digraph.

m time periods. Let $c_{i,j}$ equal the cost of replacing the incumbent machine at period i with one which will last until period j , then v_j can be interpreted as the minimum cost of maintaining the system until period j and the order of the vertices clearly has the above property. Shortest path problems in acyclic digraphs can also arise in scheduling problems and resource allocation problems.

If the structure of the problem does not yield a suitable order then it is possible to calculate one, the time required being linear in the number of edges in the digraph. Since this significantly reduces the solution time for the problem, it is worthwhile if a solution is to be calculated for several sets of edge lengths on the same digraph.

For this special case of the shortest path problem, the version of Ford's algorithm described above converges after only one iteration when one processor is used. In figure 1.1, the solution procedure is illustrated for a simple example with $m = 4$.

Nicol [38] describes three methods of distributing Ford's algorithm when the digraph is acyclic and compares their performance on a class of randomly generated problems using a shared memory MIMD multiprocessor. While two of the algorithms are based on commonly used techniques, the third is a novel approach which is referred to as the block window algorithm. It is suggested that this algorithm will be particularly effective for sparse digraphs and the results obtained appear to bear this out. The remainder of this section describes the three algorithms considered by Nicol in [38] and

INTRODUCTION

examines some of the points brought to light by the comparison.

For the rest of this section, it is assumed that the digraph is acyclic and the vertices are ordered so that an edge from i to j satisfies $i < j$.

The calculation of v_i in (1.4) only requires the values in $\{v_k : k = 1, \dots, i-1\}$ and so the values $\{v_i : i = 2, \dots, m\}$ can be calculated in sequence. The first parallel algorithm considered follows this approach using several processors in the computation of each value. This can be achieved by partitioning the sets E_i into P sets denoted by $E_{i,1}, \dots, E_{i,P}$, where P is the number of processors used, and calculating $\min_{k \in E_{i,t}} \{c_{k,i} + v_k\}$ on processor t . The calculation of v_i is then completed by the coordinated use of the processors in a global minimisation. This final step will typically require a lot of synchronisation, but in the first step processors can function independently.

The second approach uses the general purpose iterative method described earlier.

In the block window algorithm, the vertex set of the digraph is partitioned into blocks which (with the possible exception of the final block) consist of P vertices where P is the number of processors used. These blocks are denoted by

$$W_t = \left\{ (t-1)P + 1, \dots, \max\{tP, m\} \right\} \text{ for } t = 1, \dots, \left\lceil \frac{m}{P} \right\rceil.$$

Each block is considered in sequence and the values for vertices within a block are calculated in parallel using a two step procedure. In the first step, it is assumed that the values for vertices within the block being considered do not depend upon each other and so only edges with roots outwith the block need be considered. The second step checks the validity of this assumption and repeats any calculation that is necessary. Explicitly the algorithm can be written as follows.

Calculation procedure for block r

Step 1

On processor t , calculate

$$u_j = \min_{i \in E_j \setminus W_r} \{c_{i,j} + v_i\} \text{ where } j = (r-1)P + t.$$

If $E_j \setminus W_r = \emptyset$ then let $u_j = \infty$.

INTRODUCTION

Step 2

On processor t , if

$$z_j \equiv \min_{i \in E_j \cap W_r} \{c_{i,j} + u_i\} < u_j,$$

where again $j = (r - 1)P + t$, then set a flag. If $E_j \cap W_r = \emptyset$ then let $z_j = \infty$. If the flag is not set on any processor, $v_j = u_j \ \forall j \in W_r$. Otherwise determine $\hat{t}$, the lowest processor number for which the flag is set, and then, with $\hat{k} = (r - 1)P + \hat{t}$, $v_j = u_j$ for $j \in W_r$ satisfying $j < \hat{k}$ and the values $\{v_j : j \in W_r, j \geq \hat{k}\}$ have to be recalculated sequentially using

$$v_j = \min \left\{ u_j, z_j, \min_{i \in E_j, i \geq \hat{k}} \{c_{i,j} + v_i\} \right\}.$$

Note that if $(r - 1)P + t > m$ where $1 < t \leq P$, processor t takes no part in the calculation for block r . This situation can only arise when the final block is being processed. The calculation of the values for the first block will have to be performed sequentially. The algorithm requires synchronisation before the start of the second step and also before the value of $\hat{t}$ can be determined.

The validity of step 2 hinges on proposition 1 of [38] which states that

$$\begin{aligned} \text{if } u_j &\leq \min_{i \in E_j \cap W_r} \{c_{i,j} + u_i\} \text{ for all } j \in W_r \text{ satisfying } j < k, \\ \text{then } v_\ell &= u_\ell \text{ for all } \ell \in W_r \text{ satisfying } \ell < k. \end{aligned}$$

If $E_j \cap W_r = \emptyset$ then the condition is taken as being satisfied. The proof of this proposition is straightforward using induction.

Since the calculation of every edge sum involves the same computation, it is always possible to achieve a good load balance for the first algorithm by, as far as possible, assigning the same number of edges to each set $E_{j,t}$. The load imbalance can be no worse than one addition and one comparison.

The first algorithm is effective when the digraph is large and dense, because in this situation the in-degree of the vertices is large compared to the number of processors, and consequently the time saved by the parallel evaluation of the edge

INTRODUCTION

sums (i.e. $c_{k,i} + v_k$) is far greater than the overheads of the synchronisation required by the global minimisation. For highly sparse digraphs this will not be the case.

The other two algorithms partition the problem on the vertex set and so do not rely on the high density of the digraph to provide scope for parallelism. However for a fixed number of vertices, the overheads associated with the synchronisation required in these algorithms will become less significant as the density of the digraph increases.

If the vertex set is large compared to the number of processors, it will usually be possible to find a partition for the iterative method which results in good load balancing. For digraphs with shortest paths consisting of few edges, the number of iterations required will remain small as the number of processors used increases and hence the efficiency of the iterative method will be high. This is certainly not a factor which affects the efficiency of the other two algorithms.

As presented the block window algorithm only achieves good load balancing during step 1 if the cardinality of the sets $E_j \setminus W_r$ is similar for all $j \in W_r$. If this is not the case, the vertex set of the digraph can be partitioned into blocks consisting of more than P vertices. Each block can then be partitioned into P sets in a manner which will achieve good load balancing during step 1.

The efficiency of the block window algorithm also depends upon the number of vertices recalculated sequentially during step 2. If the proportion of edges which join vertices in the same block is low, this number is likely to be low and any load imbalance during the test in step 2 will be insignificant. This will be the case if, for example, the in-degree of the vertices is large compared to the size of the blocks.

Nicol [38] implements the three algorithms on the Flex/32 multicomputer [35] which is a shared memory MIMD parallel computer. The test problems have 1024 vertices and are generated in a way which ensures that the expected in-degree of the vertices is the same for almost all vertices. Edges joining i to j where $j - i > \omega$ are not considered by the generation procedure. In the test problems ω is chosen to be 64 and 128. The sparsity of the digraphs varies from 1% to 12.5%.

The theoretical analysis determines conditions under which the assumption in the

INTRODUCTION

first step of the block window algorithm (i.e. that values for vertices within the same block do not influence each other) has a high probability of being valid. Although not all the test problems meet these conditions, the results show that the block window algorithm consistently out performs the other two on 2, 4, 8 and 16 processors. On 16 processors the block window algorithm achieves an efficiency of just over 50% when the sparsity of the digraph is 12.5%, while the other two algorithms achieve efficiencies of under 20%.

1.4.3 Optimal splitting problems

One dynamic programming formulation which frequently arises in combinatorial optimisation problems has the following functional equation.

$$v_{i,j} = \min_{i \leq k < j} \{c_{i,k,j} + v_{i,k} + v_{k+1,j}\} \text{ for } 1 \leq i < j \leq m \quad (1.5)$$

For example, this approach can be used to determine the optimal parenthesisation of a matrix product $M_1 M_2 \dots M_m$. Let r_i be the number of rows in M_i and r_{m+1} be the number of columns in M_m . For the product to be well defined, M_i must have r_{i+1} columns. With $v_{i,i} = 0$ and $c_{i,k,j} = r_i(2r_{k+1} - 1)r_{j+1}$ (the number of operations performed when calculating the product of one matrix with order $r_i \times r_{k+1}$ and another with order $r_{k+1} \times r_{j+1}$), $v_{i,j}$ can be interpreted as the minimum number of operations required in the calculation of the matrix product $M_i M_{i+1} \dots M_j$.

To appreciate the value of an optimal parenthesisation consider the following trivial case involving three small matrices. Calculate $M_1 M_2 M_3$ where M_1 , M_2 and M_3 have orders (3×2) , (2×4) and (4×1) respectively. Computing $M_1(M_2 M_3)$ requires 23 operations, while computing $(M_1 M_2)M_3$ requires 47 operations. The fact that the range should be so great in such a small example, illustrates the importance of parenthesisation to the efficient calculation of large matrix products.

Another application of this dynamic programming formulation arises in the design of chemical plant to separate chemical mixtures into their components [26]. Since on serial machines the problem cannot be solved quickly enough to be of use to design engineers, parallel solution methods have been sought [20].

INTRODUCTION

Since the solution of (1.5) is computationally intensive (requiring $\frac{m^3}{2} - \frac{m^2}{2}$ operations even when $c_{i,k,j}$ requires no calculation) and the formulation is widely applicable, the development of efficient parallel algorithms for this problem has received a lot of attention recently.

Rytter [45] shows that on a shared memory multiprocessor which allows concurrent reads, but does not permit concurrent writes (the CREW P-RAM model), (1.5) can be solved in $\log^2 m$ time using $\frac{m^6}{\log m}$ processors. Further if concurrent writes are permitted (the CRCW P-RAM model) the solution time can be reduced to $\log m$, because with this model the minimum of a set of values can be determined in a time which is independent of the size of the set [30].

Louka and Tchente [31] propose an algorithm for a distributed memory parallel computer which solves (1.5) in $O(m)$ time using a systolic array consisting of $\frac{m^2}{8} + O(m)$ processors. A number of other methods which achieve $O(m)$ time were known at the time, but the best of these requires twice this number of processors [22].

The performance achieved by the algorithms presented in these two papers must be considered as a theoretical bound on the performance which can be obtained, as these algorithms are not practical for solving realistic examples of (1.5) on the parallel computers currently available because of the large number of processors they require.

Miguet and Robert [36] and Edmonds *et al* [18] consider the case in which the number of processors P is much smaller than m . The former uses a distributed memory multiprocessor, the latter a shared memory multiprocessor. Although designing algorithms for different architectures, the two papers use a similar strategy to distribute the calculation among the processors. This is loosely based on the following observations.

The values in the set $S_d = \{v_{i,j} : j - i = d\}$ can be calculated independently given a knowledge of the values in the set $\bigcup_{t=0}^{d-1} S_t$. More precisely, calculating $v_{i,j}$ where $j - i = d$ requires $\{v_{i,k} : k - i < d\}$ and $\{v_{k,j} : j - k < d\}$. Thus in a parallel algorithm to solve (1.5), the sets S_d can be evaluated in order of increasing d , with the calculation of the values in each set being distributed. The set S_d corresponds to a diagonal of the

INTRODUCTION

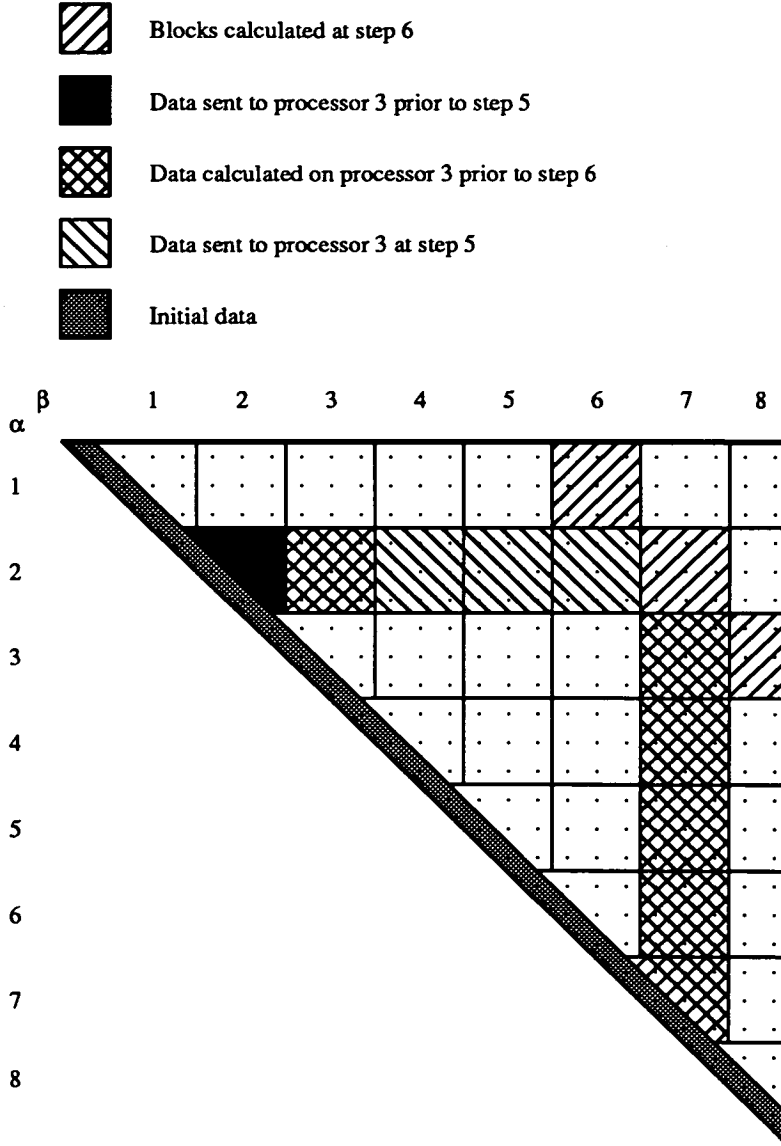


Figure 1.2: Diagram of the cost matrix highlighting the data used by processor 3 at step 6 of one algorithm for a simple example with $m = 24$, $P = 4$ and $\ell = 3$.

cost matrix $\{v_{i,j}\}_{i < j}$, which is upper triangular (see figure 1.2).

The number of values in S_d is $m-d$ and calculating any value in the set requires $3d-1$ operations. Therefore calculating the same number of values from S_d on each processor will achieve perfect load balancing, and when this is not possible the imbalance need be no more than $3d-1$ operations.

If this imbalance is significant (as it might be since S_{m-1} contains only one element which requires $3m-4$ operations to update), then, at the expense of some

INTRODUCTION

additional communication or synchronisation, the calculation of individual values could be distributed or the calculation of values in S_d could be started as soon as the evaluation of S_e , where $e = \left\lceil \frac{d-1}{2} \right\rceil$, is complete.

Neither approach attempts this kind of fine load balancing. Edmonds *et al* [18] switch to a serial algorithm near the end of the solution, when the scope for parallel execution is not sufficient to keep all processors occupied. This prevents the synchronisation cost becoming dominant. Miguet and Robert [36] assume that the load imbalance in the late stages of the algorithm is insignificant.

In both cases, the aim is to find an allocation of the values of the cost matrix to processors for calculation which achieves a good load balance throughout the solution, but does not introduce high communication or synchronisation overheads.

One method is to allocate the values in $V_t = \{v_{i,j} : 1 \leq i < j, j \equiv t \bmod P\}$ to processor t and evaluate the sets S_d in sequence. This approach allocates columns of the cost matrix to processors in a wrap around fashion and achieves as good a load balance as possible, since the number of values in V_t satisfying $j - i = d$ is less than or equal to $\left\lceil \frac{m-d}{P} \right\rceil$. This is easy to see because $S_d = \{v_{i,j} : d+1 \leq j \leq m, i = j-d\}$ and out of g consecutive numbers, at most $\left\lceil \frac{g}{h} \right\rceil$ are equal to f modulo h .

Both studies consider such an allocation, but find that the communication or synchronisation overheads are high and this limits the speed-ups which can be obtained. It is possible to evaluate a number of the sets S_d concurrently, and in so doing the cost of communication or synchronisation can be reduced at the expense of a poorer load balance.

This is the basis of the narrow-band algorithm proposed by Edmonds *et al* [18]. The number of sets evaluated in parallel falls as the algorithm proceeds and is chosen to keep the load imbalance to a minimum. It is shown that the arithmetic cost of the algorithm is $O\left(\frac{m^3}{6P}\right)$, so that the dominant term is the same as for the method above, but the synchronisation cost is much less, $O(|\log_c m|)$ where $c = \frac{2P-1}{2P+1}$ provided P is much smaller than m .

Applied to a distributed memory parallel computer, this method of allocating values

INTRODUCTION

to processors results in a complex distribution of the data. The approach adopted by Miguet and Robert [36] allocates blocks of ℓ consecutive columns of the cost matrix to processors, the r^{th} block being allocated to processor t if $r \equiv t \pmod{P}$. This results in a straightforward distribution of the data. Note that if ℓ does not divide $m - 1$, the final block will have less than ℓ columns (as in figure 1.2 for example).

The rows are partitioned in a similar way so that the cost matrix is partitioned into blocks $B_{\alpha,\beta}$ where $1 \leq \alpha \leq \beta \leq \left\lceil \frac{m-1}{\ell} \right\rceil$. At the r^{th} step of the algorithm, processor t calculates the values in blocks $B_{\beta-r+1,\beta}$ where $\beta \equiv t \pmod{P}$ and then sends these values to processor $t + 1$ along with the values in the previous $P - 2$ (or $r - 1$ if $r < P$) blocks from row $\beta - r + 1$. For blocks in the final column (i.e. $\beta = \left\lceil \frac{m-1}{\ell} \right\rceil$) no values need be passed on, although doing so will not increase communication overheads and may simplify the code. This procedure is illustrated in figure 1.2.

The communication protocol described is optimal in the sense that no processor receives any value more than once and all values received by a processor are required in its calculations. Since processor t communicates with processors $t - 1$ and $t + 1$ only, a ring of processors is the optimal topology for this protocol. In this case the communication cost is $O\left(\frac{m^2}{2}\right)$. The communication protocol used by Miguet and Robert [36] involves some redundancy and so is less efficient, it has a cost of $O\left(\frac{m^3}{P\ell}\right)$. The arithmetic cost of the algorithm is $O\left(\frac{m^3}{6P}\right)$. Miguet and Robert [36] construct an estimator for the choice of ℓ which maximises the efficiency for a given m and P .

Both papers present numerical results. Edmonds *et al* [18] use a Sequent S27 parallel computer with 10 processors and 32 Mbytes of global shared memory. Using 9 processors the narrow-band algorithm achieves an efficiency of 92% for one application with $m = 1000$. Miguet and Robert [36] use a ring of 16 T414 transputers [51], each transputer having its own local memory. Using all 16 processors and a block size of 6, an efficiency of 81% is achieved for a different application with $m = 1080$.

1.4.4 Parallel processing and Markov decision processes

In optimal splitting problems a decision results in transitions to two states, whereas in shortest path problems a decision results in only one transition. In a Markov decision

INTRODUCTION

process a decision determines the probability of a transition to each state, so there can be many possible transitions arising from a single decision. If the transition probabilities and the discount factor are suitably defined, shortest path problems and optimal splitting problems can be treated as special classes of Markov decision processes.

Bertsekas and Tsitsiklis [12] consider very briefly a parallel version of one algorithm for Markov decision processes (this algorithm is known as pre-Jacobi see chapter 2) assuming that the transition matrices are dense. This algorithm is similar to the parallel iterative method proposed by Bertsekas and Tsitsiklis [12] for shortest path problems (see section 1.4.2).

Bertsekas [10] proposes the following asynchronous parallel algorithm for dynamic programming and proves that it converges under certain assumptions. The algorithm applies to a wide class of formulations including shortest path problems and finite and infinite horizon Markov decision processes. A general iterative algorithm is considered which when applied to Markov decision processes encompasses the solution technique known as value iteration (see chapter 2).

The state space is partitioned into P sets denoted by $S_1, \dots, S_P$ and processor r is responsible for updating the states in S_r . Processor r is called a neighbour of processor t if estimates of the values of states in S_r influence estimates of the values of states in S_t . For Markov decision processes this happens if and only if under some action there exists a transition with non-zero probability from a state in S_t to a state in S_r . The estimates for states updated on a processor depend only on the estimates for states which are updated on one of its neighbours. Therefore a processor need only receive estimates from its neighbours and send its estimates to processors of which it is a neighbour.

At any time a processor can be in one of three phases. It can be updating the states it has been allocated or transmitting its latest estimates to one or more processors of which it is a neighbour or idling. It is assumed that a processor can update and receive concurrently, transmit and receive concurrently and transmit to more than one processor at a time. However processors are not able to update and transmit in parallel.

Under the following two assumptions, it is shown that an asynchronous algorithm

INTRODUCTION

organised in this way converges to the optimal solution of the problem.

Assumption 1

There exists $T > 0$ such that for all processors r and for all time intervals of length T , processor r performs at least one update of all states in S_r and at least one transmission to all processors of which it is a neighbour.

Assumption 2

There exists vectors $\underline{v}$ and $\bar{v}$ such that $\underline{v}$ converges monotonically increasing to v^* , $\bar{v}$ converges monotonically decreasing to v^* and $\underline{v} \leq v^{0,r} \leq \bar{v}$ for all processors r , where v^* is the optimal solution and $v^{0,r}$ is the initial approximation on processor r .

Further it is shown that for shortest path problems and finite horizon Markov decision problems, convergence occurs in finite time.

1.5 Overview of the remaining chapters

Chapter 2 gives full details of all the value iteration methods implemented in this study. This includes proofs of convergence, stopping rules and action elimination techniques. Two generalisations of the traditional Gauss-Seidel algorithm which are amenable to parallel processing are proposed.

Chapter 3 describes several properties of Markov decision processes which affect the performance of iterative solution methods and classifies the test problems according to these properties. This chapter also includes a detailed description of a new problem generation technique which allows the key properties of Markov decision processes to be controlled.

Chapter 4 compares the computational performance of a number of serial value iteration algorithms using test problems with distinct properties. This comparison is used to identify techniques which are of little use and so need not be incorporated in the parallel algorithms.

The main results of the thesis are to be found in chapter 5 which develops parallel value iteration algorithms using a simple non-overlapping approach, which is efficient

INTRODUCTION

for small problems, and a sophisticated phased pipeline approach, suitable for large sparse problems. Section 5.2 describes several simple parallel value iteration algorithms and identifies their communication requirements. Section 5.3 develops communication protocols to satisfy these requirements for three connection topologies. The model for communication between transputers developed in section 5.3 is used to determine the relative merits of the connection topologies considered. Some of the topics dealt with in section 5.3 have been addressed for other applications. Boreddy and Paulraj [13] use similar models for communication between processors, but do not perform detailed analysis. Brochard [14] looks at communication protocols for several connection topologies, but these are less efficient than the procedures described in this study. Section 5.4 compares the computational performance of the simple parallel value iteration algorithms and concludes that communication overheads are the major factor limiting the efficiency of these algorithms. This motivates the development of asynchronous value iteration algorithms in section 5.5 and phased pipeline algorithms in section 5.6. Section 5.7 develops a model for overlapped calculation and communication on a transputer and this model is used in section 5.8 to optimise the performance of the phased pipeline algorithms.

Chapter 6 uses the results of the computational comparisons of the serial and parallel value iteration algorithms to guide the development of a number of serial and parallel algorithms based on reward revision. The performance of the resulting algorithms is compared with that of the best value iteration algorithms using multiple item inventory problems.

Definitions of all the abbreviations used in the text can be found on pages xii and xiii and an explanation of the notation used in the result tables can be found on page xiv.

Footnotes

1. The L_∞ norm of a vector u is defined by $\|u\|_\infty = \max_i \{u_i\}$.
2. Digraph is a term used to describe a graph which has directed edges. The terms

INTRODUCTION

edge and path will be assumed to refer to a directed edge and a directed path when digraphs are being discussed.

Chapter 2

Value iteration methods

2.1 Basic Algorithms

The close relationship between Markov decision processes and systems of linear equations (if K_i consists of a single action for all i , (1.2) is simply a system of linear equations) motivates the application of solution methods for systems of linear equations to Markov decision processes. Value iteration methods (sometimes referred to as successive approximation methods) are analogous to iterative methods for solving systems of linear equations (Varga [52] describes many iterative methods for linear equations in detail) and the terminology used is similar. These algorithms are good candidates for solution methods for Markov decision processes because, as explained below, the properties of Markov decision processes guarantee convergence in most cases and the transition matrices for policies are often large and sparse, the conditions under which iterative methods are often preferred for systems of linear equations [56].

The algorithms to be considered in this study are defined as follows.

Pre-Jacobi (PJ)

$$v_i^n = (Ev^{n-1})_i \equiv \max_{k \in K_i} \left\{ H_i^k v^{n-1} \equiv r_i^k + \beta \sum_{j \in S} p_{i,j}^k v_j^{n-1} \right\} \quad n \geq 1 \quad (2.1)$$

v^0 specified

In some other texts this method is referred to simply as value iteration or successive approximation. The pre-Jacobi method is not often used to solve systems of linear equations.

VALUE ITERATION METHODS

Jacobi (J)

$$v_i^n = (Fv^{n-1})_i \equiv \max_{k \in K_i} \left\{ \frac{r_i^k + \beta \sum_{j \neq i} p_{i,j}^k v_j^{n-1}}{1 - \beta p_{i,i}^k} \right\} \quad n \geq 1 \quad (2.2)$$

v^0 specified

Although the convergence properties of Jacobi are known to be poor, it is worth considering this algorithm for the purpose of comparison.

Gauss-Seidel (GS)

$$v_i^n = \max_{k \in K_i} \left\{ \frac{r_i^k + \beta \sum_{j < i} p_{i,j}^k v_j^n + \beta \sum_{j > i} p_{i,j}^k v_j^{n-1}}{1 - \beta p_{i,i}^k} \right\} \quad n \geq 1 \quad (2.3)$$

v^0 specified

Unlike in PJ and Jacobi where the components of v^n can be calculated independently of each other, in GS the components of v^n have to be calculated in the order of increasing state.

Generalised Gauss-Seidel

$$v^n = Gv^{n-1} \quad n \geq 1 \quad (2.4)$$

where $(Gv)_i = \max_{k \in K_i} \left\{ H_i^k v \equiv \frac{r_i^k + \beta \sum_{j \in T_i} p_{i,j}^k (Gv)_j + \beta \sum_{j \notin T_i \cup \{i\}} p_{i,j}^k v_j}{1 - \beta p_{i,i}^k} \right\}$

and the state space can be reordered so that $T_i \subset \{1, 2, \dots, i-1\}$

v^0 specified

With the reordering of the state space the components of v^n can be calculated in the order of increasing state, but certain choices of T_i will allow several components of v^n to be calculated concurrently. With $T_i = \emptyset \forall i \in S$ generalised GS is equivalent to Jacobi and with $T_i = \{1, 2, \dots, i-1\} \forall i \in S$ generalised GS is equivalent to GS. Note that the definition of the function H_i^k in generalised GS is different from the definition of the function H_i^k in PJ. Using this notation simplifies the description of action elimination tests.

Variable generalised Gauss-Seidel

$$\begin{aligned}
 v^n &= G^n v^{n-1} \quad n \geq 1 \\
 \text{where } (G^n v)_i &= \max_{k \in K_i} \left\{ H_i^{n,k} v \equiv \frac{r_i^k + \beta \sum_{j \in T_i^n} p_{i,j}^k (G^n v)_j + \beta \sum_{j \notin T_i^n \cup \{i\}} p_{i,j}^k v_j}{1 - \beta p_{i,i}^k} \right\} \\
 \text{and the state space can be reordered so that } T_i^n &\subset \{1, 2, \dots, i-1\} \\
 v^0 &\text{ specified}
 \end{aligned} \tag{2.5}$$

Variable generalised GS is an abstraction of generalised GS in which the exact form of the update rule varies from iteration to iteration.

Successive over-relaxation (SOR) with relaxation parameter $\omega \geq 1$

$$\begin{aligned}
 v_i^n &= \omega \max_{k \in K_i} \left\{ \frac{r_i^k + \beta \sum_{j < i} p_{i,j}^k v_j^n + \beta \sum_{j > i} p_{i,j}^k v_j^{n-1}}{1 - \beta p_{i,i}^k} \right\} + (1 - \omega) v_i^{n-1} \quad n \geq 1 \\
 v^0 &\text{ specified}
 \end{aligned} \tag{2.6}$$

With $\omega = 1$ SOR is equivalent to GS. SOR can be generalised in the same way as GS, see section 5.2 for further details.

Since the update rules in all these methods are linear, the vector v^n may be expressed in terms of v^{n-1} by $v^n = \tilde{r}^\delta + \tilde{P}^\delta v^{n-1}$ where δ is the policy selected at the n^{th} iteration. $\tilde{P}^\delta$ is called the iteration matrix for δ . This notation is useful when describing error bounds and action elimination tests for the basic algorithms and will be used throughout this chapter.

The verification of the convergence of value iteration algorithms uses the following three properties.

Let B be the set of all bounded real valued functions on S .

Monotonic property

A mapping $H : B \rightarrow B$ is monotonic if for any $u, v \in B$ satisfying $u \leq v$, $Hu \leq Hv$.

Contraction property

A mapping $H : B \rightarrow B$ is a contraction if for some α satisfying $0 \leq \alpha < 1$, $\|Hu - Hv\|_\infty \leq \alpha \|u - v\|_\infty \quad \forall u, v \in B$.

Fixed point theorem for contraction mappings

If $H : B \rightarrow B$ is a contraction mapping then there exists a unique function $f^* \in B$ satisfying $Hf^* = f^*$ and $\lim_{n \rightarrow \infty} \|(\prod_{i=1}^n H)f - f^*\|_\infty = 0 \forall f \in B$.

Further information about the properties of contraction mappings and the proof of the fixed point theorem for contraction mappings can be found in [32].

For PJ it is well known that the mapping E defined in (2.1) has the monotonic and contraction properties, see for example Bertsekas [11]. Hence PJ converges to a unique value and by comparing (1.2) and (2.1) it is clear that this value is v^* and that the maximising actions for v^* comprise an optimal policy.

The series of results below apply to the mapping G in (2.4) and these will establish the convergence of generalised GS. Lemmas 2.6 and 2.7 are essentially specialisations of the results of Denardo [17].

Proposition 2.1

G is a monotonic mapping.

Proof

Let $u, v \in B$ satisfy $u \leq v$ and assume that the state space is ordered so that $T_i \subset \{1, 2, \dots, i-1\}$. $T_1 = \emptyset$ and

$$\begin{aligned} (Gu)_1 &= \frac{r_1^d + \beta \sum_{j \neq 1} p_{1,j}^d u_j}{1 - \beta p_{1,1}^d} \text{ for some } d \in K_1 \\ &\leq \frac{r_1^d + \beta \sum_{j \neq 1} p_{1,j}^d v_j}{1 - \beta p_{1,1}^d} \text{ since } p_{1,j}^d \geq 0 \forall j \in S \\ &\leq (Gv)_1. \end{aligned}$$

Suppose $(Gu)_j \leq (Gv)_j \forall j < i$, then $(Gu)_j \leq (Gv)_j \forall j \in T_i$ and

$$\begin{aligned} (Gu)_i &= \frac{r_i^d + \beta \sum_{j \in T_i} p_{i,j}^d (Gu)_j + \sum_{j \notin T_i \cup \{i\}} p_{i,j}^d u_j}{1 - \beta p_{i,i}^d} \text{ for some } d \in K_i \\ &\leq \frac{r_i^d + \beta \sum_{j \in T_i} p_{i,j}^d (Gv)_j + \sum_{j \notin T_i \cup \{i\}} p_{i,j}^d v_j}{1 - \beta p_{i,i}^d} \text{ since } p_{i,j}^d \geq 0 \forall j \in S \\ &\leq (Gv)_i. \end{aligned}$$

Therefore by induction $Gu \leq Gv$ and hence G is monotonic.

VALUE ITERATION METHODS

Lemma 2.2

$G(u + c) \leq Gu + \beta c$ and $G(u - c) \geq Gu - \beta c$ where $u \in B$ and $c \in \mathfrak{R}^+$.

Proof

Assume that the state space is ordered so that $T_i \subset \{1, 2, \dots, i-1\}$. $T_1 = \emptyset$ and

$$\begin{aligned}
 (G(u + c))_1 &= \max_{k \in K_1} \left\{ \frac{r_1^k + \beta \sum_{j \neq 1} p_{1,j}^k (u_j + c)}{1 - \beta p_{1,1}^k} \right\} \\
 &= \max_{k \in K_1} \left\{ H_1^k u + \frac{\beta c \sum_{j \neq 1} p_{1,j}^k}{1 - \beta p_{1,1}^k} \right\} \\
 &= \max_{k \in K_1} \left\{ H_1^k u + \frac{\beta c (1 - p_{1,1}^k)}{1 - \beta p_{1,1}^k} \right\} \\
 &\leq \max_{k \in K_1} \{H_1^k u + \beta c\} \text{ since } 0 \leq \beta < 1 \text{ and } 0 \leq p_{1,1}^k \leq 1 \\
 &= (Gu)_1 + \beta c.
 \end{aligned}$$

Suppose $(G(u + c))_j \leq (Gu)_j + \beta c \forall j < i$, then $(G(u + c))_j \leq (Gu)_j + \beta c \forall j \in T_i$

and

$$\begin{aligned}
 (G(u + c))_i &= \max_{k \in K_i} \left\{ \frac{r_i^k + \beta \sum_{j \in T_i} p_{i,j}^k (G(u + c))_j + \beta \sum_{j \notin T_i \cup \{i\}} p_{i,j}^k (u_j + c)}{1 - \beta p_{i,i}^k} \right\} \\
 &\leq \max_{k \in K_i} \left\{ \frac{r_i^k + \beta \sum_{j \in T_i} p_{i,j}^k ((Gu)_j + \beta c) + \beta \sum_{j \notin T_i \cup \{i\}} p_{i,j}^k (u_j + c)}{1 - \beta p_{i,i}^k} \right\} \\
 &= \max_{k \in K_i} \left\{ H_i^k u + \beta c \left(\frac{\beta \sum_{j \in T_i} p_{i,j}^k + \sum_{j \notin T_i \cup \{i\}} p_{i,j}^k}{1 - \beta p_{i,i}^k} \right) \right\} \\
 &\leq \max_{k \in K_i} \left\{ H_i^k u + \beta c \left(\frac{\sum_{j \neq i} p_{i,j}^k}{1 - \beta p_{i,i}^k} \right) \right\} \text{ since } 0 \leq \beta < 1 \text{ and } p_{i,j}^k \geq 0 \\
 &= \max_{k \in K_i} \left\{ H_i^k u + \beta c \left(\frac{1 - p_{i,i}^k}{1 - \beta p_{i,i}^k} \right) \right\} \\
 &\leq \max_{k \in K_i} \{H_i^k u + \beta c\} \text{ since } 0 \leq \beta < 1 \text{ and } 0 \leq p_{i,i}^k \leq 1 \\
 &= (Gu)_i + \beta c.
 \end{aligned}$$

Therefore by induction $G(u + c) \leq Gu + \beta c$.

A similar argument proves that $G(u - c) \geq Gu - \beta c$.

Proposition 2.3

For any $u, v \in B$ $\|Gu - Gv\|_\infty \leq \beta \|u - v\|_\infty$.

VALUE ITERATION METHODS

Proof

Let $c = \|u - v\|_\infty$ where $u, v \in B$.

$$u - c \leq v \leq u + c$$

$\Rightarrow G(u - c) \leq Gv \leq G(u + c)$ using proposition 2.1

$\Rightarrow Gu - \beta c \leq Gv \leq Gu + \beta c$ using lemma 2.2

$$\Rightarrow -\beta c \leq Gv - Gu \leq \beta c$$

$$\Rightarrow \|Gu - Gv\|_\infty \leq \beta \|u - v\|_\infty$$

Corollary 2.4

G is a contraction mapping.

Proof

Since $0 \leq \beta < 1$, proposition 2.3 shows that G is a contraction mapping.

Corollary 2.5

There exists a unique value $f^* \in B$ satisfying $Gf^* = f^*$ and generalised GS converges to f^* from any starting value.

Proof

Combine corollary 2.4 and the fixed point theorem for contraction mappings.

Lemma 2.6

When G is applied to f^* , the maximising actions comprise a policy δ with value $v^\delta = f^*$ and consequently $f^* \leq v^*$.

Proof

$$\begin{aligned} f_i^* = (Gf^*)_i &= H_i^{\delta(i)} f^* = \frac{r_i^{\delta(i)} + \beta \sum_{j \in T_i} p_{i,j}^{\delta(i)} (Gf^*)_j + \beta \sum_{j \notin T_i \cup \{i\}} p_{i,j}^{\delta(i)} f_j^*}{1 - \beta p_{i,i}^{\delta(i)}} \\ \Rightarrow (1 - \beta p_{i,i}^{\delta(i)}) f_i^* &= r_i^{\delta(i)} + \beta \sum_{j \neq i} p_{i,j}^{\delta(i)} f_j^* \text{ since } \beta p_{i,i}^{\delta(i)} < 1 \\ \Rightarrow f_i^* &= r_i^{\delta(i)} + \beta \sum_{j \in S} p_{i,j}^{\delta(i)} f_j^* \end{aligned}$$

Therefore f^* is a solution of (1.2) with K_i restricted to $\delta(i)$. Hence $v^\delta = f^*$.

Since v^* equals the maximum value attainable by any policy, $f^* \leq v^*$.

VALUE ITERATION METHODS

For any policy δ , let $(G^\delta v)_i$ be equal to $(Gv)_i$ with $K_i = \{\delta(i)\} \forall i \in S$. It follows that the properties of G shown above also apply to G^δ .

Lemma 2.7

$$f^* \geq v^*.$$

Proof

Let δ be any feasible policy and assume that the state space is ordered so that $T_i \subset \{1, 2, \dots, i-1\}$. $T_1 = \emptyset$ and

$$(G^\delta f^*)_1 = \frac{r_1^{\delta(1)} + \beta \sum_{j \neq 1} p_{1,j}^{\delta(1)} f_j^*}{1 - \beta p_{1,1}^{\delta(1)}} \leq (Gf^*)_1.$$

Suppose $(G^\delta f^*)_j \leq (Gf^*)_j \forall j < i$, then $(G^\delta f^*)_j \leq (Gf^*)_j \forall j \in T_i$ and

$$(G^\delta f^*)_i \leq \frac{r_i^{\delta(i)} + \beta \sum_{j \in T_i} p_{i,j}^{\delta(i)} (Gf^*)_j + \beta \sum_{j \notin T_i \cup \{i\}} p_{i,j}^{\delta(i)} f_j^*}{1 - \beta p_{i,i}^{\delta(i)}} \leq (Gf^*)_i.$$

Hence by induction $G^\delta f^* \leq Gf^* = f^*$. Repeated application of the monotonic property shows that $(\prod_{i=1}^n G^\delta) f^* \leq f^*$. Corollary 2.5 and proposition 2.6 show that $\lim_{n \rightarrow \infty} (\prod_{i=1}^n G^\delta) f^*$ equals v^δ . Hence $v^\delta \leq f^* \forall \delta \in \Delta$. Since $v^* = v^\delta$ for some δ , $v^* \leq f^*$.

Proposition 2.8

$v^* = f^*$ and when G is applied to f^* the maximising actions comprise an optimal policy.

Proof

This result is an immediate consequence of lemmas 2.6 and 2.7.

Corollary 2.5 and proposition 2.8 prove that generalised GS (and hence Jacobi and GS) converges to v^* from any starting approximation $v^0 \in B$.

Lemma 2.9

$$\|Gu - v^*\|_\infty \leq \beta \|u - v^*\|_\infty \forall u \in B.$$

Proof

$$\|Gu - v^*\|_\infty = \|Gu - Gv^*\|_\infty \leq \beta \|u - v^*\|_\infty \text{ using proposition 2.3.}$$

Proposition 2.10

Variable generalised GS converges to v^* from any starting approximation, $v^0 \in B$.

Proof

$$\begin{aligned} \left\| \left(\prod_{t=1}^n G^t \right) v^0 - v^* \right\|_\infty &\leq \beta \left\| \left(\prod_{t=1}^{n-1} G^t \right) v^0 - v^* \right\|_\infty \text{ using lemma 2.9} \\ &\leq \dots \leq \beta^n \|v^0 - v^*\|_\infty \end{aligned}$$

$$\text{Hence } \lim_{n \rightarrow \infty} \left\| \left(\prod_{t=1}^n G^t \right) v^0 - v^* \right\|_\infty = 0$$

For SOR the mapping defined by (2.6) need not be a contraction for $\omega > 1$ and therefore the algorithm will not always converge. However SOR must converge when $\omega = 1$ (because in this case SOR is equivalent to GS) and using continuity it is possible to argue that SOR converges if ω lies within a small range of values greater than 1.

2.2 Stopping rules for value iteration

In order to terminate these iterative procedures upper and lower bounds, $\bar{v}$ and $\underline{v}$, on the optimal value v^* are required. These bounds are calculated from the sequence of values $\{v^n\}$ and this study will concentrate on the two methods favoured in the comparisons performed by Porteus [42] and Thomas *et al* [49], namely the Porteus bounds (PB) [39] and the L_∞ norm bounds ($L_\infty B$). However there are several other ways of calculating such bounds, for example the MacQueen bounds [33] and the Thomas second order bounds [47].

The L_∞ norm bounds use the fact that

$$\begin{aligned} \text{if } \|v^n - v^{n-1}\|_\infty &\leq \alpha \|v^{n-1} - v^{n-2}\|_\infty \quad \forall n \geq 2 \text{ where } 0 \leq \alpha < 1 \\ \text{then } v^n - \frac{\alpha}{1-\alpha} \|v^n - v^{n-1}\|_\infty &\leq v^* \leq v^n + \frac{\alpha}{1-\alpha} \|v^n - v^{n-1}\|_\infty. \end{aligned} \tag{2.7}$$

VALUE ITERATION METHODS

Therefore if

$$\frac{\alpha}{1-\alpha} \|v^n - v^{n-1}\|_\infty < \frac{\epsilon}{2}$$

the convergence criterion in (1.3) is satisfied.

For PJ it is known that the condition in (2.7) is satisfied with $\alpha = \beta$. This is also the case in Jacobi, GS, generalised GS and variable generalised GS, see proposition 2.3. For SOR it is not possible to obtain a value for α in general, but by substituting

$$\frac{\|v^n - v^{n-1}\|_\infty}{\|v^{n-1} - v^{n-2}\|_\infty}$$

for α in (2.7) a heuristic stopping rule (heuristic $L_\infty B$) can be obtained. This heuristic is proposed in [49] where for all the test problems the values and policies found using SOR are consistent with those found using the other value iteration algorithms.

If $v^0 \leq v^*$ and $r_i^k \geq 0 \forall i \in S, \forall k \in K_i$ then $\{v^n\}$ is a non-decreasing sequence and therefore the $L_\infty B$ can be tightened as follows.

$$v^n \leq v^* \leq v^n + \frac{\alpha}{1-\alpha} \|v^n - v^{n-1}\|_\infty \quad (2.8)$$

This modification may lead to significant improvements in the performance of the value iteration algorithms, because the convergence criterion in (1.3) is now satisfied when

$$\frac{\alpha}{1-\alpha} \|v^n - v^{n-1}\|_\infty < \epsilon.$$

A similar result holds if $v^0 \geq v^*$ and $r_i^k \leq 0 \forall i \in S, \forall k \in K_i$.

With the transformation $\hat{r}_i^k = r_i^k - R \forall i \in S, \forall k \in K_i$ where $R = \min_{j \in S, \ell \in K_j} \{r_j^\ell\}$ and the initial approximation $v^0 = 0$, the above conditions are satisfied and $\hat{v}^*$, the solution of the resulting Markov decision process, is related to v^* by $v^* = \hat{v}^* + \frac{1}{1-\beta} R$.

The Porteus bounds are tighter bounds which exploit the monotonic and contraction properties of the mappings in PJ, Jacobi, GS and generalised GS. Since SOR does not generally have these properties, the PB cannot be applied to this algorithm. The PB state that

$$v^n + \frac{\alpha^n}{1-\alpha^n} L^n \leq v^* \leq v^n + \frac{\gamma^n}{1-\gamma^n} U^n \quad (2.9)$$

$$\text{where } L^n = \min_{i \in S} \{v_i^n - v_i^{n-1}\}, \alpha^n = \begin{cases} \min_{i \in S} \{\sum_{j \in S} \tilde{P}_{i,j}^\delta\} & \text{if } L^n \geq 0 \\ \max_{i \in S} \{\sum_{j \in S} \tilde{P}_{i,j}^\delta\} & \text{otherwise} \end{cases},$$

$$U^n = \max_{i \in S} \{v_i^n - v_i^{n-1}\}, \gamma^n = \begin{cases} \bar{\alpha} \equiv \max_{\pi \in \Delta, i \in S} \{\sum_{j \in S} \tilde{P}_{i,j}^\pi\} & \text{if } U^n \geq 0 \\ \underline{\alpha} \equiv \min_{\pi \in \Delta, i \in S} \{\sum_{j \in S} \tilde{P}_{i,j}^\pi\} & \text{otherwise} \end{cases},$$

and δ is the policy selected at the n^{th} iteration.

Hence the convergence criterion in (1.3) is satisfied if

$$\frac{\gamma^n}{1 - \gamma^n} U^n - \frac{\alpha^n}{1 - \alpha^n} L^n < \epsilon.$$

For PJ $\tilde{P}^\delta = \beta P^\delta$ and P^δ is a stochastic matrix. Hence $\alpha^n = \gamma^n = \beta$ and so, since calculating $\|v^n - v^{n-1}\|_\infty$ and calculating U^n and L^n both require two comparisons per state, applying the PB to PJ requires only one operation more than applying the $L_\infty B$. Further the eigenvalues of $\tilde{P}^\delta$ are equal to β times the eigenvalues of P^δ and the eigenvectors are the same. Since P^δ is stochastic its largest eigenvalue is 1 and $(1, 1, \dots, 1)^T$ is a corresponding eigenvector which is unique if P^δ has only one ergodic class¹. The PB are equivalent to an extrapolation along the direction of this eigenvector and it can be shown that the convergence of the PB is determined by the sub-radius² of the transition matrices for the policies used [37]. Hence in most cases these bounds are far superior to the $L_\infty B$.

For Jacobi, GS and generalised GS the iteration matrices only have equal row sums in exceptional circumstances. Consequently the PB have less impact on these algorithms than they do on PJ and require significantly more calculation than the $L_\infty B$. $\bar{\alpha}$ and $\underline{\alpha}$ are calculated by performing one iteration of the value iteration scheme with $r_i^k = 0 \forall i \in S, \forall k \in K_i$ starting from $(1, 1, \dots, 1)^T$ and $(-1, -1, \dots, -1)^T$ respectively. α^n is calculated by performing one iteration of the value iteration scheme with $K_i = \{\delta(i)\} \forall i \in S$ (where δ is the policy selected at the n^{th} iteration) and $r_i^k = 0 \forall i \in S, \forall k \in K_i$ starting from either $(1, 1, \dots, 1)^T$ or $(-1, -1, \dots, -1)^T$ depending on the sign of L^n . $\tilde{r}^\delta + \tilde{P}^\delta u = G^\delta u$ by definition. Let u satisfy $u_j = 1$ and $u_i = 0 \forall i \neq j$, then $u \geq 0$ so, by proposition 2.1, $G^\delta u \geq G^\delta 0$ and hence $\tilde{P}_{i,j}^\delta \geq 0 \forall i \in S$. Therefore for Jacobi, GS and generalised GS $\tilde{P}^\delta$ is non-negative. Lemma 2.2 shows that $\forall u \in B$ $G^\delta(u + 1) \leq G^\delta u + \beta$ and, by considering the case $u = 0$, this leads to $\sum_{j \in S} \tilde{P}_{i,j}^\delta \leq \beta$.

Hence for Jacobi, GS and generalised GS $0 \leq \alpha^n \leq \beta$ and $0 \leq \gamma^n \leq \beta$.

The PB can also be applied to variable generalised GS, but in this case, since the update rule used changes from iteration to iteration, $\bar{\alpha}$ and $\underline{\alpha}$ vary from iteration to iteration. Therefore in general, one of $\bar{\alpha}$ and $\underline{\alpha}$ will have to be calculated each time the PB are applied to variable generalised GS. This calculation is expensive as it involves a full iteration of the value iteration algorithm. Further overheads will be required to ensure that the update rule used at the n^{th} iteration is repeated when α^n is calculated. For these reasons the PB are not a practical stopping rule for variable generalised GS.

Thomas *et al* [48] derive a weaker form of the PB for Jacobi and GS which requires much less calculation. However, since the results of [49] show that these bounds do not reduce solution times significantly, these bounds will not be considered.

2.3 Elimination of non-optimal actions

The efficiency of value iteration methods can be greatly improved by the addition of a technique to eliminate actions which cannot occur in an optimal policy. MacQueen [34] was the first to describe a technique to identify non-optimal actions and the action elimination test developed uses the MacQueen bounds [33] which apply to PJ only.

MacQueen Action Elimination Test (MAE) for PJ

An action k in state i is non-optimal if

$$H_i^k v^{n-1} < v_i^n + \frac{\beta}{1-\beta}(L^n - U^n) \quad (2.10)$$

where L^n and U^n are as in (2.9).

The key result in the proof of this test is that if

$$\bar{v}, \underline{v} \text{ satisfy } \underline{v} \leq v^* \leq \bar{v} \text{ and } H_i^k \bar{v} < \max_{a \in K_i} \{H_i^a \underline{v}\} \quad (2.11)$$

then action k in state i is non-optimal. MacQueen proves this result for PJ and the following argument proves that the result is applicable to generalised GS also.

VALUE ITERATION METHODS

Lemma 2.11

If $u, v \in B$ satisfy $u \leq v$, then $H_i^k u \leq H_i^k v \forall k \in K_i, \forall i \in S$ where H_i^k is as defined in (2.4).

Proof

$$\begin{aligned} H_i^k u &= \frac{r_i^k + \beta \sum_{j \in T_i} p_{i,j}^k (Gu)_j + \beta \sum_{j \notin T_i \cup \{i\}} p_{i,j}^k u_j}{1 - \beta p_{i,i}^k} \\ &\leq \frac{r_i^k + \beta \sum_{j \in T_i} p_{i,j}^k (Gv)_j + \beta \sum_{j \notin T_i \cup \{i\}} p_{i,j}^k v_j}{1 - \beta p_{i,i}^k} \text{ since } G \text{ is monotonic and } p_{i,j}^k \geq 0 \\ &= H_i^k v \end{aligned}$$

Proposition 2.12

MacQueen's criterion for a non-optimal action (2.11) holds for generalised GS.

Proof

Suppose $\underline{v} \leq v^* \leq \bar{v}$ and $H_i^k \bar{v} < \max_{a \in K_i} \{H_i^a \underline{v}\} = (G\underline{v})_i$.

$v^* \leq \bar{v} \Rightarrow H_i^k v^* \leq H_i^k \bar{v}$ using lemma 2.11.

$\underline{v} \leq v^* \Rightarrow G\underline{v} \leq Gv^* = v^*$ using propositions 2.1 and 2.8.

Hence $H_i^k \bar{v} < (G\underline{v})_i \Rightarrow H_i^k v^* < v_i^*$ and therefore action k in state i is non-optimal.

For variable generalised GS the same argument can be used to show that if $\underline{v} \leq v^* \leq \bar{v}$ and $H_i^{n,k} \bar{v} < \max_{a \in K_i} \{H_i^{n,a} \underline{v}\}$ then action k in state i is non-optimal.

Since only heuristic bounds are known for SOR, there is no easy way to eliminate actions without the risk of eliminating optimal actions.

Porteus [40] uses (2.11) to develop two action elimination tests which are based on the PB and apply to PJ and generalised GS. These tests are stated below using the notation of (2.9). Although it is possible to develop versions of these tests which are applicable to variable generalised GS, since the PB are considered impractical for variable generalised GS (see section 2.2), the resulting tests are not useful.

Porteus Action Elimination Test — version 1 (PAE1)

An action k in state i is non-optimal if

$$H_i^k v^{n-1} < v_i^n + \frac{\alpha^n}{1 - \alpha^n} L^n - \frac{\gamma^n}{1 - \gamma^n} U^n. \quad (2.12)$$

Porteus Action Elimination Test — version 2 (PAE2)

An action k in state i is non-optimal if

$$H_i^k v^n < H_i^{\delta(i)} v^n + \frac{(\alpha^n)^2}{1 - \alpha^n} L^n - \frac{(\gamma^n)^2}{1 - \gamma^n} U^n \quad (2.13)$$

and the maximising actions for states in $\{j : j < i, j \in S\}$ are the same at the n^{th} and $(n + 1)^{\text{th}}$ iterations, where δ is the policy selected at the n^{th} iteration.

Note that for PJ PAE1 is equivalent to MAE. The right hand side of (2.10) and (2.12) can only be calculated after the n^{th} iteration is complete. Therefore, since it has proved unsuccessful to recalculate $H_i^k v^{n-1}$ whenever this value is required in MAE and PAE1 [49], these tests can only be efficiently implemented if the values $H_i^k v^{n-1} \forall i \in S, \forall k \in K_i$ can be stored. All the implementations of these tests in this study will use such an approach. Hastings and Mello [24] state that an application of MAE or PAE1 requires a separate pass through the remaining actions in every state at the end of each iteration. However it is not necessary to do this as these tests can easily be integrated into the basic calculation required by the value iteration algorithms as follows. When dealing with action k in state i at the $(n + 1)^{\text{th}}$ iteration, if

$$H_i^k v^{n-1} < v_i^n + \frac{\alpha^n}{1 - \alpha^n} L^n - \frac{\gamma^n}{1 - \gamma^n} U^n \text{ where } n \geq 1$$

then eliminate action k from K_i , otherwise calculate and store $H_i^k v^n$. Precautions have to be taken to ensure that no actions are eliminated during the first iteration.

PAE2 requires no additional storage to implement, but the order in which the calculation is performed must be modified from iteration to iteration. When dealing with state i at the $(n + 1)^{\text{th}}$ iteration $H_i^{\delta(i)} v^n$ is calculated first, followed by $H_i^k v^n$ for all other actions remaining in state i , with action elimination occurring where appropriate.

For PJ PAE2 applies even if the maximising actions for states in $\{j : j < i, j \in S\}$ change between the n^{th} and $(n + 1)^{\text{th}}$ iterations. Further $\delta(i)$ on the right hand side of (2.13) can be replaced with any action in K_i , so the largest known value for $H_i^k v^n$

can be used in place of $H_i^{\delta(i)}v^n$. During the iterations prior to the optimal policy being detected this modification can lead to a stronger test.

Since applying the PB to generalised GS requires a significant amount of calculation both initially and following every iteration, two new action elimination tests based on the $L_\infty B$ have been developed. These two tests apply to PJ, generalised GS and variable generalised GS.

L_∞ action elimination test — version 1 ($L_\infty AE1$)

An action k in state i is non-optimal if

$$H_i^k v^{n-1} < v_i^n - \frac{2\beta}{1-\beta} \|v^n - v^{n-1}\|_\infty. \quad (2.14)$$

Proof of $L_\infty AE1$

This test follows immediately from (2.12) using $L^n \geq -\|v^n - v^{n-1}\|_\infty$, $U^n \leq \|v^n - v^{n-1}\|_\infty$, $\alpha^n \leq \beta$ and $\gamma^n \leq \beta$ or it can be shown using MacQueen's criterion (2.11) with an alternative form of the $L_\infty B$ which states that

$$v^{n-1} - \frac{1}{1-\beta} \|v^n - v^{n-1}\|_\infty \leq v^* \leq v^{n-1} + \frac{1}{1-\beta} \|v^n - v^{n-1}\|_\infty.$$

Since all the values which appear in (2.14) are calculated at the same iteration and each iteration of variable generalised GS is equivalent to an iteration of some generalised GS algorithm, $L_\infty AE1$ also applies to variable generalised GS. Hence in variable generalised GS an action k in state i is non-optimal if

$$H_i^{n,k} v^{n-1} < v_i^n - \frac{2\beta}{1-\beta} \|v^n - v^{n-1}\|_\infty.$$

L_∞ action elimination test — version 2 ($L_\infty AE2$)

An action k in state i is non-optimal if

$$H_i^k v^n < H_i^d v^n - \frac{2\beta^2}{1-\beta} \|v^n - v^{n-1}\|_\infty \text{ for any } d \in K_i. \quad (2.15)$$

VALUE ITERATION METHODS

Proof of L_∞ AE2

This test is based on the form of the L_∞ B in (2.7).

For PJ $H_i^a v^n = r_i^a + \beta \sum_{j \in S} p_{i,j}^a v_j^n$. Therefore since $\sum_{j \in S} p_{i,j}^a = 1$, $H_i^a(v^n + c) = H_i^a v^n + \beta c$ where $c \in \mathfrak{R}$.

For generalised GS $H_i^a v^n = (G^\delta v^n)_i$ where $\delta(i) = a$, $\delta(j)$ equals the maximising action in state j at the $(n+1)^{\text{th}}$ iteration for $j < i$ and $\delta(j)$ is arbitrary for $j > i$. Therefore using lemma 2.2, $H_i^a(v^n + c) \leq H_i^a v^n + \beta c$ and $H_i^a(v^n - c) \geq H_i^a v^n - \beta c$ where $c \in \mathfrak{R}^+$.

$$\begin{aligned}
 \text{Suppose } H_i^k v^n &< H_i^d v^n - \frac{2\beta^2}{1-\beta} \|v^n - v^{n-1}\|_\infty \text{ for any } d \in K_i. \quad (2.16) \\
 H_i^k \left(v^n + \frac{\beta}{1-\beta} \|v^n - v^{n-1}\|_\infty \right) \\
 &\leq H_i^k v^n + \frac{\beta^2}{1-\beta} \|v^n - v^{n-1}\|_\infty \\
 &< H_i^d v^n - \frac{\beta^2}{1-\beta} \|v^n - v^{n-1}\|_\infty \text{ from (2.16)} \\
 &\leq H_i^d \left(v^n - \frac{\beta}{1-\beta} \|v^n - v^{n-1}\|_\infty \right) \\
 &\leq \max_{a \in K_i} \left\{ H_i^a \left(v^n - \frac{\beta}{1-\beta} \|v^n - v^{n-1}\|_\infty \right) \right\}
 \end{aligned}$$

Hence using (2.7), MacQueen's criterion (2.11) is satisfied and therefore action k in state i is non-optimal.

The proof of L_∞ AE2 can be reworked for variable generalised GS by substituting $H_i^{n+1,k}$, $H_i^{n+1,a}$ and $G^{n+1,\delta}$ for H_i^k , H_i^a and G^δ respectively, where $G^{n,\delta}$ is equal to G^n with $K_i = \{\delta(i)\}$. Hence in variable generalised GS an action k in state i is non-optimal if

$$H_i^{n+1,k} v^n < H_i^{n+1,d} v^n - \frac{2\beta^2}{1-\beta} \|v^n - v^{n-1}\|_\infty \text{ for any } d \in K_i.$$

L_∞ AE1 is applied in the same manner as MAE and PAE1. When applying L_∞ AE2 to actions in state i at the $(n+1)^{\text{th}}$ iteration, calculate $H_i^{\delta(i)} v^n$ first and use the largest known value for $H_i^d v^n$ (the technique which is used when PAE2 is applied to PJ).

If $\{v^n\}$ is a non-decreasing sequence then, since the L_∞ B can be tightened, L_∞ AE1 and L_∞ AE2 can be strengthened by increasing the right hand side of (2.14) to

$$v_i^n - \frac{\beta}{1 - \beta} \|v^n - v^{n-1}\|_\infty$$

and increasing the right hand side of (2.15) to

$$H_i^d v^n - \frac{\beta^2}{1 - \beta} \|v^n - v^{n-1}\|_\infty \text{ for some } d \in K_i.$$

For PJ applying PAE1 or PAE2 requires only one operation more than applying L_∞ AE1 or L_∞ AE2 and the PB are usually much tighter than the L_∞ B. Hence it is not worth considering either L_∞ AE1 or L_∞ AE2 in this case.

2.4 Reordering the state space

Although the ordering of the states does not affect the performance of PJ and Jacobi, it is an important consideration in GS and SOR. GS will solve a set of linear equations in one iteration if all the entries above the diagonal of the coefficient matrix are zero. Both GS and SOR take few iterations if the entries above the diagonal are small compared with the entries on or below the diagonal. (A matrix of this form will be described as having a weighted lower triangular (WLT) structure.) Similarly GS and SOR work well for Markov decision processes when most of the transition matrices used are of this form. An ordering of the states for which most transition matrices used have a WLT structure shall be referred to as a WLT ordering.

The reordering algorithms suggested by Porteus [41], can be used to find an ordering of the state space for which the transition matrix P for a given policy has a WLT structure, if such an ordering exists. One algorithm chooses the n^{th} state to minimise the n^{th} row sum of the GS iteration matrix corresponding to P subject to the $(n - 1)$ states already chosen. The simplest algorithm orders the states so that the column sums of P are decreasing.

Since the effort involved in reordering the states by these methods is less than three iterations of GS or SOR and in a Markov chain there is only one possible transition matrix, these techniques are likely to be effective when calculating the expected return over an infinite time horizon in a Markov chain. In Porteus [41] these techniques reduce

VALUE ITERATION METHODS

the time required to compute the infinite horizon expected return in a Markov chain using GS by between 10% and 40%. They will also be of use in the policy evaluation step of a policy iteration method for Markov decision processes (see [23] for details of policy iteration methods), as this involves the calculation of the infinite horizon expected return in a Markov chain.

However for value iteration methods reordering techniques are unlikely to be of benefit. (Thomas *et al* [48] report that the results were very poor, although it should be noted that a consequence of the procedure used to generate their test problems is that transition matrices are unlikely to have WLT structures and for such problems little benefit should be expected from reordering.) This is because the policy used changes from iteration to iteration and, since the structures of transition matrices for different policies can differ greatly, a new ordering must be calculated for each policy selected. Further, the policy selected at an iteration is not known until that iteration is complete, so the ordering calculated will only be of use if successive iterations select the same policy. Consequently reordering techniques are not implemented in any of the serial or parallel value iteration algorithms developed in later chapters.

When dealing with structured problems there is often some natural ordering suggested by the interpretation of the states which is a WLT ordering and where known such natural orderings will be used in this study.

The following four orderings will be considered.

1. A WLT ordering. Algorithms using this ordering will be denoted by LGS and LSOR.
2. The reverse of the ordering described in 1. Algorithms using this ordering will be denoted by UGS.
3. A random ordering. Algorithms using this ordering will be denoted by RGS and RSOR.
4. A particular ordering and the reverse of this ordering alternately. This has a computational advantage because at every iteration the value of one of the two

VALUE ITERATION METHODS

summations in (2.3) is the same as it was during the most recently completed iteration and so need not be recalculated [16]. Algorithms based on this method are called alternating GS (AGS). Those which use the ordering in 1 will be denoted by ALGS, while those which use the ordering in 3 will be denoted by ARGS.

Footnotes

1. Let G be the digraph corresponding to the stochastic matrix P^δ . $\tilde{S} \subset S$ is an ergodic class of P^δ if the fact that $i \in \tilde{S}$ is connected to $j \in S$ by a path in G implies that $j \in \tilde{S}$ and there exists a path in G connecting j to i .
2. The sub-radius of a matrix is the modulus of the eigenvalue with second largest modulus.

Chapter 3

Properties of Markov decision processes

3.1 Properties which affect the performance of iterative solution methods

The following arguments show that the nine properties of Markov decision processes listed in table 3.1 affect the performance of both serial and parallel versions of the iterative solution methods considered in this study. The computational results presented in chapters 4 and 5 will be explained in terms of these properties and will show that these properties can be used to determine which of the many iterative methods considered will perform best for a given problem.

Varying the discount factor or the tolerance has an obvious effect on performance — reducing the discount factor improves the convergence of all methods, while reducing the tolerance increases the number of iterations required by all methods.

At each iteration the number of operations required to update the value of an action in a state is equal to twice the number of transitions with positive probabilities arising from that action in that state and the number of operations required to calculate the maximising action for a state is equal to the number of actions in that state minus one. Hence the work required to update a state during an iteration is determined by properties 4 and 5. The number of states and the work per state affect the scope for parallelism — generally the greater the number of states and the greater the work per state, the more scope there is for parallelism.

Rapidly mixing problems have transition matrices with small sub-radii and so, since the convergence of PJ is governed by the sub-radii of the transition matrices used, PJ

Table 3.1: Important properties of Markov decision processes

1. Discount factor, β .
2. Tolerance, ϵ .
3. Number of states, m .
4. Average and variation of the number of actions per state.
5. Average and variation of the number of transitions per action.
6. Speed of mixing. In this study a problem is considered to be rapidly mixing if given any starting state, the probability of being in any other state n stages later converges quickly to a limit as n increases. More discussion on the speed of mixing in Markov chains can be found in [2].
7. Concentration of transition probabilities in the lower triangle of the transition matrices. When this concentration is high, the ordering of the states is a WLT ordering.
8. Closeness of the structures of the transition matrices of the non-optimal policies to that of the optimal policy.
9. Closeness of the values of the non-optimal policies to that of the optimal policy.

converges quickly.

It has already been noted in section 2.4, that when transition probabilities are concentrated in the lower triangle of the transition matrices used, convergence of GS and SOR is rapid.

When the transition matrices for the non-optimal policies have strong WLT structures, GS and SOR will exhibit good convergence during the iterations prior to the optimal policy being detected and when these transition matrices are rapidly mixing, PJ will exhibit good convergence during this stage. The results in chapter 4 will show that property 8 also influences the effectiveness of action elimination techniques.

When the values of the non-optimal policies are close to the value of the optimal policy, it is difficult to detect and eliminate non-optimal actions. The results in chapter 4 show that property 9 also affects the rate of convergence of value iteration methods.

3.2 Test problems based on real models

3.2.1 Howard's car replacement problem (HOWARD)

HOWARD is a replacement problem based on the car industry which was first used as an example in [28]. A stage in the problem is a period of three months. The state of the system is the age of the incumbent car in units of three months. A car of age 40 is considered to be in the worst condition possible and so ages no more. Therefore the problem has 40 states. (Note that when a decision is made the age of the incumbent car must be at least 1.) In each state at any stage, the possible actions are to replace the incumbent car with one of age k , where $0 \leq k \leq 39$, or to keep the incumbent car. Therefore there are 41 actions in all states. During any stage if the incumbent car suffers a serious breakdown a transition to state 40 is made, otherwise the incumbent car ages by 1 unit. If the incumbent car is replaced with one of age 0 or 39 or the incumbent car is kept when the system is in state 39 or 40, there is only one possible transition. In all other cases, there are two possible transitions. The data required are the cost of buying a car of age i , the trade-in value of a car of age i , the expected one stage operating cost for a car of age i and the probability that a car of age i will suffer a serious breakdown during a stage. This data can be found in [28]. A discount factor of 0.97 is used.

In every state under all but one action, there is a possibility of a transition to state 40. Further if the action chosen is to keep the incumbent car, then in addition there is a possibility of the incumbent car ageing by 1. It is known that the optimal policy for this type of replacement problem is a control limit policy [50]. This means that when $i > T$, the optimal action is to replace the incumbent car with one of age t and when $t < i \leq T$, the optimal action is to keep the incumbent car. When $i \leq t$ the optimal action is either to keep the incumbent car or to replace it with one of age t . Hence when $i \leq T$ all transitions are to states j where $j \geq i$. Therefore when states are ordered by decreasing age of car, the transition matrices for many policies, including the optimal, have WLT structures.

3.2.2 Single item inventory problem (INVENT1)

The single item inventory model has already been described in section 1.2. The example used as a test problem has a storage capacity of 119 individual units and hence 120 states. The fixed order cost is £10, the unit cost is £0.50, the holding cost is £0.035 per unit per period and the stock-out cost is £20 per unit. The lead time is one period and the demand per period is modelled by a Poisson distribution with mean 2.5 units per period. Probabilities less than 0.00005 are ignored so there are at most 12 transitions per action per state. Since the number of items ordered plus the number of items in stock cannot exceed the storage capacity, the number of actions in state i is $120 - i$.

If the lead time is zero, it is known that the optimal policy is an (s, S) policy [50]. This means that if $i > s$ then no order is placed, but if $i \leq s$ then $S - i$ items are ordered so that the level of stock is brought up to S . Although the lead time is one period in INVENT1, the optimal policy will still be such that when the level of stock is above a certain amount no order will be placed. Hence since the average demand per period is small compared to the storage capacity, for many states the optimal action will be to place no order. When the chosen action in state i is to place no order, all transitions go to states j such that $j \leq i$. Therefore, when the states are ordered by increasing number of units in stock, the transition matrix for the optimal policy has a WLT structure. With this ordering there will be many other policies which also have this property.

The procedure used to generate INVENT1 allows all the parameters in a single item inventory problem to be varied. This means it can be used to investigate the relationships between the various parameters and the optimal policy and value.

3.2.3 Multiple item inventory problems

This study will consider the class of multiple item inventory problem characterised by the following description. The stage depends upon the frequency with which stock levels are reviewed. Let the number of items be denoted by n , then the state is an

PROPERTIES OF MARKOV DECISION PROCESSES

n -vector whose i^{th} component is equal to the current stock level of item i and an action is an n -vector whose i^{th} component is equal to the number of units of item i ordered.

The state space is restricted by the maximum number of units of each item which can be stored and any number of linear constraints, so that

$$S = \{s \in Z^n : 0 \leq s_i \leq C_i^h \text{ for } 1 \leq i \leq n \text{ and } \forall t \ s \cdot w^t \leq W^t\}.$$

The action space for state s is constrained by the maximum order quantity of each item and the restrictions on the stock levels at the next stage, so that

$$K_s = \{k \in Z^n : 0 \leq k_i \leq C_i^o \text{ for } 1 \leq i \leq n \text{ and } s + k \in S\}.$$

The lead time can either be zero or one stage. The demands for the items are assumed to be independent and are modelled by Poisson distributions. The immediate reward consists of a fixed cost for placing an order plus a cost for each item. The cost for each item consists of a fixed cost for placing an order, a cost per unit ordered, a cost per unit of stock held and a cost per unit of demand which cannot be met.

The procedure used to generate these multiple inventory problems allows all the parameters described and the discount factor to be varied. This means it could be used to investigate the relationships between the various parameters and the optimal policy and value.

If the states are ordered by increasing number of units in stock and the chosen action in state i is to place no order, all transitions go to states j such that $j \leq i$. Therefore if there is a high proportion of states for which the optimal action is to place no order, the transition matrix for the optimal policy has a WLT structure when this ordering is used. The proportion of states for which the optimal action is to place no order will depend on the parameters used to generate the problem. For example if the expected demand for an item is close to the holding capacity for that item, there will be few states in which the optimal action is to place no order.

3.3 Generation of Markov decision processes

3.3.1 Motivation for problem generation

The lack of easily obtainable data for real examples of Markov decision processes, especially large ones, means that a sizable set of test problems can only be obtained by generating problems. The methods suggested by Porteus [42] and Thomas *et al* [48] to generate problems nearly always produce problems which are rapidly mixing (and do not allow any control over many of the other properties in table 3.1). However most real examples of Markov decision processes, including the two described in section 3.2, are slowly mixing. For this reason a new procedure was developed which allows control over the degree of mixing as well as all of the other properties in table 3.1.

3.3.2 Existing generation techniques

The procedure used by Thomas *et al* [48] always results in problems which are rapidly mixing and as a result PJ out performs GS in all cases. In this procedure the number of actions in a state is generated using a uniform distribution. The immediate rewards are also chosen from a uniform distribution, the range and mean of which are generated separately for each state using a uniform distribution. The transition probabilities for action k in state i are generated in two steps. First the values of j for which $p_{i,j}^k$ is to be non-zero are chosen using a series of Bernoulli trials. Following this the non-zero transition probabilities are generated using a $U(0, 1)$ distribution and normalised so that $\sum_{j \in S} p_{i,j}^k = 1$.

In the test problems used by Thomas *et al* [48] there are between 10 and 100 states, the sparsity of the transition matrices is approximately 10% and the discount factor is 0.9.

Porteus [42] uses a more complex procedure to generate Markov chains and this technique can be applied repeatedly to generate Markov decision processes. r_i , the immediate reward in state i , is set equal to $10i$ if $1 \leq i \leq 10$, set equal to 0 if $i = M$ and generated from a $U(0, 1)$ distribution otherwise. This elaborate procedure is intended to eliminate the possibility of a nearly constant optimal value vector. Two parameters are

PROPERTIES OF MARKOV DECISION PROCESSES

used in the generation of the transition probabilities — the maximum number of non-zero transition probabilities per action per state and the desired maximum transition probability, *NZ* and *MP* respectively. In the generation of the transition probabilities for state *i*, a transition is selected at random from those not already allocated a non-zero probability and assigned a probability according to the expression $\min(q, X)$, where *q* is generated from a $U(0, MP)$ distribution and *X* is equal to 1 minus the sum of the probabilities already generated for state *i*. This step is repeated until *NZ* non-zero transitions have been generated or *X* is less than or equal to zero. The final transition probability generated is adjusted so that the sum of the transition probabilities for state *i* equals 1.

The *MP* parameter allows some control over the speed of mixing. When *MP* is low, mixing is rapid and *PJ* performs well and when *MP* is high, mixing is slower and consequently *GS* performs slightly better than *PJ*.

Porteus [42] uses test problems which have between 50 and 200 states and a discount factor of 0.9. In the generation of these problems *NZ* varies between 3 and 10 (in such a way that the sparsity is between 1.5% and 20%) and *MP* varies between 0.18 to 0.9.

3.3.3 PROCON — a new problem generator

Outline

PROCON is a new procedure for generating Markov decision processes which allows control over all of the properties in table 3.1. PROCON consists of three stages. The first stage generates a connected weighted digraph with the state space of the Markov decision process as the vertex set. The structure of this digraph can be used to control the speed of mixing in the transition matrix for the optimal policy. At the second stage additional weighted edges are added to the original digraph, so that the sparsity of the digraph is brought up to the desired level, and immediate rewards are generated. This results in a one action Markov decision process which comprises the reward and transition data for the optimal policy in the generated problem. At the third stage the problem generation is completed by the addition of non-optimal actions.

PROPERTIES OF MARKOV DECISION PROCESSES

Detailed description of the procedure

Stage 1: Generation of the transition structure for the optimal policy

There are three parameters for the first stage — M , LEN and CYC . M is the number of states. From an arbitrary starting state, a random walk of length LEN is constructed through the state space. At every step, the walk is prevented from revisiting the current state or any of the last CYC states on the walk. In order to remove the possibility of an absorbing state¹, the destination state at the final step of the walk must be a previously visited state. Let Ψ be the set of states in this random walk. While there exists an isolated state j , a random walk from j is constructed through the state space terminating as soon as a state in Ψ is reached. This results in a connected digraph. Each edge in this digraph is assigned a weight (these weights have a $U(0, 1)$ distribution) and then the weights are normalised so that the sum of the weights on the edges out of each state is equal to 1.

Stage 2: Finalising reward and transition data for the optimal policy

There are eight parameters for the second stage — NZ , SDT , $JUMP$, CUT , A , B , C , and $BETA$. μ , the number of additional edges required to bring the total number of transitions up to $NZ \times M$ divided by the number of the edges already generated, is calculated. For each edge $u \rightarrow_e v$ in the digraph an additional $\lfloor X_e \rfloor$ edges are generated, where X_e has a $N(\mu, SDT)$ distribution. All of these additional edges are of the form $u \rightarrow z$ where there exists a path from u to z or from z to u in the original digraph consisting of at most $JUMP$ edges and if $u \in \Psi$ then $z \in \Psi$. The second condition on the choice of destination means that there is no path from a state in Ψ to a state not in Ψ . This ensures that some states (namely those not in Ψ) are transient². A proportion of the weight on edge e no greater than $CUT\%$ is removed and distributed randomly between the $\lfloor X_e \rfloor$ additional edges. At this point the weighted digraph corresponds to the transition matrix for the optimal policy in the Markov decision process being generated. r , the immediate reward vector for the optimal policy, is then calculated using $r_{\alpha(i)} = A + B(i) + C(i)$ where A is a constant, B is a decreasing function which introduces a trend to the value of the optimal policy, C is some function which introduces a random element to the

PROPERTIES OF MARKOV DECISION PROCESSES

value of the optimal policy and $\alpha(i)$ is the i^{th} state encountered in the first stage of the generation procedure. This one action Markov decision process is then solved using PJ with a discount factor of BETA and v^o , the resulting approximation to the optimal value, is stored. The number of iterations required is an indication of the degree of mixing in the optimal policy.

Stage 3: Addition of non-optimal actions

There are five parameters for this stage — MA, SDA, DEL, MOVE and PERT. The number of actions in state i is set equal to $\max(1, \lfloor A_i \rfloor)$ where A_i has a $N(\text{MA}, \text{SDA})$ distribution. For a non-optimal action k in state i , the transition probabilities $p_{i,j}^k$ are generated in the following manner. $p_{i,j}^k$ is set equal to $T_{i,j+\theta}$ where θ is generated separately for each i and k using a discrete uniform distribution with range $(-\text{MOVE}, \text{MOVE})$ and T is the transition matrix for the optimal policy. $p_{i,j}^k$ is then perturbed by multiplying by the factor $1 + \gamma_j \text{PERT}$, where γ_j has a $U(-1, 1)$ distribution, and finally the transition probabilities are normalised so that $\sum_{j \in S} p_{i,j}^k = 1$. For a non-optimal action k in state i , the immediate reward r_i^k is chosen from a

$$U \left(v_i^o - \text{BETA} \sum_{j \in S} p_{i,j}^k v_j^o - \text{DEL}, v_i^o - \text{BETA} \sum_{j \in S} p_{i,j}^k v_j^o - \frac{\text{DEL}}{2} \right)$$

distribution. Assuming v^o is a close approximation to v^* , this ensures that

$$v_i^* - \text{DEL} \leq H_i^k v^* \leq v_i^* - \frac{\text{DEL}}{2} \quad \forall k \in K_i.$$

Examining the action elimination tests described in section 2.3 shows that an action k in state i can be eliminated if $H_i^d v^{n-1} - H_i^k v^{n-1} > \|\bar{v} - \underline{v}\|_\infty$ using PAE1 or $L_\infty \text{AE1}$ or if $H_i^d v^n - H_i^k v^n > \beta \|\bar{v} - \underline{v}\|_\infty$ using PAE2 or $L_\infty \text{AE2}$ where d is some (not necessarily arbitrary) action in K_i and $\bar{v}$ and $\underline{v}$ are the upper and lower bounds on v^* calculated at the end of the n^{th} iteration. Convergence is detected when $\|\bar{v} - \underline{v}\|_\infty < \epsilon$. Hence reducing DEL is likely to increase the proportion of iterations which are completed before any actions can be eliminated, because the algorithms have to be closer to convergence. Further if $\frac{\text{DEL}}{2} < \epsilon$ then in every state i a number of non-optimal actions $k \in K_i$ satisfy $v_i^* - H_i^k v^* < \epsilon$ and it is unlikely that these actions can be eliminated before convergence is detected. The expected proportion of actions in this category is equal to $\frac{2\epsilon - \text{DEL}}{\text{DEL}}$.

PROPERTIES OF MARKOV DECISION PROCESSES

Control of properties

The parameters **LEN** and **CYC** allow control over the degree of mixing by controlling the lengths of paths between pairs of states. Increasing **LEN** tends to increase the number of short paths, while increasing **CYC** tends to increase the minimum path length between pairs of states. By making the length of paths between pairs of states large, slowly mixing problems are generated.

If the parameters **JUMP** and **CUT** are small then the edges added at the second stage will join states which lie close together in the digraph generated at the first stage and these new edges will be assigned small weights. Consequently the speed of mixing in the original digraph will not be changed greatly.

Suitable choices of **A**, **B** and **C** reduce the likelihood of a near constant optimal value vector which may produce spurious good results when the starting values of the iterative procedure are constant.

Reducing **BETA** improves the convergence of all the iterative methods.

The effect of varying these parameters is illustrated in table 3.2. In all the examples shown in this table **NZ** = 6 and **SDT** = 1. These two parameters are not varied because they do not affect the convergence of the iterative methods. However they do affect the work per state. All actions in a given state result in the same number of non-zero transitions, but the number of non-zero transitions may vary from state to state. **NZ** determines the mean of the number of non-zero transitions per action and **SDT** controls the variation of this number with state.

For the following reasons the variation in the number of transitions per action also depends upon the choice of **LEN** and **CYC**. The first **CYC** + 1 steps of the random walk generated during the first stage of the PROCON procedure visit **CYC** + 2 different states. After this, the walk proceeds for a further **LEN** - **CYC** - 1 steps during which some of the states on the walk may be revisited. If $\frac{\text{CYC}}{\text{LEN}} \geq \frac{1}{2}$ then only the first **LEN** - **CYC** - 1 states on the walk can be revisited and no state can be revisited more than once. This means that when the walk is complete the out-degree of states in Ψ is either 1 or 2. The expected number of transitions per action for a state in the final problem

PROPERTIES OF MARKOV DECISION PROCESSES

Table 3.2: Varying the parameters in the PROCON procedure

M	LEN	CYC	JUMP	CUT	A	B(x)	C(x)	BETA	N _{iter}
1200	502	500	4	20%	10	$\frac{100}{x+1}$	$3 \sin x$	0.995	2430
	1000								343
	5000								26
1200	1000	500 800	4	20%	10	$\frac{100}{x+1}$	$3 \sin x$	0.995	343 1978
200	200	180	4	20%	10	$\frac{100}{x+1}$	$3 \sin x$	0.995	2407
			20	20%					1049
			20	80%					395
1200	1000	500	4	20%	10	$\frac{100}{x+1}$	$3 \sin x$	0.995	343
					10	$\frac{1}{x+1}$	$\frac{\sin x}{100}$		214
					10	0	0		1
1200	1000	500	4	20%	10	$\frac{100}{x+1}$	$3 \sin x$	0.995	343
								0.750	32
								0.500	14

N_{iter} denotes the number of iterations required to solve the one action problem using PJ & PB.

is proportional to the out-degree of that state at the end of the first stage. Hence if the states are in the order in which they are encountered in the PROCON procedure, some of the first $LEN - CYC - 1$ states will have significantly more transitions per action than the other states in Ψ . States which remain isolated after this walk is complete are connected by constructing a series of random walks which terminate when a state in Ψ is encountered. If the number of states in $\bar{\Psi}$ is large then this process forms a complex network involving the states in $\bar{\Psi}$. Further the later a state is encountered during the process, the less likely it is to be revisited. Hence when all states have been connected, the average out-degree of states in $\bar{\Psi}$ will be large compared to that of states in Ψ and the out-degree of states in $\bar{\Psi}$ will exhibit a downward trend when states are in the order in which they are encountered in the PROCON procedure. Therefore the average number of transitions per action for states in $\bar{\Psi}$ will be greater than that for states in Ψ and the number of transitions per action for states in $\bar{\Psi}$ will exhibit a downward trend.

This phenomenon is illustrated in figure 3.1 for a problem in which $\frac{CYC}{LEN} = \frac{3}{4}$, Ψ consists of 775 states and $\bar{\Psi}$ consists of 425 states (this problem is called GEN1, see



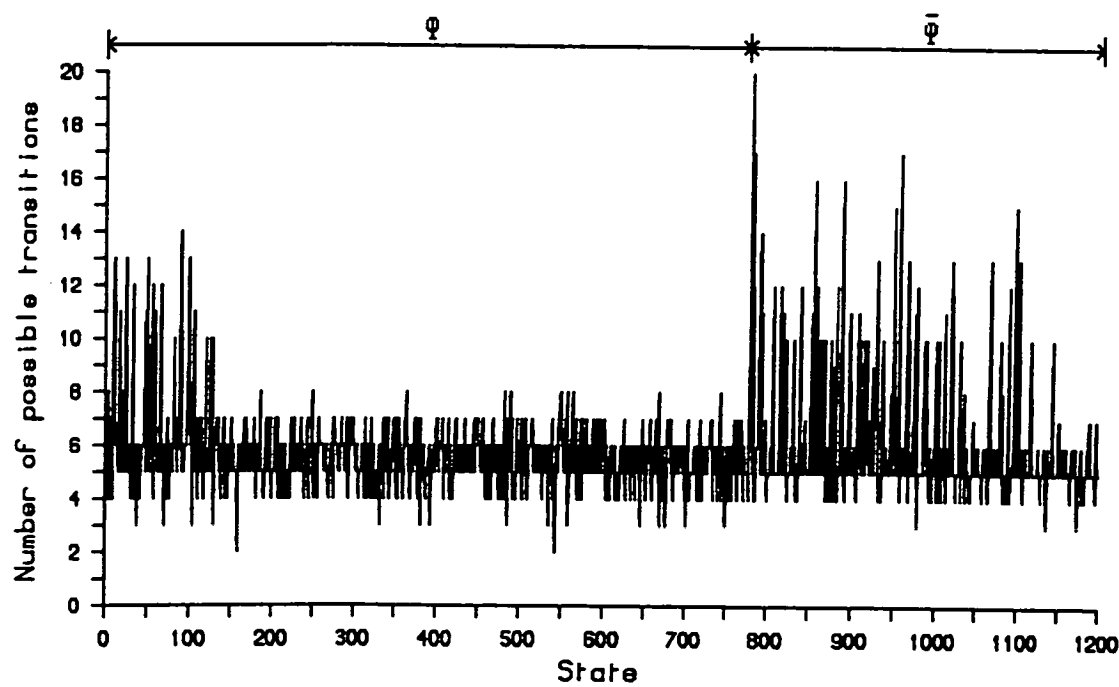


Figure 3.1: Variation in transitions per action with state for GEN1.

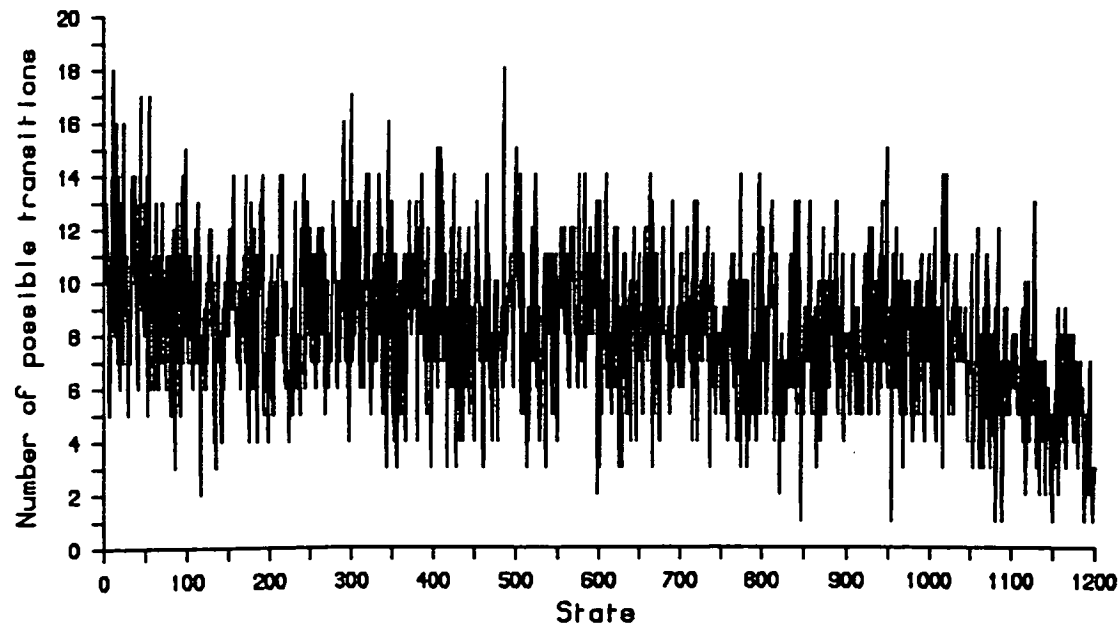


Figure 3.2: Variation in transitions per action with state for GEN2.

PROPERTIES OF MARKOV DECISION PROCESSES

section 3.4). This is in contrast with figure 3.2 which shows a plot of the number of transitions per action against state for a problem in which $\frac{CYC}{LEN} = \frac{1}{10}$, Ψ consists of 1197 states and $\bar{\Psi}$ consists of only 3 states (this problem is called GEN2, see section 3.4). The number of transitions per action exhibits a downward trend because, in this case, the later a state is encountered in the PROCON procedure, the less likely it is to be revisited. However this problem displays none of the structure apparent in figure 3.1.

When the ratio of CYC to LEN is large, the transition matrix for the optimal policy will be slowly mixing and if the states are put into the order in which they are encountered in the PROCON procedure, this transition matrix has the structure shown in figure 3.3. If the order of the states in Ψ is reversed then a WLT structure is obtained. The PROCON procedure puts the states into this order unless a random ordering of the states is requested.

MA and SDA determine the mean and variation for the number of actions per state. MOVE and PERT control the closeness of the structure of the transition matrices for non-optimal policies to that of the optimal policy. DEL controls the closeness of the values of non-optimal policies to that of the optimal policy.

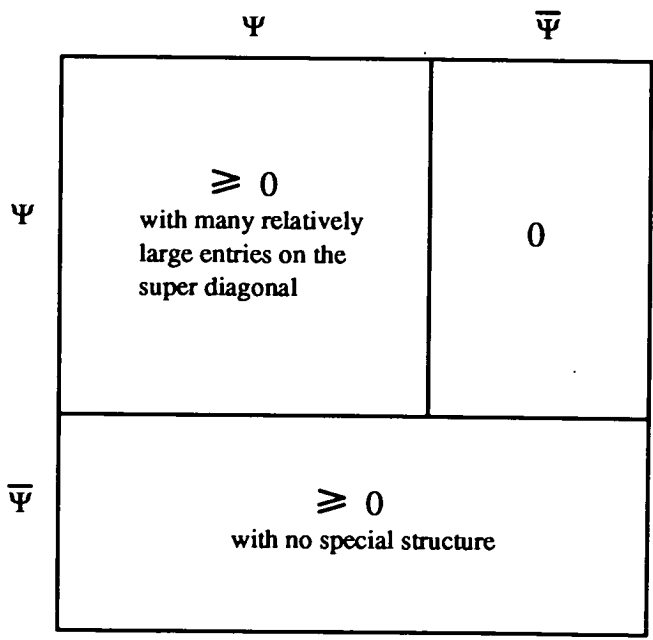


Figure 3.3: Structure of the transition matrix for the optimal policy.

The effect of varying these parameters will be discussed in chapters 4 and 5 which present computational results for problems generated using the PROCON procedure.

3.4 Generated test problems

The PROCON procedure was used to generate a series of test problems with distinct properties.

In the three GEN1 problems (GEN1FF, GEN1CV and GEN1CS) the transition matrix for the optimal policy and the value of the optimal policy are the same. This optimal policy is slowly mixing. In GEN1FF and GEN1CV the transition probabilities corresponding to non-optimal actions produce random transition structures (which are rapidly mixing), while in GEN1CS they produce transition matrices with a similar structure to that produced by the optimal policy. In GEN1CV about $\frac{1}{3}$ of the actions cannot be eliminated before convergence is detected, but in GEN1FF and GEN1CS all actions can be eliminated before convergence is detected. Note that the letters following GEN1 relate to the closeness of the structures and the values of non-optimal policies to those of the optimal policy, see properties 8 and 9 in table 3.1. CS stands for close structures, CV for close values and FF for far structures and values.

All the policies in GEN2 are rapidly mixing and about $\frac{1}{3}$ of the actions cannot be eliminated before convergence is detected.

Assessing the performance of some of the parallel algorithms developed in chapter 5 requires test problems with much larger state spaces than those of the problems considered so far. For this purpose a series of large scale problems was generated. The most important difference between these problems lies in the scope for parallelism. This is varied by adjusting the number of states and the average number of actions per state. Details of these problems are given in table 3.5.

3.5 Generated problems with special transition structures

The existence of a known WLT structure is very important to the performance of GS based algorithms. To assess how successfully different WLT structures can be exploited

PROPERTIES OF MARKOV DECISION PROCESSES

by the various GS algorithms developed later in this study, six large scale problems differing only in transition structure were generated. In each problem there is one action per state (so there is only one possible transition matrix) and almost always six possible transitions arising from each action. In GEN6A, GEN6B and GEN6C 90% of the weight in the transition matrix lies in a dense band, of width 3 elements, 2 elements and 1 element respectively, situated just below the diagonal. The remaining weight is distributed randomly. In GEN6D 90% of the weight lies in a dense band of width 3 elements situated well below the diagonal. Again the remaining weight is distributed randomly. In GEN6E the positive entries in the transition matrix are distributed at random on or below the main diagonal so that the matrix is lower triangular. Note that for the first five states in this problem there are fewer than six possible transitions. In GEN6F the transition matrix has a random structure. Further properties of these problems are given in table 3.6.

3.6 Summary of the properties of the test problems

Table 3.3: Properties of the small test problems

Property	HOWARD	INVENT1	GEN1FF	GEN1CV	GEN1CS	GEN2
1. Discount β	0.97	0.95	0.995	0.995	0.995	0.995
2. Tolerance ϵ	0.001	0.001	0.001	0.001	0.001	0.001
3. States	40	120	1200	1200	1200	1200
4. Actions	41	Systematic from 1–120	Normally distributed with mean 16 and variance 1.5			
5. Transitions	Mostly 2	Mostly 12	Average of 6			Average of 8
6. Mixing	Becoming slower as policy converges				Slow	Rapid
7. WLT	Strong in many including optimal		Strong in optimal Weak in others		Strong in all	None
8. Structure	Variable	Variable	Far	Far	Close	Far
9. Values	Wide spread		Far	Close	Far	Close

These problems are used as initial test problems for all the solution methods considered in this study.

PROPERTIES OF MARKOV DECISION PROCESSES

Table 3.4: Properties of the multiple item inventory problems

Property	mi1	mi2	mi3	mi4	mi5
1. Discount β	0.99	0.99	0.99	0.99	0.99
2. Tolerance ϵ	0.001	0.001	0.001	0.001	0.001
3. States	540	540	540	540	540
4. Actions	Ranging from 1 to 48				
5. Transitions	Ranging from 1 to 540				
6. Mixing	Becoming slower as policy converges				
7. WLT	Strong in many				
8. Structure	Variable				
9. Values	Wide spread				

These problems are used to assess the performance of the parallel reward revision algorithms developed in chapter 6.

Table 3.5: Properties of the large test problems

Property	GEN3	GEN4	GEN5 [†]
1. Discount β	0.995	0.995	0.995
2. Tolerance ϵ	0.001	0.001	0.001
3. States	15000	60000	30000
4. Actions	1	Normally distributed with mean 3 and variance 1	Constant (in the range [1, 10]) for each problem
5. Transitions	Average of 6	Average of 6	Average of 6
6. Mixing	Slow	Slow	Slow
7. WLT	Strong in all	Strong in all	Strong in all
8. Structure	—	Close	Close
9. Values	—	Close	Far

[†] GEN5 is a family of ten 30000 state problems which differ from each other only in the number of actions per state.

These problems are used in section 5.8 to determine the effectiveness of the phased pipeline algorithm.

Table 3.6: Properties of the generated problems with special transition structures

Property	GEN6A	GEN6B	GEN6C	GEN6D	GEN6E	GEN6F
1. Discount β	0.995	0.995	0.995	0.995	0.995	0.995
2. Tolerance ϵ	0.001	0.001	0.001	0.001	0.001	0.001
3. States	29304	29304	29304	29304	29304	29304
4. Actions	1	1	1	1	1	1
5. Transitions	6	6	6	6	Mostly 6	6
6. Mixing	Slow	Slow	Slow	Slow	Rapid	Rapid
7. WLT	Strong	Strong	Strong	Strong	Strong	None

These problems are used in section 5.8 to compare the rates of convergence of several GS based versions of the phased pipeline algorithm.

Footnotes

1. A state i is called an absorbing state if the probability that the system is in state i after n stages tends to 1 as n tends to infinity.
2. A state i is called a transient state if the probability of being in state i after n stages tends to 0 as n tends to infinity.

Chapter 4

Serial value iteration

4.1 Introduction

This chapter compares the performance of the value iteration algorithms described in chapter 2 on a single transputer using the six small test problems described in table 3.3. Several earlier papers have presented such a comparison. Porteus [42] and Thomas *et al* [49] compare the performance of several value iteration algorithms using generated problems which tend to be rapidly mixing and as a consequence the computational results presented favour PJ. Hendrikx *et al* [25] present results for serial value iteration on three different problem types, but do not include any analysis. The comparison presented in this chapter is novel because it uses six problems with different properties and explains the performance of each algorithm in terms of these properties. This insight makes it possible to predict which algorithms will perform well for a given problem.

The comparison and timings are also important for the chapters which deal with the parallel implementation of value iteration algorithms for Markov decision processes because they identify the most successful serial algorithms, which provides a focus for the effort of parallelisation, and they provide a base with which to relate the solution times for the parallel algorithms.

This chapter also evaluates two action elimination tests for GS. Action elimination for GS has often been overlooked in the past, for example Thomas *et al* [49] conclude that it is not worthwhile because PJ performs so well on the test problems considered.

Another feature of this comparison is that the size of the test problems is much

SERIAL VALUE ITERATION

larger than those used previously. Since one aim of this study is to develop algorithms which are capable of solving large scale problems, it is important to understand how problem size affects the performance of the algorithms.

4.2 Results

The programs used have been designed specifically for serial machines (i.e. they are entirely sequential). The times presented in the result tables are in seconds and were obtained using a single T800-20MHz transputer. The solution time does not include the time required to obtain the data initially. In HOWARD and INVENT1 the data is generated locally on the transputer and the time required for this is small compared to the solution time. In the four generated problems the data is read from disk and the time required is highly variable due to multiuser access. See page xiv for details of the notation used in the tables.

Figure 4.1 shows the number of non-optimal actions in the policy selected at each iteration of PJ & PB and LGS & $L_\infty B$ for each test problem. Figure 4.2 shows the number of feasible actions at each iteration of PJ & PB with PAE2 and LGS & $L_\infty B$ with $L_\infty AE2$ for each test problem.

Table 4.1: Serial value iteration and HOWARD

Algorithm	N_{iter}	T_{soln}	t_{aver}	T_{stan}
PJ & PB	141	6.33	0.045	3.97
PJ & PB with MAE	141	1.79	0.013	1.13
PJ & PB with PAE2	141	1.59	0.011	1.00
J & $L_\infty B$	530	23.06	0.044	14.48
RGS & $L_\infty B$	322	14.43	0.045	9.06
LGS & $L_\infty B$	77	3.47	0.045	2.18
UGS & $L_\infty B$	486	21.79	0.045	13.68
RGS & PB	309	14.32	0.046	8.99
LGS & PB	75	3.49	0.047	2.19
ARGS & $L_\infty B$	334	10.05	0.030	6.31
ALGS & $L_\infty B$	148	4.43	0.030	2.78
LGS & $L_\infty B$ with $L_\infty AE1$	77	2.65	0.034	1.66
LGS & $L_\infty B$ with $L_\infty AE2$	77	2.19	0.028	1.38
LGS & PB with PAE1	75	2.75	0.037	1.73
LGS & PB with PAE2	75	2.31	0.031	1.45

Table 4.2: Serial value iteration and INVENT1

Algorithm	N_{iter}	T_{soln}	t_{aver}	T_{stan}
PJ & PB	93	57.41	0.62	7.30
PJ & PB with MAE	93	14.52	0.16	1.84
PJ & PB with PAE2	93	13.97	0.15	1.78
J & $L_\infty B$	215	132.64	0.62	16.85
RGS & $L_\infty B$	118	72.42	0.61	9.20
LGS & $L_\infty B$	29	17.75	0.61	2.26
UGS & $L_\infty B$	200	122.82	0.61	15.61
RGS & PB	112	71.12	0.63	9.04
LGS & PB	28	18.58	0.66	2.36
ARGS & $L_\infty B$	158	53.30	0.34	6.77
ALGS & $L_\infty B$	56	18.98	0.34	2.41
LGS & $L_\infty B$ with $L_\infty AE1$	29	7.90	0.27	1.00
LGS & $L_\infty B$ with $L_\infty AE2$	29	7.87	0.27	1.00
LGS & PB with PAE1	28	8.55	0.31	1.09
LGS & PB with PAE2	28	8.56	0.31	1.09

Table 4.3: Serial value iteration and GEN1FF

Algorithm	N_{iter}	T_{soln}	t_{aver}	T_{stan}
PJ & PB	613	594.9	0.97	3.93
PJ & PB with MAE	613	236.0	0.39	1.56
PJ & PB with PAE2	613	220.0	0.36	1.45
J & $L_\infty B$	1980	1929.3	0.97	12.75
RGS & $L_\infty B$	977	957.6	0.98	6.33
LGS & $L_\infty B$	198	194.1	0.98	1.28
UGS & $L_\infty B$	1758	1697.2	0.97	11.21
RGS & PB	893	939.5	1.05	6.21
LGS & PB	187	198.3	1.06	1.31
ARGS & $L_\infty B$	1364	815.1	0.60	5.39
ALGS & $L_\infty B$	362	216.9	0.60	1.43
LGS & $L_\infty B$ with $L_\infty AE1$	198	168.8	0.85	1.12
LGS & $L_\infty B$ with $L_\infty AE2$	198	151.3	0.76	1.00
LGS & PB with PAE1	187	170.5	0.91	1.13
LGS & PB with PAE2	187	153.8	0.82	1.02

Table 4.4: Serial value iteration and GEN1CV

Algorithm	N_{iter}	T_{soln}	t_{aver}	T_{stan}
PJ & PB	230	223.2	0.97	1.57
PJ & PB with MAE	230	242.3	1.05	1.70
PJ & PB with PAE2	230	223.4	0.97	1.57
J & $L_\infty B$	1318	1284.4	0.97	9.04
RGS & $L_\infty B$	640	627.4	0.98	4.41
LGS & $L_\infty B$	145	142.1	0.98	1.00
UGS & $L_\infty B$	1128	1089.2	0.97	7.66
RGS & PB	556	585.8	1.05	4.12
LGS & PB	134	142.7	1.06	1.00
ARGS & $L_\infty B$	910	544.3	0.60	3.83
ALGS & $L_\infty B$	260	156.1	0.60	1.10
LGS & $L_\infty B$ with $L_\infty AE1$	145	167.8	1.16	1.18
LGS & $L_\infty B$ with $L_\infty AE2$	145	150.4	1.04	1.06
LGS & PB with PAE1	134	166.2	1.24	1.17
LGS & PB with PAE2	134	149.4	1.12	1.05

Table 4.5: Serial value iteration and GEN1CS

Algorithm	N_{iter}	T_{soln}	t_{aver}	T_{stan}
PJ & PB	1508	1464.0	0.97	7.96
PJ & PB with MAE	1508	1233.0	0.82	6.70
PJ & PB with PAE2	1508	1148.2	0.76	6.24
J & $L_\infty B$	2995	2900.5	0.97	15.77
RGS & $L_\infty B$	1382	1346.2	0.97	7.32
LGS & $L_\infty B$	230	224.0	0.97	1.22
UGS & $L_\infty B$	2756	2642.4	0.96	14.37
RGS & PB	1297	1355.4	1.05	7.37
LGS & PB	219	230.4	1.05	1.25
ARGS & $L_\infty B$	1866	1113.6	0.60	6.06
ALGS & $L_\infty B$	436	260.6	0.60	1.42
LGS & $L_\infty B$ with $L_\infty AE1$	230	206.6	0.90	1.12
LGS & $L_\infty B$ with $L_\infty AE2$	230	183.9	0.80	1.00
LGS & PB with PAE1	219	209.5	0.96	1.14
LGS & PB with PAE2	219	188.5	0.86	1.03

Table 4.6: Serial value iteration and GEN2

Algorithm	N_{iter}	T_{soln}	t_{aver}	T_{stan}
PJ & PB	26	32.5	1.25	1.00
PJ & PB with MAE	26	36.7	1.41	1.13
PJ & PB with PAE2	26	34.8	1.34	1.07
J & $L_\infty B$	2861	3596.9	1.26	110.59
RGS & $L_\infty B$	1148	1449.5	1.26	44.57
RGS & PB	1048	1418.5	1.35	43.61
ARGS & $L_\infty B$	1516	1116.0	0.74	34.32
RGS & $L_\infty B$ with $L_\infty AE1$	1148	1648.2	1.44	50.68
RGS & $L_\infty B$ with $L_\infty AE2$	1148	1515.7	1.32	46.60
RGS & PB with PAE1	1048	1604.9	1.53	49.35
RGS & PB with PAE2	1048	1463.9	1.40	45.01

4.3 Analysis

Pre-Jacobi

In all problems except GEN2 and GEN1CS the solution times of the best GS and PJ based algorithms are within a factor of 2 of each other. In GEN2 PJ & PB is 34 times faster than the best GS based algorithm, the difference being due to the number of iterations taken. Two facts explain this. Firstly GEN2 is a rapidly mixing problem whereas all the other are slowly mixing, so PJ behaves relatively well on GEN2. Secondly GEN2 has no WLT ordering of the state space whereas all the other problems do, so GS behaves relatively badly on GEN2. In GEN1CS LGS & $L_\infty B$ with $L_\infty AE2$ is 6 times faster than the best PJ based algorithm. This is because in GEN1CS every policy is slowly mixing and every iteration matrix has a WLT structure, whereas in the other problems the structure of the policies and iteration matrices is variable.

In GEN1CV the immediate rewards make the non-optimal actions in each state good alternatives to the optimal action. As a result of this more iterations are required before PJ converges to the optimal policy for this problem than for GEN1FF. This is shown in figure 4.1. The transition matrices used during these early iterations allow rapid mixing which produces rapid convergence in PJ. Consequently fewer iterations are required to solve GEN1CV by PJ than to solve GEN1FF by PJ.

In all problems except GEN2, PJ requires more iterations than the best GS

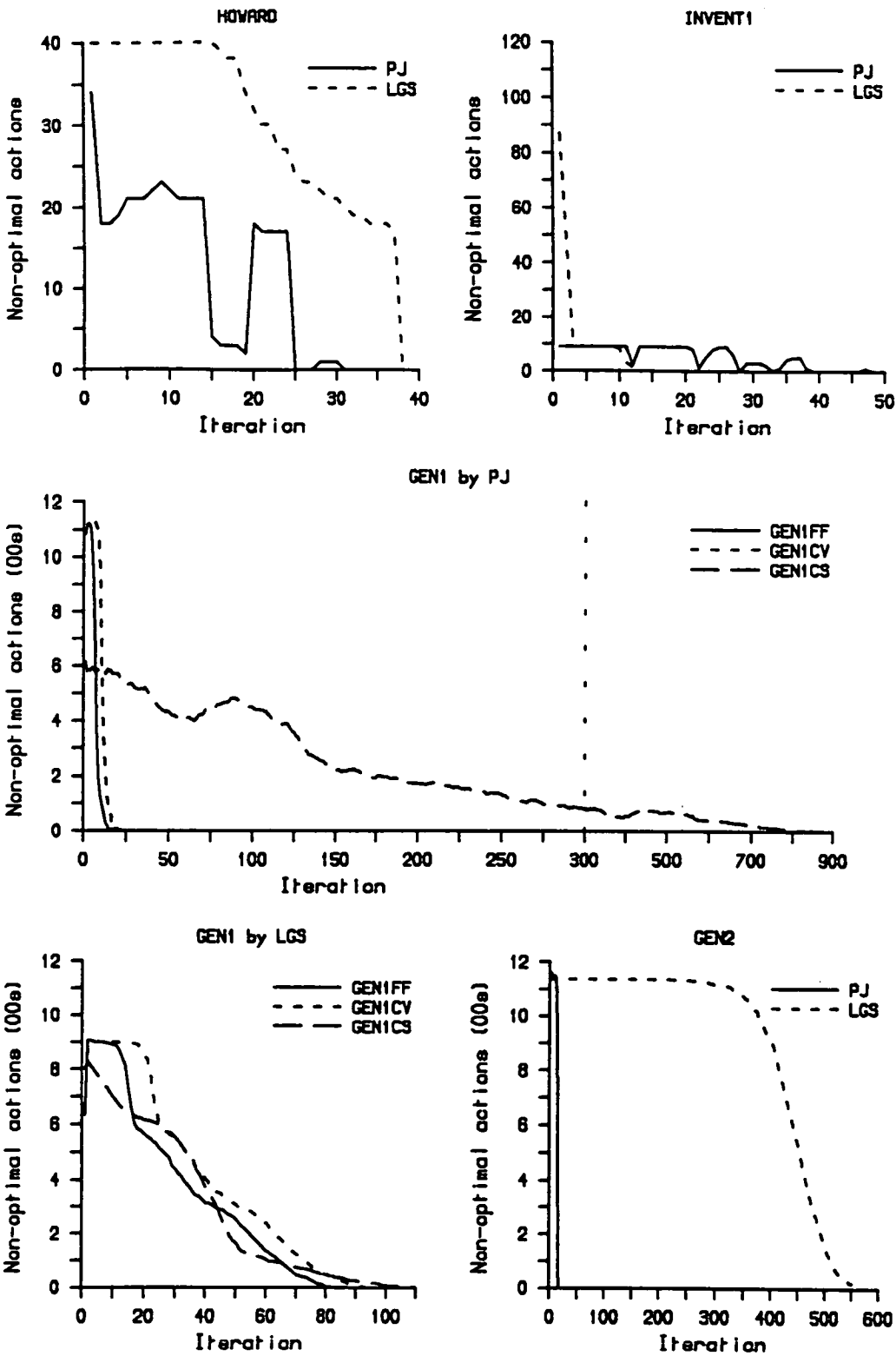


Figure 4.1: Graphs showing the variation in the number of non-optimal actions in the policies selected by PJ & PB and LGS & $L_{\infty}B$.

SERIAL VALUE ITERATION

algorithm (up to 7 times as many in the case of GEN1CS) because of the WLT ordering of the state space and the slow mixing. This is in sharp contrast with the papers by Thomas *et al* [49], where GS requires about 4 times the number of iterations, and Porteus [42], where GS requires between $\frac{2}{3}$ and 3 times the number of iterations.

Jacobi

In all problems Jacobi is by far the worst algorithm to use. The solution times for J & $L_\infty B$ are between 6 and 111 times longer than the best algorithm not employing action elimination. Jacobi cannot exploit the WLT structure or the rapid mixing property of a problem. It will only be successful when the off-diagonal entries in the transition matrices for the policies selected are small compared with the entries on the diagonal. This structure is unlikely to be found in real examples of Markov decision processes and, even if it did arise, could be exploited by a GS algorithm as well.

Gauss-Seidel

In HOWARD and INVENT1 the mainly lower triangular ordering of the state space in LGS & $L_\infty B$ is a factor of about 7 better than the mainly upper triangular ordering in UGS & $L_\infty B$, which is little better than Jacobi, and a factor of about 4 better than the random ordering in RGS & $L_\infty B$. In the three GEN1 problems the difference is even greater with LGS & $L_\infty B$ being between 7 and 12 times faster than UGS & $L_\infty B$ and between 4 and 6 times faster than RGS & $L_\infty B$. The differences in the solution times are entirely due to the differences in the number of iterations taken to converge, which is due to the strong WLT structure in these problems. In such cases the UGS update is very similar to the Jacobi update (both updates would be the same if the transition matrices used were purely upper triangular). The results show the importance of a WLT ordering of the state space because even in problems which are slowly mixing (for example HOWARD, INVENT1, GEN1FF or GEN1CV), the convergence of GS can be much poorer than that of PJ unless a WLT ordering can be found.

The reordering techniques used by Porteus [42] do not achieve improvements as large as those described above because the Markov chains generated in [42] have a much weaker WLT structure than HOWARD, INVENT1 and the three GEN1 problems.

SERIAL VALUE ITERATION

GEN2 has no WLT structure, so reordering the state space will not improve the solution time for GS significantly.

Compared with the $L_\infty B$, the PB reduce the number of iterations required by LGS by between 2% and 8%. In no case does this small reduction result in an improved solution time. However compared with the $L_\infty B$, the PB reduce the number of iterations required by RGS by as much as 13% and improve the solution time for all problems except GEN1CS. This is to be expected for the following reason. In most sparse Markov decision processes, regardless of the ordering of the state space, there will be some action for which all transitions are to states of higher order. It is easy to show that in such cases $\bar{\alpha}$ for GS is equal to β . With a WLT ordering of the state space α^n will be close to 0 if $L^n \geq 0$, but may be close to β if $L^n < 0$, and consequently $\underline{\alpha}$ will be close to 0. Therefore unless U^n and L^n have opposite signs, the PB are little better than the improved form of the $L_\infty B$ (2.8). This situation is unlikely to arise with a random ordering of the state space.

Using the optimal policy at every stage in the solution of GEN1 by GS requires more iterations than solving GEN1FF, GEN1CV or GEN1CS by GS. The non-optimal actions result in faster convergence because when $v_n \neq v^*$ they may give a greater improvement than the optimal actions. It is to be expected that when the non-optimal actions are good alternatives to the optimal actions more progress will be made during the early iterations. Hence convergence of GS is expected to be faster for GEN1CV than for GEN1FF or GEN1CS. This also contributes to the faster convergence of PJ for GEN1CV.

Although in GEN1CS every transition matrix has a WLT structure, LGS & $L_\infty B$ requires more iterations to solve GEN1CS than GEN1FF. This is because convergence to the optimal policy takes longer in GEN1CS than in GEN1FF (see figure 4.1) and consequently, since non-optimal actions are poor alternatives to the optimal actions in GEN1CS, convergence is poorer initially.

Employing the tighter form of the $L_\infty B$ (2.8) has not proved successful. Only INVENT1 (whose immediate rewards are all negative) benefits, with convergence being

SERIAL VALUE ITERATION

detected after 28 iterations instead of 29. In the other problems the immediate rewards do not all have the same sign and the transformation suggested in section 2.2 can result in a very large optimal value which causes loss of accuracy in the solution. However even if loss of accuracy is not a factor, the tighter form of the $L_\infty B$ cannot be better than the PB. Hence the results for the PB show that when a WLT ordering of the state space exists, the improvement in the number of iterations required is slight.

Alternating Gauss-Seidel

As explained in section 2.4, during each iteration of AGS one of the two summations in (2.3) can be saved and reused at the next iteration, but all other parts have to be recalculated. It follows that the time per iteration in AGS can never be less than half the time per iteration in GS, and will be close to a half only for problems where the number of transitions per action is large. The results are consistent with this observation. The ratio of average time per iteration for AGS to that for GS is about 0.55, 0.58, 0.61 and 0.67 for INVENT1, GEN2, the GEN1 problems and HOWARD respectively, where the average numbers of transitions per action are about 12, 8, 6 and 2 respectively.

When a WLT ordering of the state space is used ALGS takes almost twice as many iterations as LGS, indicating that iterations which use the reverse order contribute very little to the convergence of the algorithm. This is to be expected because in the iterations which use the reverse order, it is only the elements above the diagonal of the transition matrices used which change the values and there are relatively few of these when a WLT ordering is used. For WLT orderings the net effect of the increase in number of iterations and decrease in time per iteration is an increase of between 7% and 28% in solution time.

When a random ordering of the state space is used ARGS takes between 1.03 and 1.42 times as many iterations as RGS and in all cases the net effect of the increase in number of iterations and decrease in time per iteration is a reduction in the solution time. However ARGS is still poorer than LGS for those problems which have a WLT ordering. For GEN2, which does not, ARGS & $L_\infty B$ is the best GS algorithm. However in this problem PJ is far superior.

SERIAL VALUE ITERATION

It follows that AGS is uncompetitive in all the test problems. In AGS with action elimination, the proportion of the calculation which can be saved between iterations falls as actions are eliminated. AGS with action elimination will therefore be uncompetitive as well, and so has not been tried. Also in the parallel case, a smaller proportion of the summations can be saved for use at the next iteration. Therefore a parallel algorithm based on AGS is unlikely to be successful (see section 5.2 for further details).

Action Elimination

For all the test problems PAE2 is better than MAE with PJ and L_∞ AE2 is better than L_∞ AE1 with GS, but the differences are often very small. For all the test problems except INVENT1, PAE2 is better than PAE1 with GS. This is to be expected as L_∞ AE2 and PAE2 are based on tighter bounds than L_∞ AE1 and PAE1.

In HOWARD and INVENT1 PAE2 improves PJ by a factor of about 4 whereas L_∞ AE2 and PAE2 improve GS by a factor less than 2.5. The same effect can be observed in the results for GEN1FF and GEN1CS where PAE2 reduces the solution time of PJ by more than L_∞ AE2 and PAE2 reduce the solution time of GS. This is because the PB exploit the stochastic¹ property of the PJ iteration matrices, but in GS the iteration matrices are not stochastic¹ and so the PB are less effective. The L_∞ B do not exploit the structure of the iteration matrices. The reductions in solution times are due to changes in the average time per iteration which result from the elimination of actions and are not due to changes in the number of iterations, which remains the same.

Action elimination is less successful in the generated problems than in HOWARD and INVENT1. This is partially due to the fact that there are fewer actions per state in the generated problems (an average of about 40 for HOWARD, 60 for INVENT1 and only 16 for the generated problems) and so the maximum possible gain from action elimination is less. In GEN1CV and GEN2 it is better not to use action elimination because the way in which the rewards are chosen (see section 3.3.3) means that the elimination of non-optimal actions starts at a later stage than in the other problems and not all non-optimal actions can be eliminated before convergence is detected. This

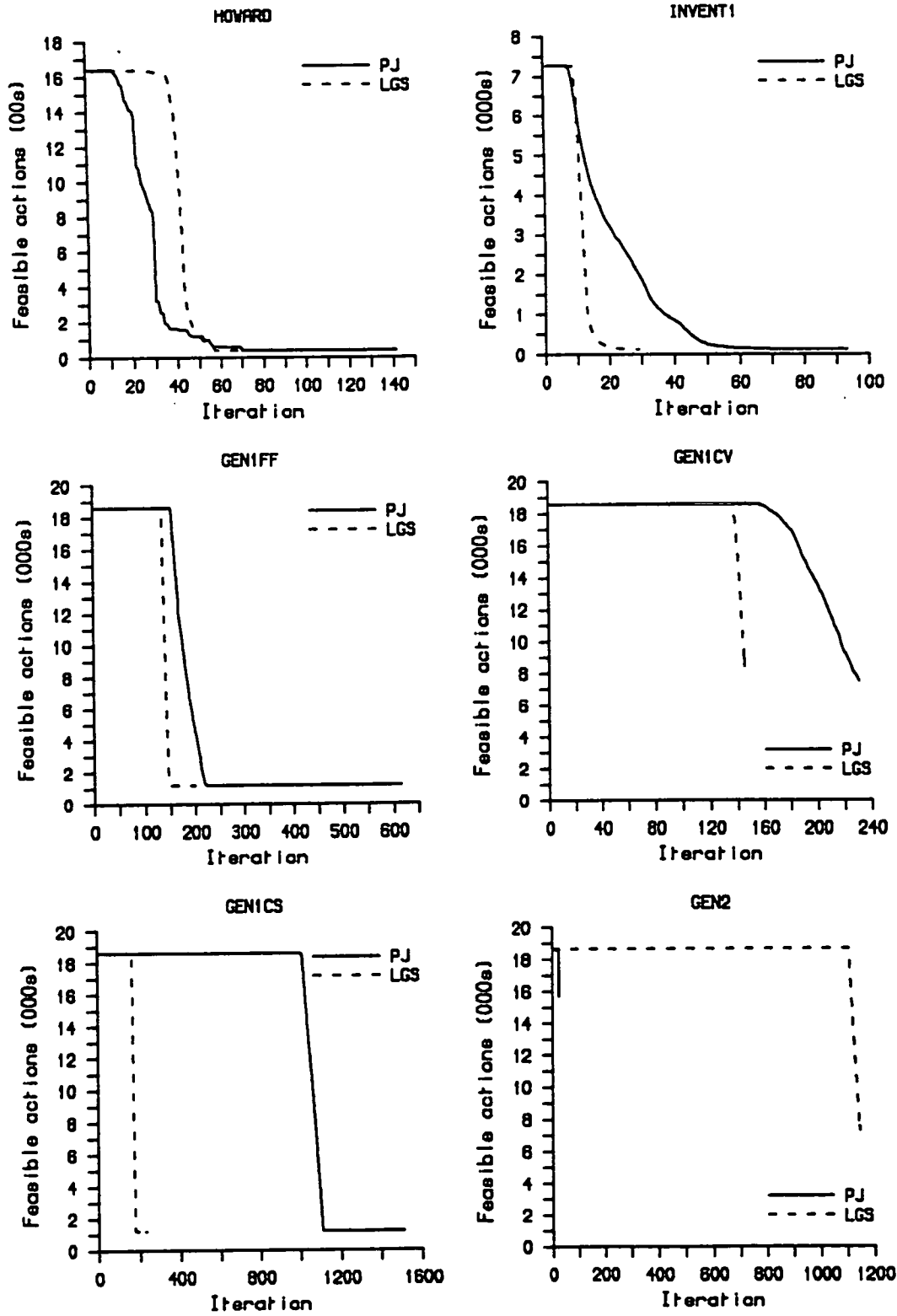


Figure 4.2: Graphs showing the pattern of action elimination in PJ & PB with PAE2 and LGS & $L_{\infty}B$ with $L_{\infty}AE2$.

SERIAL VALUE ITERATION

can be seen in figure 4.2.

When PAE2 is applied to PJ & PB in GEN1FF and GEN1CS all non-optimal actions are eliminated very quickly after the first non-optimal action is eliminated. However in GEN1CS 63% of the iterations are complete before the first non-optimal action is detected, while in GEN1FF only 25% of the iterations are performed before the elimination of actions begins. Consequently action elimination is more effective with GEN1FF than GEN1CS. This is illustrated in figure 4.2.

With PAE2 applied to PJ & PB, an action k in state i is eliminated when

$$H_i^d(v^n) - H_i^k(v^n) > \beta \|\bar{v} - \underline{v}\|_\infty \text{ for some } d \in K_i.$$

In GEN1CS all policies are slowly mixing, while in GEN1FF non-optimal policies are rapidly mixing. Hence while the rate of convergence of the PB is rapid initially in GEN1FF, it is slow throughout in GEN1CS. As a result at a comparable point in the solution procedure $\|\bar{v} - \underline{v}\|_\infty$ is larger in GEN1CS than in GEN1FF. For example after 25% of the iterations are complete $\|\bar{v} - \underline{v}\|_\infty$ equals 0.03519 and 0.00025 in GEN1CS and GEN1FF respectively.

For GS the $L_\infty B$ converge at approximately the same rate in GEN1FF and GEN1CS and so action elimination has a similar effect in both cases. In fact action elimination commences after about 70% of the iterations are complete in both cases and all non-optimal actions are eliminated very quickly thereafter, see figure 4.2.

The results show that action elimination is successful with GS, improving solution time by as much as 55%. $L_\infty AE2$ is the best test to use with GS in the four test problems for which action elimination reduces solution time, but PAE2 proves to be slightly better than $L_\infty AE2$ in the other two cases. In three cases LGS & $L_\infty B$ with $L_\infty AE2$ is the best algorithm to use.

Successive over-relaxation

Trial and error was used to determine a good choice for the relaxation parameter for LSOR & heuristic $L_\infty B$ and RSOR & heuristic $L_\infty B$ for each problem individually and this choice is denoted by ω^* in tables 4.7 and 4.8. (The over-relaxation parameter

Table 4.7: Results of LSOR & heuristic $L_\infty B$

Problem	HOWARD	INVENT1	GEN1FF	GEN1CV	GEN1CS
ω^*	1.05	1.04	1.02	1.02	1.02
N_{iter}	42	14	112	72	128
T_{soln}	1.86	8.51	110	70.9	125
t_{aver}	0.044	0.61	0.98	0.98	0.98

Table 4.8: Results of RSOR & heuristic $L_\infty B$

Problem	HOWARD	INVENT1	GEN1FF	GEN1CV	GEN1CS	GEN2
ω^*	1.04	1.18	1.05	1.05	1.05	1.24
N_{iter}	279	78	797	489	1163	419
T_{soln}	12.3	47.7	781	479	1132	528
t_{aver}	0.044	0.61	0.98	0.98	0.97	1.26

was increased from 1 in steps of 0.01 until the algorithm failed to converge and ω^* was chosen to be the value which resulted in fewest iterations.) Although the results suggest that a larger ω is appropriate when the problem is rapidly mixing or a random ordering of the state space is used, no effective rule for choosing a suitable value of ω for a given problem could be found. This is in contrast with the findings of Thomas *et al* [49], who report that an over-relaxation parameter of 1.28 gave “robust and speedy” convergence.

The investigation has also shown that the range of values of ω for which good convergence is obtained is small. For example with $\omega = 1.03$ in GEN1CV, LSOR & heuristic $L_\infty B$ required 19 times as many iterations as with $\omega = 1.02$. Choosing a value of ω which is lower than the optimum can also have a drastic effect. For example with $\omega = 1.10$ in GEN2, RSOR & heuristic $L_\infty B$ requires about twice as many iteration as with $\omega = 1.24$. So although the solution times in table 4.7 show LSOR & heuristic $L_\infty B$ to be the best algorithm for the three GEN1 problems, if the value of ω had been chosen less carefully the results could have been very different.

As well as the danger of SOR & heuristic $L_\infty B$ failing to converge, even when convergence does occur the resulting approximation to v^* is not guaranteed to satisfy the convergence criterion in (1.3). Table 4.9 gives the number of iterations required

Table 4.9: Number of iterations required by LGS & heuristic $L_\infty B$

Problem	HOWARD	INVENT1	GEN1FF	GEN1CV	GEN1CS
N_{iter}	69	24	157	106	189

by LGS & heuristic $L_\infty B$ (i.e. LSOR & heuristic $L_\infty B$ with $\omega = 1$) and these results demonstrate that with the heuristic form of the $L_\infty B$, LGS terminates after significantly fewer iterations (27% fewer in one case). This suggests that the good performance of SOR is partly due to the application of less demanding bounds and raises questions about the validity of the approximations to v^* found using SOR & heuristic $L_\infty B$. For this reason the solution times shown in tables 4.7 and 4.8 should not be directly compared with those given in section 4.2. However even allowing for a 25% increase in the number of iterations required, LSOR & heuristic $L_\infty B$ is still the best algorithm for the three GEN1 problems when the optimal value of ω is used.

Modifications to the basic algorithms

If the bounds are calculated less frequently, the average time per iteration for all the value iteration algorithms will be less. However since this modification generally leads to a slight increase in the number of iterations performed, any improvement in solution time is typically small. For example in the solution of GEN1FF by PJ & PB, the calculation of the PB requires 3.52s. Calculating the PB after every 5th iteration instead of after every iteration reduces this time by 80%, but two more iterations are required and the net effect is a reduction in solution time of only 0.3%.

Figure 4.2 shows that in problems for which action elimination is successful, all non-optimal actions are eliminated very quickly after the first non-optimal action is detected. It follows that there are many iterations during which the action elimination test detects no non-optimal actions. Consequently reducing the frequency with which the action elimination test is applied often results in a reduction in solution time. The benefits of less frequent bound calculations are more significant when action elimination is successful, because any additional iterations required involve little work. In the solution of GEN1FF by LGS & $L_\infty B$ with $L_\infty AE2$, applying the action elimination test

SERIAL VALUE ITERATION

when all actions are feasible requires 0.07s and reducing the frequency of the bound calculation and the application of the action elimination test to every 10th iteration reduces the solution time by just over 4%.

4.4 Conclusions

This comparison has shown that there is no algorithm which is best for all problems and also shows that the performance of an algorithm on a particular problem can be predicted from the properties of that problem. The two most important properties to use are speed of mixing and the existence of a known WLT ordering of the state space. If the problem is rapidly mixing or no WLT ordering is known, PJ will perform well. If a WLT ordering is known then GS will perform well. Failure to identify the correct algorithm to use for a problem can dramatically increase the solution time required.

Action elimination is a useful addition to both PJ and GS provided non-optimal policies do not have transition structures or vectors of values which are close to those of the optimal policy. Note that for all problems except GEN2, PJ requires more iterations than the best GS algorithm, but takes much less time per iteration whenever there is significant scope for action elimination.

The convergence of all methods is improved when non-optimal policies have vectors of values which are close to those of the optimal policy. The convergence of PJ is improved when the transition structure of non-optimal policies is rapidly mixing.

With sparse problems for which a WLT ordering of the state space is known, GS does not benefit from a stopping rule or an action elimination test based on the PB. This is because the PB, which require a significant amount of calculation each time they are applied, are not much tighter than the $L_\infty B$ in this case.

Modest reductions in solution times can often be achieved by reducing the frequency of the bound calculation and the application of the action elimination test. This fact influences the implementation of the parallel algorithms described in the next chapter (see section 5.3.4), although the modification is never implemented.

This comparison has identified Jacobi, AGS and GS & PB as being of little value,

SERIAL VALUE ITERATION

while the best algorithms appear to be PJ & PB and GS & $L_\infty B$. When action elimination is appropriate, PAE2 is best for PJ and $L_\infty AE2$ is best for GS. The next chapter will concentrate on the development and implementation of parallel versions of these serial algorithms. Although the performance of SOR is very unpredictable, the results show that in certain circumstances SOR can be better than GS and PJ. For this reason consideration will also be given to parallel versions of SOR & heuristic $L_\infty B$.

Footnotes

1. To be precise, the iteration matrices in PJ are not stochastic, but are equal to a scalar multiple of a stochastic matrix, see section 2.2.

Chapter 5

Parallel value iteration

5.1 Introduction

This chapter develops parallel versions of the most successful serial value iteration algorithms from chapter 4. Both synchronous and asynchronous algorithms are considered and the degree of overlapping of calculation and communication is varied. The performance of the parallel algorithms is compared using test problems with distinct properties. A large part of this chapter is devoted to the development of a model for communication between processors. This model is used to guide the choice of connection topology and communication protocols and to optimise the performance of the parallel algorithms.

It has already been noted that the solution of large Markov decision processes is computationally intensive and involves handling a large quantity of data. Therefore to be successful parallel solution algorithms must allow both the computational load and the data storage to be divided between the processors being used. In the value iteration algorithms described in chapter 2, the calculation of v_i^n requires some recently calculated values $v_j^t \forall j \in S$ where $t = n$ or $n - 1$ and some data which is unique to i and independent of n , namely the immediate rewards $r_i^k \forall k \in K_i$ and the transition probabilities $p_{i,j}^k \forall k \in K_i, \forall j \in S$. Therefore an algorithm in which each processor being used calculates a fixed subset of the components of v^n allows the data to be distributed as well as the calculation. With this approach processors are required to communicate with each other so that the values v_j^t are available to a processor when its calculation requires them. This method of exploiting the parallelism inherent in the

PARALLEL VALUE ITERATION

solution of Markov decision processes will be employed in all the parallel algorithms developed in this study.

Let P denote the number of processors used and let $\{S_i\}_{i=1}^P$ denote a partition of the state space consisting of P sets, where $S_i = \{n_i, \dots, n_{i+1} - 1\}$, $n_1 = 1$ and $n_{P+1} = m + 1$.

5.2 Synchronous non-overlapped parallel value iteration

This section describes parallel value iteration algorithms in which the calculation and communication tasks are not overlapped and the processors are synchronised so that the convergence properties of the corresponding serial algorithm are preserved (this is achieved by ensuring that v^{n-1} is available to all processors at the start of the n^{th} iteration). This technique is similar to that used by Bertsekas and Tsitsiklis [12] and Freeman [21] to distribute other iterative algorithms.

Synchronous non-overlapped parallel PJ & PB

Let processor r perform the following steps sequentially.

Step 1 Initialisation

Let $v_i^0 = 0 \forall i \in S$ and let $n = 1$

Step 2 Calculation

Calculate $v_i^n \forall i \in S_r$, $U_r^n = \max_{i \in S_r} \{v_i^n - v_i^{n-1}\}$ and $L_r^n = \min_{i \in S_r} \{v_i^n - v_i^{n-1}\}$

Step 3 Communication

Send $v_i^n \forall i \in S_r$ to all other processors and receive $v_i^n \forall i \notin S_r$ (in a procedure known as a multinode broadcast)

Perform a global maximisation and minimisation to calculate U^n and L^n from the values $U_1^n, \dots, U_P^n$ and $L_1^n, \dots, L_P^n$ respectively

Step 4 Check for convergence

If $\frac{\beta}{1-\beta}(U^n - L^n) > \epsilon$, increase n by 1 and repeat from step 2

PARALLEL VALUE ITERATION

In GS states must be updated one after another in a fixed order, with the result of one state update being used immediately by the following state update, see (2.3). Therefore unlike in PJ, it is not generally possible to perform state updates in parallel. However the spirit of GS is to use the “most recently updated values” and modifying the definition of this leads to versions of generalised GS which are suitable for parallel implementation. Defining the “most recently updated values” corresponds to defining the sets T_i in generalised GS, see (2.4). T_i must be a subset of the set of states which have been updated before the updating of state i begins. Hence since the number of states in this set decreases as the number of processors which can be used in a parallel implementation of generalised GS increases, the higher the degree of parallelism in generalised GS, the more the convergence of generalised GS is like Jacobi. Synchronous non-overlapped parallel versions of two generalised GS variants are considered in this section, namely GS1 and GS2. In GS1 a processor uses the most recent data for states local to that processor and data from the previous iteration for all other states, so $T_i = \{j \in S : j < i \text{ and } i, j \in S_r\}$. In GS2 a processor uses the most recent data for all states, so $T_i = \{j \in S : j - n_i < i - n_r, j \in S_i \text{ and } i \in S_r\}$.

Synchronous non-overlapped parallel GS1 & $L_\infty B$

Let processor r perform the following steps sequentially.

Step 1 Initialisation

Let $v_i^0 = 0 \forall i \in S$ and let $n = 1$

Step 2 Calculation

Calculate $v_i^n \forall i \in S_r$ and $N_r^n = \max_{i \in S_r} |v_i^n - v_i^{n-1}|$

Step 3 Communication

Send $v_i^n \forall i \in S_r$ to all other processors and receive $v_i^n \forall i \notin S_r$

Perform a global maximisation to calculate $\|v^n - v^{n-1}\|_\infty$ from the values $N_1^n, \dots, N_P^n$

Step 4 Check for convergence

If $\frac{\beta}{1 - \beta} \|v^n - v^{n-1}\|_\infty > \frac{\epsilon}{2}$, increase n by 1 and repeat from step 2

PARALLEL VALUE ITERATION

This parallel algorithm is essentially the same as synchronous non-overlapped parallel PJ, the only differences being the update rules and bounds applied.

Synchronous non-overlapped parallel GS2 & $L_\infty B$

Let processor r perform the following steps sequentially.

Step 1 Initialisation

Let $v_i^0 = 0 \forall i \in S$ and let $n = 1$

Step 2 Initialisation for iteration n

Let $j(t) = n_t$ for $t = 1, 2, \dots, P$

Step 3 Update the next state

If $j(r) < n_{r+1}$ calculate $v_{j(r)}^n$

Step 4 Exchange latest updates

Send $v_{j(r)}^n$ to all other processors provided $j(r) < n_{r+1}$ and receive $v_{j(t)}^n \forall t \neq r$ satisfying $j(t) < n_{t+1}$

Step 5 Check for further updates

Increase $j(t)$ by 1 for $t = 1, \dots, P$ and if $\exists t$ such that $j(t) < n_{t+1}$ repeat from step 3

Step 6 Initiate a bound calculation

Calculate $N_r^n = \max_{i \in S_r} |v_i^n - v_i^{n-1}|$

Step 7 Complete the bound calculation

Perform a global maximisation to calculate $\|v^n - v^{n-1}\|_\infty$ from the values $N_1^n, \dots, N_P^n$

Step 8 Check for convergence

If $\frac{\beta}{1-\beta} \|v^n - v^{n-1}\|_\infty > \frac{\epsilon}{2}$, increase n by 1 and repeat from step 2

Although the structure of this parallel algorithm is very different from that of the parallel versions of PJ and GS1, the communication procedures required are the same, namely a multinode broadcast and a global maximisation.

In all three algorithms processor r uses the following data during the n^{th} iteration.

$$\left\{ \{r_i^k : \forall k \in K_i, \forall i \in S_r\}, \{p_{i,j}^k : \forall j \in S, \forall k \in K_i, \forall i \in S_r\}, v^{n-1} \right\} \quad (5.1)$$

However in GS1 and GS2 some of the components of v^{n-1} become redundant as states are updated and can therefore be overwritten. The only data which is duplicated on all processors is v^{n-1} . In most cases this duplication reduces communication time and therefore is efficient. This point will be discussed further in section 5.5. The state updates performed in all three parallel algorithms can incorporate any of the action elimination tests for problems which benefit from these techniques.

Let alternating generalised GS be an instance of variable generalised GS for which T_i^n equals T_i^f if n is even and T_i^b if n is odd and the state space can be reordered so that $T_i^f \subset \{1, \dots, i-1\}$ and $T_i^b \subset \{i+1, \dots, m\}$. Since $T_i^f \cap T_i^b = \emptyset$, the value of $\sum_{j \in T_i^f} p_{i,j}^k v_j$ during an iteration which uses T_i^f is the same as it was at the most recently completed iteration and need not be recalculated. Similarly during an iteration which uses T_i^b , $\sum_{j \in T_i^b} p_{i,j}^k v_j$ need not be recalculated. (Note that AGS is a form of alternating generalised GS in which $T_i^f = \{1, \dots, i-1\}$ and $T_i^b = \{i+1, \dots, m\}$.) Hence the average time per iteration is less in alternating generalised GS than in generalised GS. However since the number of states in T_i^f and T_i^b falls as the scope for parallelism increases, the relative advantage of alternating generalised GS over generalised GS falls as the number of processors which can be used increases. It follows therefore that alternating generalised GS cannot be more successful than AGS which is only effective when a random ordering of the state space is used. For this reason alternating generalised GS has not been implemented.

In SOR, as in GS, the states must be updated one after the other in a fixed order. However parallel generalised SOR algorithms can be developed. In this study a parallel generalised SOR algorithm (SOR1) analogous to GS1 will be used to determine the merits of parallel SOR. For completeness the steps of this algorithm are described in detail below.

Synchronous non-overlapped parallel SOR1 & heuristic $L_\infty B$

Let processor r perform the following steps sequentially.

PARALLEL VALUE ITERATION

Step 1 Initialisation

Let $v_i^0 = 0 \forall i \in S$ and let $n = 1$

Step 2 Calculation

Calculate $v_i^n = \omega(Gv^{n-1})_i + (1-\omega)v_i^{n-1} \forall i \in S_r$ and $N_r^n = \max_{i \in S_r} |v_i^n - v_i^{n-1}|$ where G is as in (2.4) with $T_i = \{j \in S : j < i \text{ and } i, j \in S_r\}$

Step 3 Communication

Send $v_i^n \forall i \in S_r$ to all other processors and receive $v_i^n \forall i \notin S_r$

Perform a global maximisation to calculate $\|v^n - v^{n-1}\|_\infty$ from the values $N_1^n, \dots, N_P^n$

Step 4 Check for convergence

If $\frac{\alpha}{1-\alpha} \|v^n - v^{n-1}\|_\infty > \frac{\epsilon}{2}$ where $\alpha = \frac{\|v^n - v^{n-1}\|_\infty}{\|v^{n-1} - v^{n-2}\|_\infty}$, increase n by 1 and repeat from step 2

This algorithm is not guaranteed to converge and action elimination is not appropriate because no bounds relating v^n to v^* are known. As in serial SOR a heuristic stopping rule is used.

5.3 Efficient communication procedures

Introduction

The parallel algorithms described in section 5.2 require procedures to perform a multinode broadcast and a global maximisation and minimisation. Hence an efficient implementation of these algorithms depends upon efficient procedures to perform these communication tasks, which in turn depends upon the way processors are connected. The choice of connection topology is therefore a key issue in the implementation of these algorithms. This section builds a model for communication along transputer links and uses this to compare efficient communication procedures on different connection topologies.

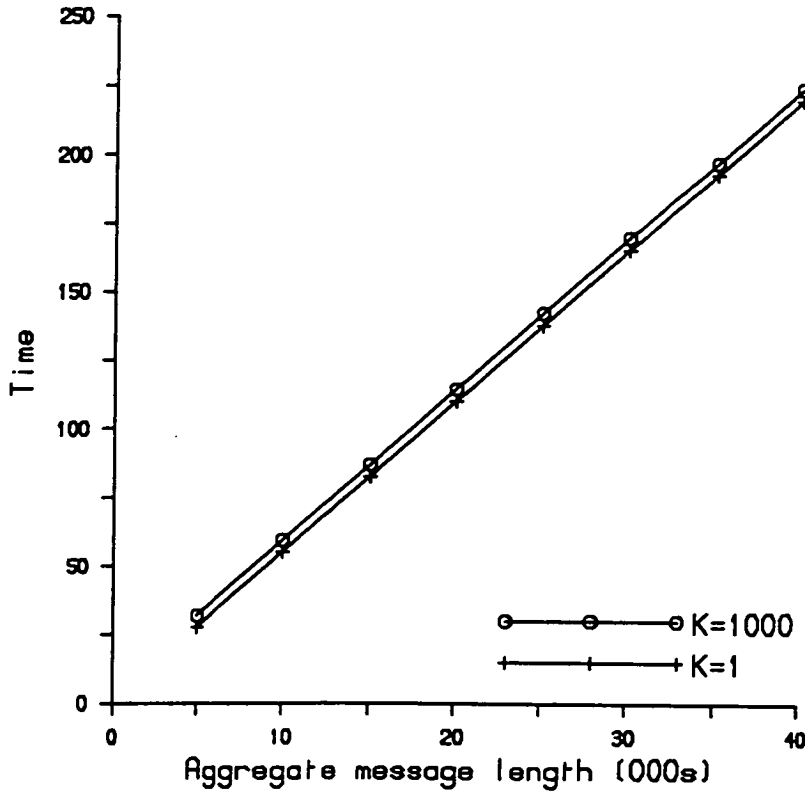


Figure 5.1: A plot of the time required (in ms) to send K separate messages each consisting of ℓ values along a link against aggregate message length $= K\ell$.

5.3.1 Modelling communication between transputers

A transputer has four links and each link consists of two channels which make communication in both directions along the link possible. In a simple experiment using two transputers connected by one link, the time required to send K separate messages each consisting of ℓ values (a value is a 64bit real) along a link was recorded with $K = 1$ and 1000 and $K\ell$ ranging from 5000 to 40000. The results of this experiment, shown in figure 5.1, suggest that communication is comprised of a set-up time plus a time proportional to the length of the message sent or received, which will be referred to as the transmission time. Therefore a suitable model for the time required to send a message consisting of ℓ values along one channel is $c + \ell\rho$, where c is the set-up time for one channel and ρ is the transmission time for a 64bit value along a link.

In another experiment the time required to send a message consisting of ℓ values

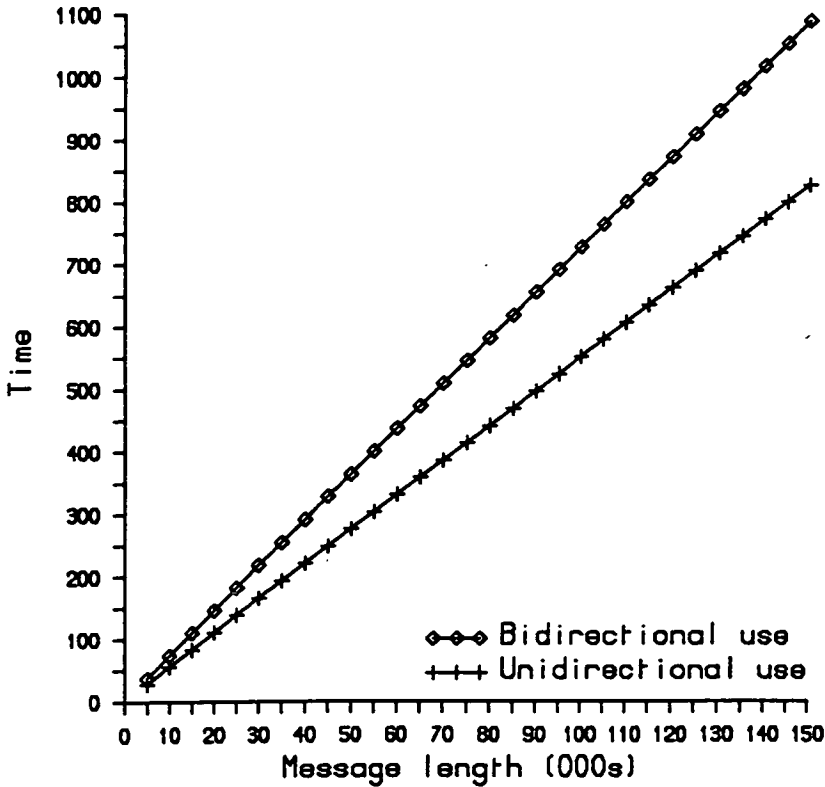


Figure 5.2: Plots of the time required (in ms) to send a message along a link which is being used unidirectionally in one case and bidirectionally in the other against message length.

along a link in both directions simultaneously was recorded with ℓ ranging from 5000 to 150000. The results of this experiment, shown in figure 5.2, show that the apparent rate of a link is less when both channels of a link are used simultaneously. This is due to the acknowledgments which are sent during transmission. Least squares estimates of the gradients of the curves in figure 5.2 yield estimates for ρ of $5.51\mu s$ when a link is used unidirectionally and $7.27\mu s$ when a link is used bidirectionally. Therefore when a link is used bidirectionally the transmission time is increased by around 32%, but twice as much data is passed. A 52% increase in the data transmitted per unit time can be achieved by using a link bidirectionally, so bidirectional use is worthwhile whenever data needs to be transmitted in both directions along a link. In the timing models developed in later sections $\frac{3}{4}$ of the bidirectional transmission time will be used as an approximation to the unidirectional transmission time.

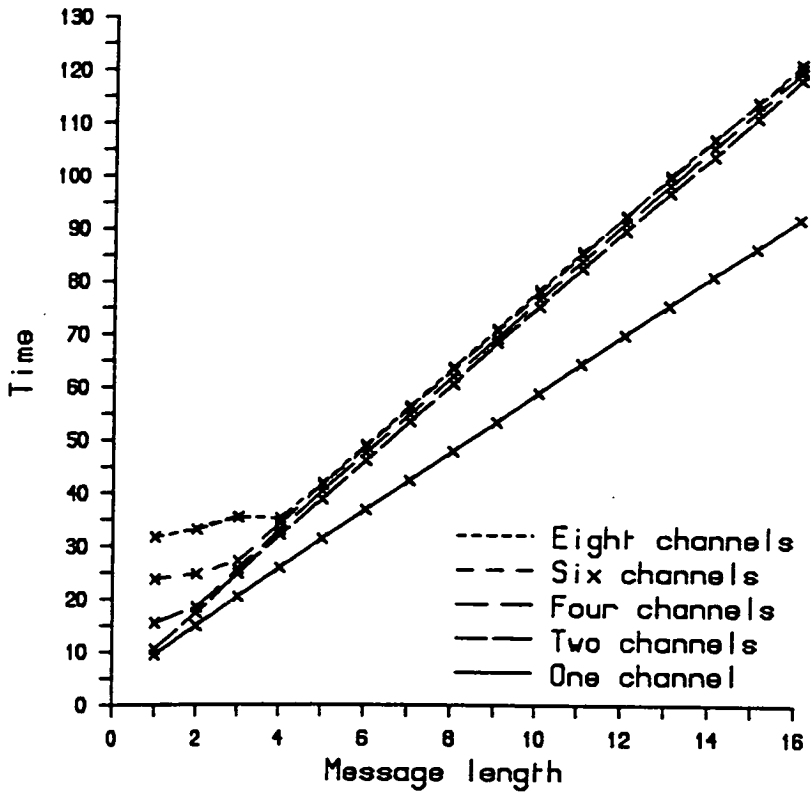
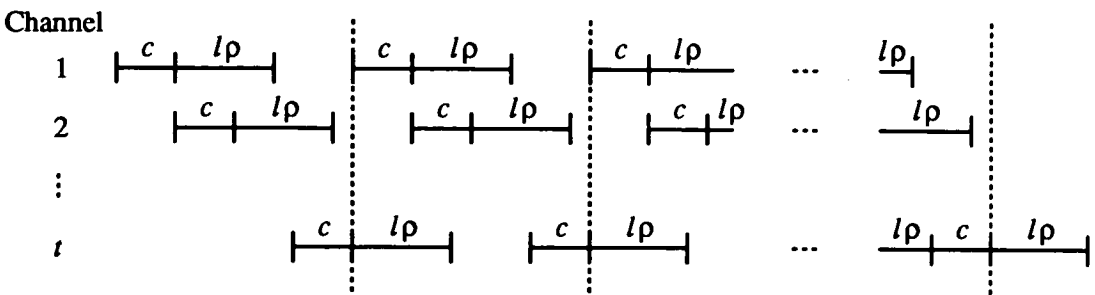


Figure 5.3: Plots of the time required (in ms) for concurrent communication of 1000 separate messages of equal length along t channels on the same transputer against message length.

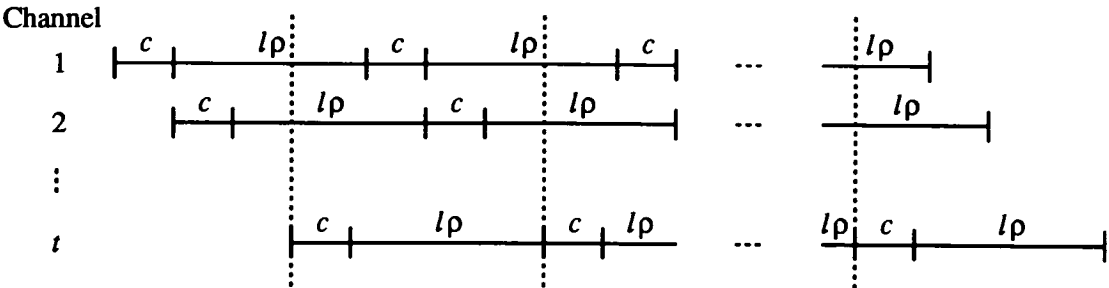
In the next experiment t channels on the same transputer were used concurrently, each communicating K separate messages each consisting of ℓ values. Figure 5.3 shows measured times for this task with one channel, one link used bidirectionally (two channels), two links used bidirectionally (four channels), three links used bidirectionally (six channels) and four links used bidirectionally (eight channels); $K = 1000$ and ℓ ranging from 1 to 16. The length of the initial kink increases as the number of channels used increases which indicates that channel set-ups occur sequentially. However for longer message lengths there is little difference between the plots for more than one channel, indicating that the effect of sequential set-ups is lost when long messages are transmitted repeatedly. (Note that due to bidirectional use of links, the gradient of the plots for more than one channel is greater than the gradient of the plot for one channel.) These observations are consistent with the following model.

PARALLEL VALUE ITERATION

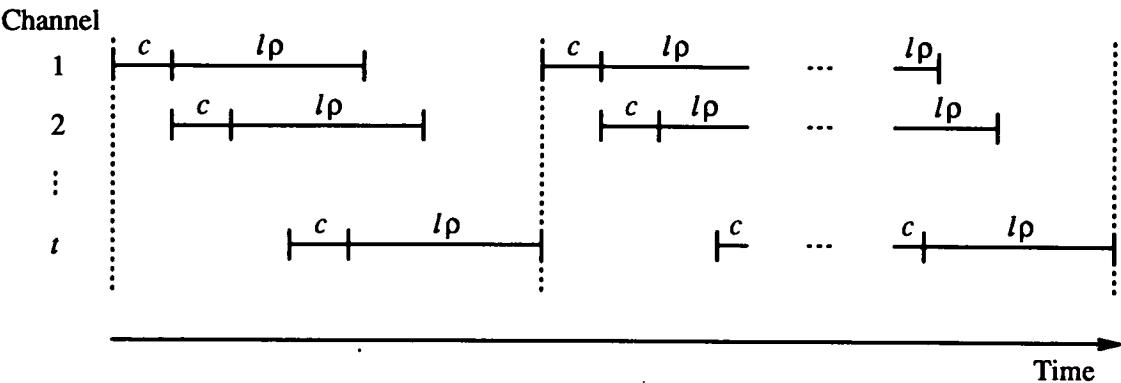
(a) Short message length, channels operating independently



(b) Long message length, channels operating independently



(c) Synchronised channels



Time

Figure 5.4: Scheduling multiple channel communication assuming there is no delay between a channel set-up and the start of transmission.

Assume that different channels on the same transputer cannot be set up in parallel, but a channel set-up and the transmission along several channels can take place concurrently. Further suppose that a channel can always start transmission as soon as it has been set up. If the transmission time for l values is less than the set-up time for $t - 1$ channels then, because the first channel to be set up is idle by the time the

PARALLEL VALUE ITERATION

other $t - 1$ channels have been set up, the total time for the task above is $Ktc + \ell\rho$. Thus for large K the average time per message sent is tc . This situation is illustrated in figure 5.4(a). For larger ℓ , the first channel to be set up is still in use after the other $t - 1$ channels have been set up, and so can be set up without delay when the transmission along it is complete. This situation is illustrated in figure 5.4(b). In this case the total time is $(t - 1)c + K(c + \ell\rho)$ and for large K the average time per message sent is $c + \ell\rho$. Applying this model to the plot for one channel in figure 5.3 gives rise to an estimate of $4\mu s$ for c , but applying the model to the portion of the plot for eight channels corresponding to $\ell \geq 4$ gives an estimate of $6\mu s$ for c . The above analysis assumes that channel set-ups on neighbouring transputers occur in a sequence which eliminates delays. However other sequences of channel set-ups can introduce a delay of up to $7c$ between the end of a channel set-up and the start of transmission along that channel. Due to the asynchronous nature of transputer communications the optimal sequence cannot be enforced and the resulting delays are included in the estimates of channel set-up time.

The third experiment was repeated with the t channels synchronised so that communication of the n^{th} message along a channel could not commence until the communication of the $(n - 1)^{\text{th}}$ message along every channel was complete. Figure 5.5 shows measured times for this task with the same scenarios as before. These results show the effect of sequential channel set-ups when channels are synchronised in this manner. Figure 5.4(c) illustrates this situation assuming that a channel can always start transmission as soon as it has been set up, although it should be noted that the sequence of channel set-ups on neighbouring transputers will not affect the total time in this case. The total time is $K(tc + \ell\rho)$ and so the average time per message sent is $tc + \ell\rho$.

5.3.2 Procedures for multinode broadcasts

When a processor performs a broadcast it sends the same message to all other processors. In a multinode broadcast each processor performs a broadcast concurrently. This communication task has been identified as being important in the parallel value

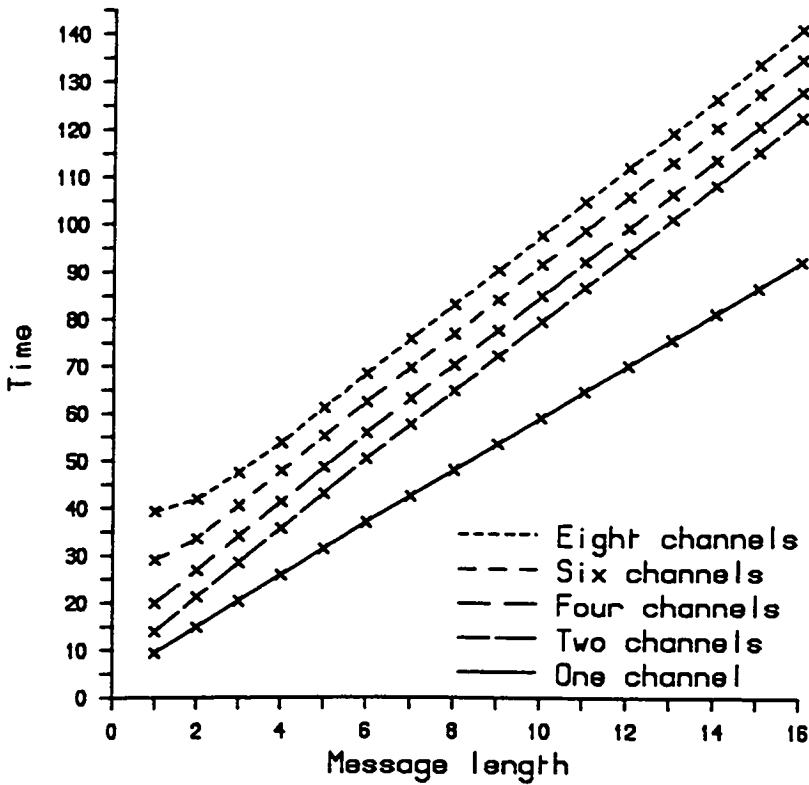


Figure 5.5: Plots of the time required (in ms) for concurrent communication of 1000 separate messages of equal length along t synchronised channels on the same transputer against message length.

iteration algorithms described in section 5.2. In this section multinode broadcast procedures for three different connection topologies are developed and analysed. The procedures are suitable for any distributed memory parallel computer, but the timing models are specific to transputer based parallel computers.

A lower bound on the transmission time required during a multinode broadcast on any network of P transputers when messages consist of ℓ values is given by $\frac{P-1}{4}\ell\rho$, where ρ is the transmission time for a 64bit value along a link which is being used bidirectionally. This bound, which will be referred to as the rate bound, can be derived as follows. During a multinode broadcast each transputer must receive at least $P - 1$ messages, each containing data from one of the other $P - 1$ transputers. Hence the total number of messages received is at least $P(P - 1)$. Since every message transmitted involves both a send and a receive, the total number of messages sent is at least $P(P - 1)$.

PARALLEL VALUE ITERATION

Hence some transputer must send at least $P - 1$ messages and, using the result of section 5.3.1, the quickest method of receiving and sending $P - 1$ messages is to use all four links bidirectionally.

Let θ_r denote the message to be broadcast by processor r during the multinode broadcast and assume this message can be split into two halves, denoted by θ_r^1 and θ_r^2 , if required. In the following analysis it is assumed that all processors start the multinode broadcast procedure at the same time, all messages consist of ℓ values and the transmission time for a half message along a link which is being used bidirectionally is longer than six channel set-ups. The last assumption ensures that the situation illustrated in figure 5.4(b) always applies.

Chain topology

In the chain topology all processors are connected to their two neighbours by two links, except for the two outside processors which can also communicate with the external environment, see figure 5.6. For reference the P processors are numbered from 1 to P as shown in figure 5.6.

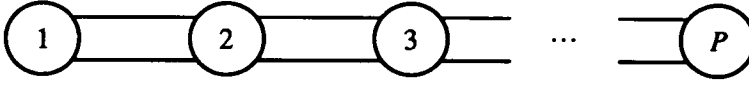


Figure 5.6: The chain topology.

BROADCAST: an algorithm for a multinode broadcast on a chain

Let processor r perform the following four tasks concurrently.

Task 1

- a Let $n = 1$
- b While $n < r$
 - read θ_{r-n}^1 from processor $r - 1$ and send θ_{r-n+1}^1 to processor $r + 1$ (provided $r \neq P$) concurrently and then increase n by 1
- c If $r \neq P$ send θ_1^1 to processor $r + 1$

Task 2

- a Let $n = 1$
- b While $n \leq P - r$
 - read θ_{r+n}^1 from processor $r + 1$ and send θ_{r+n-1}^1 to processor $r - 1$ (provided $r \neq 1$) concurrently and then increase n by 1
- c If $r \neq 1$ send θ_P^1 to processor $r - 1$

PARALLEL VALUE ITERATION

Task 3

- a Let $n = 1$
- b While $n < r$
 - read θ_{r-n}^2 from processor $r - 1$ and send θ_{r-n+1}^2 to processor $r + 1$ (provided $r \neq P$) concurrently and then increase n by 1
- c If $r \neq P$ send θ_1^2 to processor $r + 1$

Task 4

- a Let $n = 1$
- b While $n \leq P - r$
 - read θ_{r+n}^2 from processor $r + 1$ and send θ_{r+n-1}^2 to processor $r - 1$ (provided $r \neq 1$) concurrently and then increase n by 1
- c If $r \neq 1$ send θ_P^2 to processor $r - 1$

The following results show that BROADCAST performs a multinode broadcast on a chain and construct a timing model for the procedure on a transputer based parallel computer.

Proposition 5.1

BROADCAST will not deadlock¹.

Proof

The algorithm will not deadlock if the communications attempted at each step of all four tasks are completed successfully.

Consider task 1: when $n = t$, where $1 \leq t \leq P - 1$, processor r

reads θ_{r-t}^1 from processor $r - 1$ and sends θ_{r-t+1}^1 to processor $r + 1$ concurrently if $t < r$ and $r \neq 1$ or P ,

reads θ_{r-t}^1 from processor $r - 1$ if $t < r$ and $r = P$,

sends θ_1^1 to processor $r + 1$ if $t = r$ and $r \neq P$

or is idle if $t > r$,

so communication from t to $t + 1$, from $t + 1$ to $t + 2$, ..., from $P - 1$ to P is successful and no other communication is attempted. Hence each step of task 1 is completed successfully.

A similar argument can be used for the other three tasks.

PARALLEL VALUE ITERATION

Proposition 5.2

The messages sent by processor r during BROADCAST are known to it before they need to be sent.

Proof

Consider each task separately.

In task 1 if $n = 1$ processor r sends θ_r^1 , which it knows initially, while if $n = t$ (where $1 < t < P$) and $t \leq r$ processor r sends θ_{r-t+1}^1 , which is the message it read from processor $r - 1$ during the most recently completed step of task 1.

A similar argument can be used for the other three tasks.

Proposition 5.3

When BROADCAST terminates every processor knows all P messages.

Proof

Processor r knows θ_r initially, receives $\{\theta_{r-1}^1, \dots, \theta_1^1\}$ and $\{\theta_{r-1}^2, \dots, \theta_1^2\}$ from processor $r - 1$ (provided $r \neq 1$) and receives $\{\theta_{r+1}^1, \dots, \theta_P^1\}$ and $\{\theta_{r+1}^2, \dots, \theta_P^2\}$ from processor $r + 1$ (provided $r \neq P$).

Proposition 5.4

If BROADCAST is implemented on a transputer based computer and tasks are initiated in the order in which they are presented in the statement of BROADCAST, then the processor at the beginning of the chain will receive its final message in the multinode broadcast

$$\begin{cases} 4c + \frac{3\ell}{8}\rho & \text{if } P = 2 \\ (2P + 3)c + \frac{7(P - 1)\ell}{16}\rho & \text{if } P \text{ is odd} \\ 2(P + 1)c + \frac{(7P - 6)\ell}{16}\rho & \text{if } P \text{ is even and } P > 2 \end{cases} \quad (5.2)$$

after the start of the procedure, where c is the set-up time for a channel and ρ is the transmission time for a 64bit value along a link which is being used bidirectionally, and by this time all other processors will have received their final messages.

PARALLEL VALUE ITERATION

Proof

Case 1: $P = 2$

The two processors communicate four half messages using two links bidirectionally and no synchronisation is required. Hence the time required is given by $4c + \frac{3\ell}{8}\rho$.

Case 2: $P > 2$

The messages relayed from processor 1 to processor P and from processor P to processor 1 limit the time required for the multinode broadcast. If communication tasks are initiated in the order in which they are presented in the statement of BROADCAST, the message relayed from processor 1 to processor P will reach processor P before the message relayed from processor P to processor 1 reaches processor 1.

After $\left\lfloor \frac{P}{2} \right\rfloor$ steps of each task have been completed, the message relayed from processor 1 to processor P has passed the message relayed from processor P to processor 1 near the middle of the chain, and from this point on no link is ever used bidirectionally.

Note that on all processors except 1 and P , part b of each task requires the synchronised use of a pair of channels. Assuming tasks are initiated in the order in which they are presented in the statement of the algorithm, there will be a delay of $6c$ on processors $2, 3, \dots, P-1$ before task 4 is initiated. Tracking the progress of θ_P^2 along the chain gives the time required by task 4. The first $\left\lfloor \frac{P}{2} \right\rfloor - 1$ communications in which θ_P^2 is involved each require $2c + \frac{\ell}{2}\rho$ regardless of whether P is odd or even, but there are two distinct cases for the next communication.

Case A: P odd

Processor $\frac{P+1}{2}$ reads θ_P^2 from a link which is used bidirectionally and sends θ_{P-1}^2 along a link which is being used bidirectionally concurrently. The time required for this task is $2c + \frac{\ell}{2}\rho$.

Case B: P even

Processor $\frac{P}{2}$ reads θ_P^2 from a link which is used bidirectionally and sends θ_{P-1}^2 along a link which is being used unidirectionally concurrently. Hence, since processor $\frac{P}{2}$ initiates the read first and $\frac{\ell}{2}\rho - \frac{3\ell}{8}\rho > c$, the send can be performed before the read

PARALLEL VALUE ITERATION

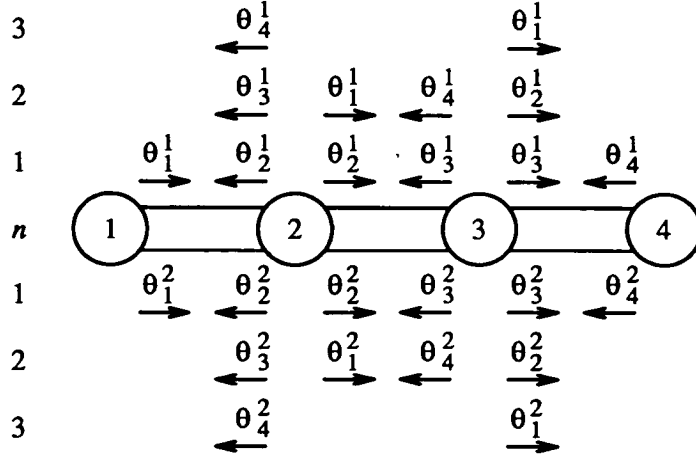


Figure 5.7: Channel usage when BROADCAST is applied to a 4 processor chain.

is complete and the time required is $c + \frac{\ell}{2}\rho$.

In both cases the next $\left\lfloor \frac{P-1}{2} \right\rfloor - 1$ communications in which θ_P^2 is involved each require $2c + \frac{3\ell}{8}\rho$ and the last requires $c + \frac{3\ell}{8}\rho$. Combining these terms gives the required expressions.

Figure 5.7 shows the communications which take place during each step of BROADCAST and figure 5.8 shows how the communication tasks in BROADCAST might be scheduled on each transputer in the chain assuming that communication tasks are initiated in the order in which they are presented in the statement of BROADCAST.

The time for which the input channels connected to transputers 1 and P are idle is due to efficient bidirectional use of links and synchronisation of pairs of channels and so cannot be reduced. Therefore BROADCAST is optimal for a chain topology when messages are long. However the time taken by BROADCAST tends to $\frac{7P-6}{16}\ell\rho$ if P is even and $\frac{7(P-1)}{16}\ell\rho$ if P is odd (approximately 75% longer than the rate bound) as the number of values in each message tends to infinity. This is not surprising because for each task in BROADCAST the average number of channels used per step is only $\frac{P}{2}$.

Since each processor has at most two neighbours, it is not possible to combine messages into longer packets without delaying one message while another is read in. Delaying messages in this way is not profitable when message lengths are long, but

PARALLEL VALUE ITERATION

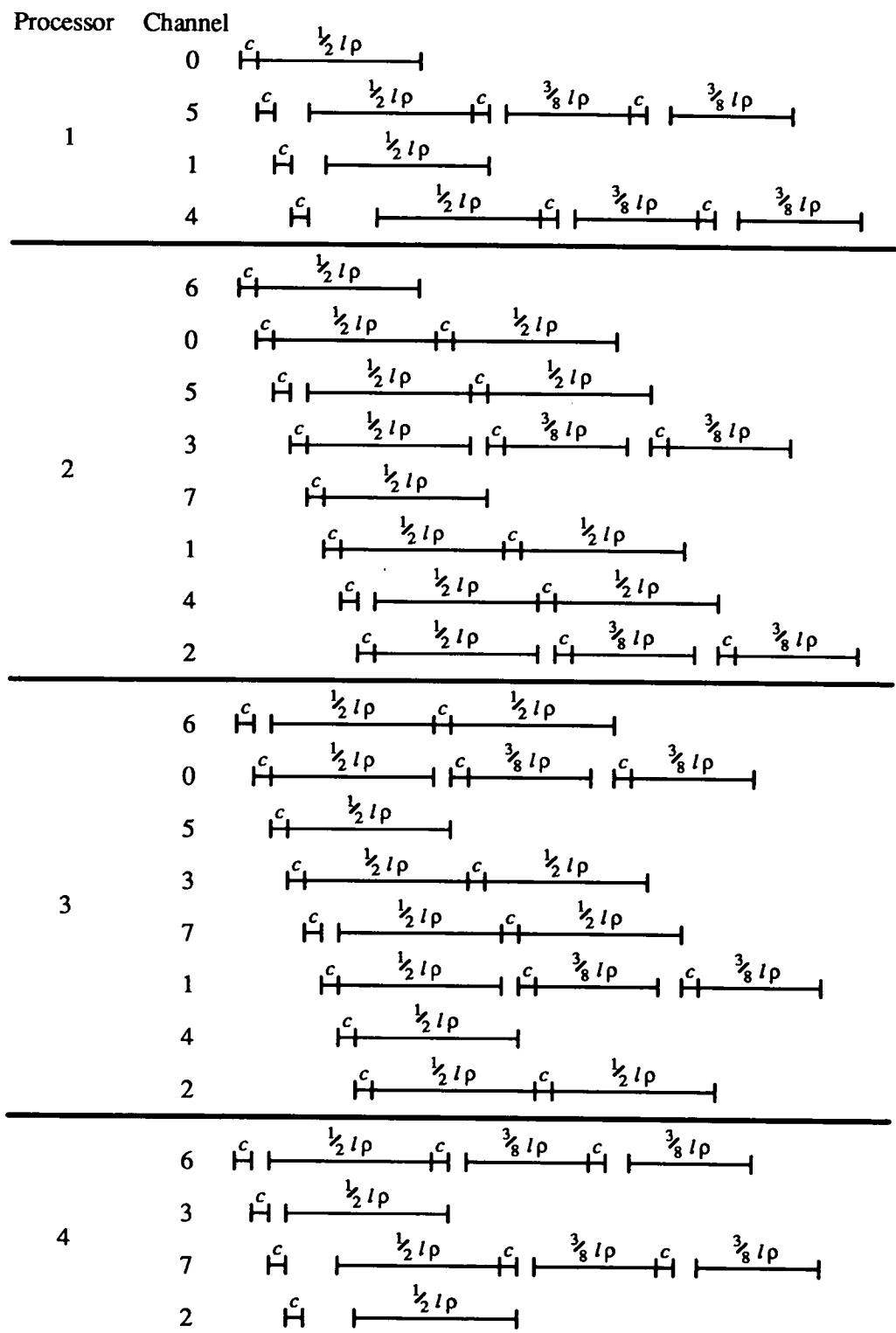


Figure 5.8: Scheduling the communication tasks in BROADCAST for the case of a 4 transputer chain.

Table 5.1: Least squares estimates of c and ρ for the three topologies

P	Chain		Ring				Torus	
	Optimal Wiring		Automatic Wiring		Optimal Wiring		Optimal Wiring	
	c_1	ρ_1	c_2	ρ_2	c_3	ρ_3	c_4	ρ_4
9	11.63	7.25	21.00	7.73	21.93	7.32	12.66	7.63
15	13.25	7.34	19.33	7.79	18.05	7.66	—	—
25	13.22	7.36	20.05	9.45	19.58	7.44	12.81	7.72
35	13.21	7.35	12.63	9.10	17.96	7.56	—	—
49	13.74	7.28	18.11	10.77	17.33	7.54	15.16	7.90
121	9.77	7.29	11.12	11.38	19.44	7.32	12.22	8.50

In all cases the error in the least squares fit is less than 1%.

when messages are very short (and consequently channel set-up time dominates) this approach can be more efficient. Since splitting messages increases the total channel set-up time, it is not beneficial when message lengths are very short.

Least squares estimates of c and ρ were obtained for the chain by measuring the time required for a multinode broadcast using BROADCAST with different numbers of transputers and different values of ℓ and applying (5.2). These estimates are denoted by c_1 and ρ_1 in table 5.1. The estimates of transmission time during a multinode broadcast on a chain are consistent with estimates of transmission time during less complicated tasks. However the estimates of channel set-up time during a multinode broadcast on a chain are approximately twice the estimate of channel set-up time during less complicated tasks. As before this is because the model on which these estimates are based does not allow for any delays in the scheduling of communication tasks.

Ring topology

In the ring topology each processor has two neighbours and is connected to both by two links, see figure 5.9. Dynamic reconfiguration of the links enables two processors to communicate with the external environment when required. The symmetry of this topology greatly simplifies the development and analysis of communication procedures. For reference the P processors are numbered from 0 to $P - 1$ as shown in figure 5.9 and arithmetic modulo P is used on the processor numbers.

PARALLEL VALUE ITERATION

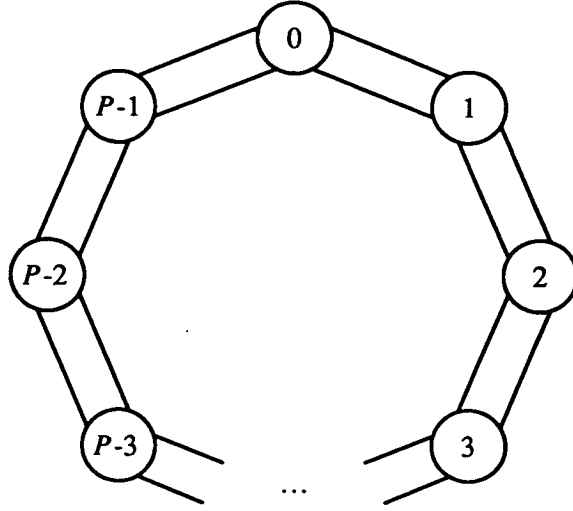


Figure 5.9: The ring topology.

BROADR: an algorithm for a multinode broadcast on a ring

Let processor r perform the following 4 tasks concurrently.

Task 1

- a Let $n = 1$
- b While $n \leq \left\lfloor \frac{P}{2} \right\rfloor$
 - read θ_{r-n}^1 from processor $r-1$ and send θ_{r-n+1}^1 to processor $r+1$ concurrently and then increase n by 1

Task 2

- a Let $n = 1$
- b While $n \leq \left\lfloor \frac{P}{2} \right\rfloor$
 - read θ_{r+n}^2 from processor $r+1$ and send θ_{r+n-1}^2 to processor $r-1$ concurrently and then increase n by 1

Task 3

- a Let $n = 1$
- b While $n < \left\lfloor \frac{P-1}{2} \right\rfloor$
 - read θ_{r-n}^2 from processor $r-1$ and send θ_{r-n+1}^2 to processor $r+1$ concurrently and then increase n by 1

Task 4

- a Let $n = 1$
- b While $n < \left\lfloor \frac{P-1}{2} \right\rfloor$
 - read θ_{r+n}^1 from processor $r+1$ and send θ_{r+n-1}^1 to processor $r-1$ concurrently and then increase n by 1

PARALLEL VALUE ITERATION

The following results show that BROADR performs a multinode broadcast on a ring topology and construct a timing model for the procedure on a transputer based parallel computer.

Proposition 5.5

BROADR will not deadlock¹.

Proof

The algorithm will not deadlock if the communications attempted at every step of all four tasks are completed successfully.

Consider task 1: when $n = t$, where $1 \leq t \leq \lfloor \frac{P}{2} \rfloor$, processor r reads θ_{r-t}^1 from processor $r - 1$ and sends θ_{r-t+1}^1 to processor $r + 1$ concurrently, so communication from 0 to 1, from 1 to 2, ..., from $P - 1$ to 0 is successful and no other communication is attempted. Hence each step of task 1 is completed successfully.

A similar argument can be used for the other three tasks.

Proposition 5.6

The messages sent by processor r during BROADR are known to it before they need to be sent.

Proof

Consider each task separately.

In task 1 if $n = 1$ processor r sends θ_r^1 which it knows initially, while if $n = t$ (where $1 < t \leq \lfloor \frac{P}{2} \rfloor$) processor r sends θ_{r-t+1}^1 which is the message it read during the most recently completed step of task 1.

A similar argument can be used for the other three tasks.

Proposition 5.7

When BROADR terminates every processor knows all P messages.

Proof

Let $g = \lfloor \frac{P-1}{2} \rfloor$ and $h = \lfloor \frac{P}{2} \rfloor$ and note that, since $g + h = P - 1$, the sets $\{\theta_{r-h}, \dots, \theta_{r+g}\}$ and $\{\theta_{r-g}, \dots, \theta_{r+h}\}$ both contain all P messages.

PARALLEL VALUE ITERATION

Processor r knows θ_r initially, receives $\{\theta_{r-1}^1, \dots, \theta_{r-h}^1\}$ and $\{\theta_{r-1}^2, \dots, \theta_{r-g}^2\}$ from processor $r-1$ and receives $\{\theta_{r+1}^1, \dots, \theta_{r+g}^1\}$ and $\{\theta_{r+1}^2, \dots, \theta_{r+h}^2\}$ from processor $r+1$.

Proposition 5.8

If BROADR is implemented on a transputer based computer and tasks are initiated in the order in which they are presented in the statement of BROADR, then all processors will receive their final messages in the multinode broadcast after the same time,

$$\begin{cases} (P+5)c + \frac{P-1}{4}\ell\rho & \text{if } P \text{ is odd} \\ (P+2)c + \frac{2P-1}{8}\ell\rho & \text{if } P \text{ is even} \end{cases} \quad (5.3)$$

after the start of communication where c is the set-up time for a channel and ρ is the transmission time for a 64bit value along a link which is used bidirectionally.

Proof

It is apparent from the statement of the algorithm that pairs of channels have to be synchronised, but the four pairs of channels can operate independently.

Case 1: P odd

On each processor four synchronised pairs of channels are used concurrently to communicate $\left\lfloor \frac{P}{2} \right\rfloor = \left\lfloor \frac{P-1}{2} \right\rfloor = \frac{P-1}{2}$ half messages each consisting of $\frac{\ell}{2}$ values. Using the model developed in section 5.3.1, the time required for this is $6c + \frac{P-1}{2}(2c + \frac{\ell}{2}\rho)$ and this expression simplifies to give the required result.

Case 2: P even

In tasks 1 and 2, two synchronised pairs of channels are used concurrently to communicate $\frac{P}{2}$ half messages each consisting of $\frac{\ell}{2}$ values. In tasks 3 and 4, two synchronised pairs of channels are used to communicate $\frac{P}{2} - 1$ half messages each consisting of $\frac{\ell}{2}$ values. Since task 1 is initiated before task 2, the time taken by the procedure is limited by the time taken by task 2. Tasks 1 and 2 are initiated before tasks 3 and 4, so there will be a delay of $2c$ before task 2 is initiated. During the final communication in tasks 1 and 2, links are used in one direction only. Therefore the

PARALLEL VALUE ITERATION

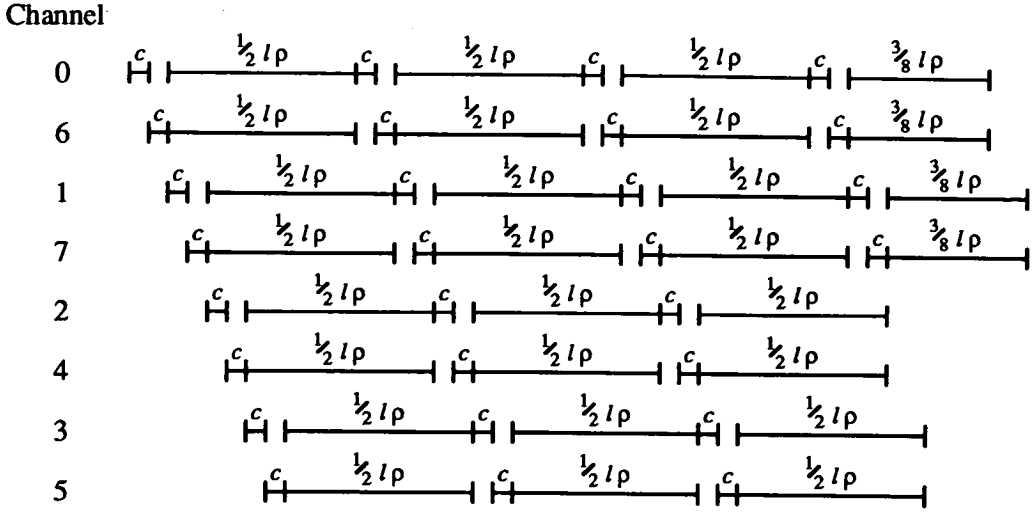


Figure 5.10: Scheduling the communication tasks in BROADCAST for the case of a 6 transputer ring.

time taken by task 2 is given by $2c + \left(\frac{P}{2} - 1\right)\left(2c + \frac{\ell}{2}\rho\right) + 2c + \frac{3\ell}{8}\rho$ and this expression simplifies to give the required result.

Figure 5.10 shows how the communication tasks in BROADCAST might be scheduled on each transputer in the ring assuming that communication tasks are initiated in the order in which they are presented in the statement of BROADCAST. Note that when P is odd the order in which the four tasks in BROADCAST are initiated does not affect the total time.

When P is odd the time required to perform a multinode broadcast on a ring using BROADCAST approaches the rate bound as the number of values in each message increases. This is not the case when P is even, but an algorithm with this property can be developed by dividing messages into 4 equal segments so that all channels can be used at the final step. As with the chain topology, it is not profitable to combine messages into longer packets unless messages are very short and when messages are very short, dividing messages is not optimal.

Least squares estimates of c and ρ were obtained for the ring by measuring the time required for a multinode broadcast using BROADCAST with different numbers of transputers and different values of ℓ and applying (5.3). These estimates are denoted

PARALLEL VALUE ITERATION

by c_2 and ρ_2 in table 5.1. The estimates of channel set-up time during a multinode broadcast on a ring are approximately three times the estimate of channel set-up time during less complicated tasks. As before this is because the model on which these estimates are based does not allow for any delays in the scheduling of communication tasks.

While the estimates of channel set-up time exhibit no trend, surprisingly the estimates of transmission time increase with P . The time required by a multinode broadcast on a ring is limited by the speed of the slowest link in the ring. Experiments have shown that the speed of a link degrades as the number of switching chips required to make the connection increases. As the number of transputers in the ring increases, the automatic wiring routine makes connections which require more switching chips and hence, the speed of the slowest link in the ring falls. An investigation into the speeds of the links in the rings used in this benchmark showed that to send 64bits along the slowest link required 7.85, 8.22, 9.83, 9.26, 10.97 and 11.15 μ s for the 9, 15, 25, 35, 49 and 121 transputer rings respectively. These times closely mirror the estimates ρ_2 shown in table 5.1.

Examination of the layout of the computer showed that regardless of the number of transputers in the ring, no connection which requires more than one switching chip need ever be made. With optimal wiring instructions the ring benchmark was repeated and the resulting estimates c_3 and ρ_3 are shown in table 5.1. The improved wiring results in more than a 30% reduction in transmission times for large rings and the apparent link speed is as good as that observed in the simple two transputer experiments of section 5.3.1.

Torus topology

In the torus topology each processor is connected to four nearest neighbours by one link, as shown in figure 5.11. As with the ring, dynamic reconfiguration of the links enables two processors to communicate with the external environment when required. Multinode broadcast procedures for this topology are more complex than for the ring and there are many procedures for which the time required approaches the rate bound

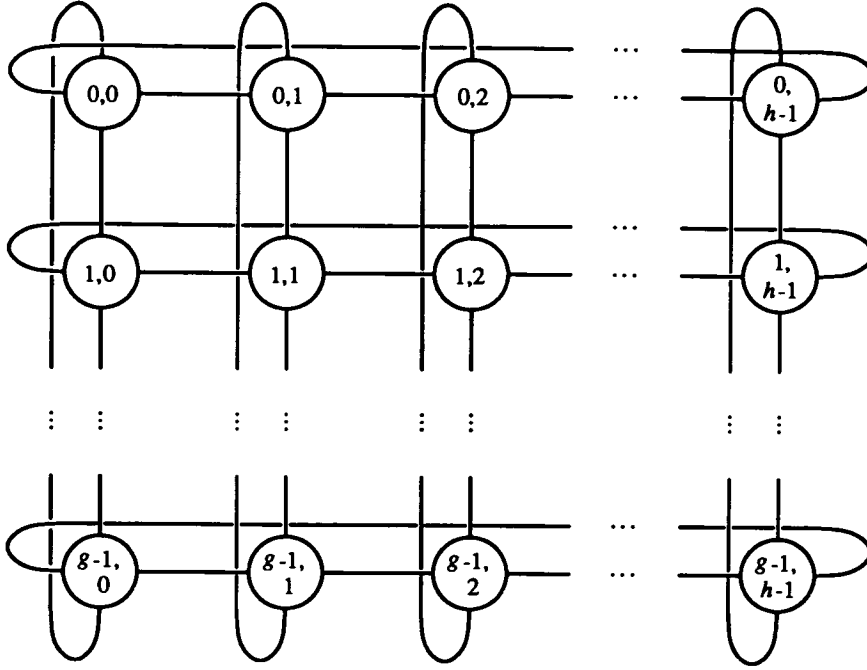


Figure 5.11: The general $g \times h$ torus topology.

as the number of values in each message increases. The advantage of this connection topology over the ring for a multinode broadcast is that, since each processor has four neighbours, it is possible to combine messages into longer packets (and so reduce the number of channel set-ups required) without delaying messages.

Let g and h denote the number of rows and columns in the torus respectively. For reference the processors are numbered in a coordinate-like fashion (see figure 5.11) using arithmetic modulo g and arithmetic modulo h on the first and second components of the coordinate respectively.

BROADT: an algorithm for a multinode broadcast on a torus

The four packets of messages defined below are used in the description of the communication tasks performed in this algorithm.

$$\begin{aligned}\phi_1^n(a, b) &= \left\{ \theta_{a-x, b+y} : x, y \in \mathbb{Z}, 0 \leq x \leq \left\lfloor \frac{g}{2} \right\rfloor, 0 \leq y \leq \left\lfloor \frac{h-1}{2} \right\rfloor - 1, x+y = n-1 \right\} \\ \phi_2^n(a, b) &= \left\{ \theta_{a+x, b+y} : x, y \in \mathbb{Z}, 0 \leq x \leq \left\lfloor \frac{g-1}{2} \right\rfloor - 1, 0 \leq y \leq \left\lfloor \frac{h-1}{2} \right\rfloor, x+y = n-1 \right\} \\ \phi_3^n(a, b) &= \left\{ \theta_{a+x, b-y} : x, y \in \mathbb{Z}, 0 \leq x \leq \left\lfloor \frac{g-1}{2} \right\rfloor, 0 \leq y \leq \left\lfloor \frac{h}{2} \right\rfloor - 1, x+y = n-1 \right\}\end{aligned}$$

PARALLEL VALUE ITERATION

$$\phi_4^n(a, b) = \left\{ \theta_{a-x, b-y} : x, y \in \mathcal{Z}, 0 \leq x \leq \left\lfloor \frac{g}{2} \right\rfloor - 1, 0 \leq y \leq \left\lfloor \frac{h}{2} \right\rfloor, x + y = n - 1 \right\}$$

Let processor (a, b) perform the following task.

- a Let $n = 1$
- b While $n \leq \left\lfloor \frac{g}{2} \right\rfloor + \left\lfloor \frac{h}{2} \right\rfloor$
 - read $\phi_1^n(a, b + 1)$ from processor $(a, b + 1)$,
 - send $\phi_1^n(a, b)$ to processor $(a, b - 1)$,
 - read $\phi_2^n(a + 1, b)$ from processor $(a + 1, b)$,
 - send $\phi_2^n(a, b)$ to processor $(a - 1, b)$,
 - read $\phi_3^n(a, b - 1)$ from processor $(a, b - 1)$,
 - send $\phi_3^n(a, b)$ to processor $(a, b + 1)$,
 - read $\phi_4^n(a - 1, b)$ from processor $(a - 1, b)$
 - and send $\phi_4^n(a, b)$ to processor $(a + 1, b)$ concurrently,
 - and then increase n by 1

The instructions performed while $n = t$ will be referred to as the t^{th} step of the algorithm. The following results show that BROADT performs a multinode broadcast on a torus topology.

Proposition 5.9

BROADT will not deadlock¹.

Proof

The algorithm will not deadlock if the communications attempted at every step are completed successfully.

Consider the first two instructions at the t^{th} step, where $1 \leq t \leq \left\lfloor \frac{g}{2} \right\rfloor + \left\lfloor \frac{h}{2} \right\rfloor$. Since $|\phi_1^t(a, b)|$ is independent of the values of a and b , all processors send and receive the same number of messages. Processor (a, b) reads $\phi_1^t(a, b + 1)$ from processor $(a, b + 1)$ and sends $\phi_1^t(a, b - 1)$ to processor $(a, b - 1)$, so communication from $(a, 0)$ to $(a, 1)$, from $(a, 1)$ to $(a, 2)$, ..., from $(a, h - 1)$ to $(a, 0)$ is successful for $a = 0, 1, \dots, g - 1$ and no other communication is attempted. Hence both instructions are completed successfully.

The other instructions can be dealt with in pairs in a similar manner.

Lemma 5.10

By the end of the t^{th} step of BROADT processor (a, b) knows the messages from all processors which can be reached from (a, b) by a path in G , the graph corresponding

PARALLEL VALUE ITERATION

to the connection topology, consisting of at most t edges.

Proof

At the 1st step processor (a, b) receives the messages being broadcast from each of its neighbours and so, since it knows $\theta_{a,b}$ initially, the result holds for $t = 1$.

Assume that the result is true for $t = u$ where $u \geq 1$ and let (c, d) be within $u + 1$ edges of (a, b) in G .

There exists a minimum edge path joining (a, b) and (c, d) consisting of x vertical edges followed by y horizontal edges where $x + y \leq u + 1$. If $x + y < u + 1$ then (c, d) is within u edges of (a, b) and so, by the inductive hypothesis, (a, b) knows $\theta_{c,d}$ by the end of the u^{th} step. If $x + y = u + 1$ then c can be written as $a + x'$ and d as $b + y'$ where $x' = \pm x$ and $y' = \pm y$. If g is odd then to satisfy the minimum edge path requirement, $-\frac{g-1}{2} \leq x' \leq \frac{g-1}{2}$. If g is even then $-\frac{g}{2} \leq x' \leq \frac{g}{2}$ for the same reason but, since $-\frac{g}{2} = \frac{g}{2} \bmod g$, x' can be assumed to satisfy $-\frac{g}{2} \leq x' \leq \frac{g}{2} - 1$. Hence $-\left\lfloor \frac{g}{2} \right\rfloor \leq x' \leq \left\lfloor \frac{g-1}{2} \right\rfloor$ and similarly $-\left\lfloor \frac{h}{2} \right\rfloor \leq y' \leq \left\lfloor \frac{h-1}{2} \right\rfloor$.

Since $u + 1 > 0$ either $|x'| > 0$ or $|y'| > 0$.

Case 1: $x' \leq 0, y' > 0$

$c = a + x', d = (b + 1) + (y' - 1)$ where $0 \leq -x' \leq \left\lfloor \frac{g}{2} \right\rfloor, 0 \leq (y' - 1) \leq \left\lfloor \frac{h-1}{2} \right\rfloor - 1$ and $-x' + (y' - 1) = u$. Hence $\theta_{c,d} \in \phi_1^{u+1}(a, b + 1)$.

Case 2: $x' > 0, y' \geq 0$

$c = (a + 1) + (x' - 1), d = b + y'$ where $0 \leq (x' - 1) \leq \left\lfloor \frac{g-1}{2} \right\rfloor - 1, 0 \leq y' \leq \left\lfloor \frac{h-1}{2} \right\rfloor$ and $(x' - 1) + y' = u$. Hence $\theta_{c,d} \in \phi_2^{u+1}(a + 1, b)$.

Case 3: $x' \geq 0, y' < 0$

$c = a + x', d = (b - 1) + (y' + 1)$ where $0 \leq x' \leq \left\lfloor \frac{g-1}{2} \right\rfloor, 0 \leq -(y' + 1) \leq \left\lfloor \frac{h}{2} \right\rfloor - 1$ and $x' - (y' + 1) = u$. Hence $\theta_{c,d} \in \phi_3^{u+1}(a, b - 1)$.

Case 4: $x' < 0, y' \leq 0$

$c = (a - 1) + (x' + 1), d = b + y'$ where $0 \leq -(x' + 1) \leq \left\lfloor \frac{g}{2} \right\rfloor - 1, 0 \leq y' \leq \left\lfloor \frac{h}{2} \right\rfloor$ and $-(x' + 1) - y' = u$. Hence $\theta_{c,d} \in \phi_4^{u+1}(a - 1, b)$.

PARALLEL VALUE ITERATION

Since during the $(u+1)^{\text{th}}$ step processor (a, b) receives $\phi_1^{u+1}(a, b+1)$, $\phi_2^{u+1}(a+1, b)$, $\phi_3^{u+1}(a, b-1)$ and $\phi_4^{u+1}(a-1, b)$, processor (a, b) knows $\theta_{c,d}$ by the end of the $(u+1)^{\text{th}}$ step.

Hence by induction the result holds.

Proposition 5.11

The messages sent by processor (a, b) during BROADT are known to it before they need to be sent.

Proof

At the 1st step processor (a, b) sends $\theta_{a,b}$, which it knows initially, to its four neighbours. At the t^{th} step, where $1 < t \leq \left\lfloor \frac{g}{2} \right\rfloor + \left\lfloor \frac{h}{2} \right\rfloor$, processor (a, b) sends $\phi_u^t(a, b)$ for $u = 1, 2, 3$ and 4 . From the definition of $\phi_u^t(a, b)$ it is clear that if $\theta_{c,d} \in \phi_u^t(a, b)$ then (c, d) is within $t-1$ edges of (a, b) in the graph corresponding to the connection topology. Hence using lemma 5.10, $\theta_{c,d}$ is known to processor (a, b) by the end of the $(t-1)^{\text{th}}$ step.

Proposition 5.12

When BROADT terminates every processor knows all P messages.

Proof

(a, b) is within $\left\lfloor \frac{g}{2} \right\rfloor$ vertical edges of any row of the torus and within $\left\lfloor \frac{h}{2} \right\rfloor$ horizontal edges of any column of the torus. Therefore all processors are within $\left\lfloor \frac{g}{2} \right\rfloor + \left\lfloor \frac{h}{2} \right\rfloor$ edges of processor (a, b) . Hence using lemma 5.10, after $\left\lfloor \frac{g}{2} \right\rfloor + \left\lfloor \frac{h}{2} \right\rfloor$ steps of the algorithm processor (a, b) knows all P messages.

In general the performance of BROADT is poor because channels are not assigned equal loads throughout the procedure. However if $g = h = 2R + 1$ for some integer R then, as the results below show, the algorithm is very efficient. In other cases the problem of unequal loads can be overcome by dividing messages into four equal segments and modifying the definition of the packets to suit the dimensions of the torus being used.

PARALLEL VALUE ITERATION

Lemma 5.13

Let $S = \{(x, y) : x, y \in \mathbb{Z}, 0 \leq x \leq a, 0 \leq y \leq b, x + y = t\}$ where a and b are positive integers satisfying $a \leq b$, then

$$|S| = \begin{cases} t + 1 & \text{for } t \leq b \\ b + 1 & \text{for } b < t \leq a \\ a + b + 1 - t & \text{for } a < t \leq a + b \\ 0 & \text{for } t > a + b \end{cases}$$

Proof

As figure 5.12 shows, there are four possible cases.

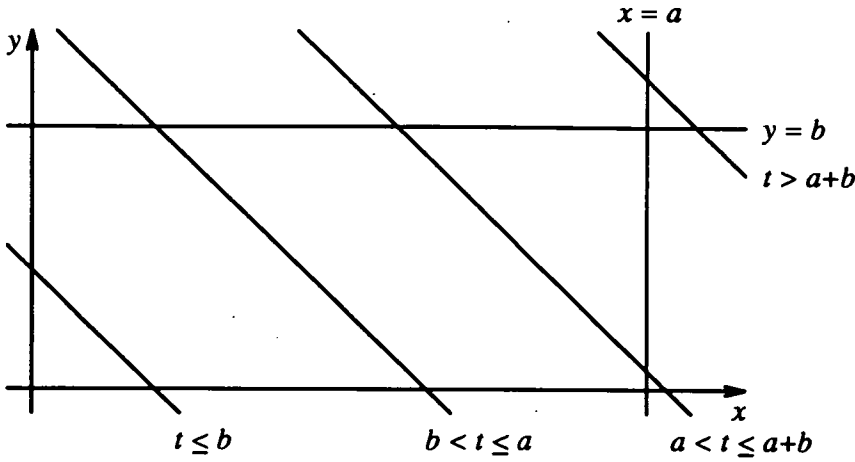


Figure 5.12: The intersection of the line $x + y = t$ with the box bounded by the x and y axes and the lines $x = a$ and $y = b$.

Case 1: $t \leq b$

The line $x + y = t$ cuts the x axis at $(t, 0)$ and the y axis at $(0, t)$. Between these two extremes there are $t - 0 + 1 = t + 1$ points on the line $x + y = t$ with integer coordinates.

Case 2: $b < t \leq a$

The line $x + y = t$ cuts the x axis at $(t, 0)$ and the line $y = b$ at $(t - b, b)$. Between these two extremes there are $t - (t - b) + 1 = b + 1$ points on the line $x + y = t$ with integer coordinates.

PARALLEL VALUE ITERATION

Case 3: $a < t \leq a + b$

The line $x + y = t$ cuts the line $x = a$ at $(a, t - a)$ and the line $y = b$ at $(t - b, b)$. Between these two extremes there are $a - (t - b) + 1 = a + b + 1 - t$ points on the line $x + y = t$ with integer coordinates.

Case 4: $t > a + b$

The line $x + y = t$ lies outside the box bounded by the x and y axes and the lines $x = a$ and $y = b$.

Lemma 5.14

If $g = h = (2R + 1)$ for some integer R then

$$(a) \quad |\phi_u^t(a, b)| = \begin{cases} t & \text{for } t \leq R \\ 2R + 1 - t & \text{for } R + 1 \leq t \leq 2R \\ 0 & \text{for } t > 2R \end{cases} \quad \text{for } u = 1, 2, 3 \text{ and } 4$$

$$(b) \quad \sum_{t=1}^{2R} |\phi_u^t(a, b)| = \frac{P - 1}{4}$$

Proof of (a)

$$\left\lfloor \frac{g}{2} \right\rfloor = \left\lfloor \frac{g-1}{2} \right\rfloor = \left\lfloor \frac{h}{2} \right\rfloor = \left\lfloor \frac{h-1}{2} \right\rfloor = R.$$

$$\begin{aligned} \text{So } |\phi_u^t(a, b)| &= |\{(x, y) : x, y \in \mathcal{Z}, 0 \leq x \leq R - 1, 0 \leq y \leq R, x + y = t - 1\}| \\ &= \begin{cases} t & \text{for } t - 1 \leq R - 1 \\ R & \text{for } t - 1 = R \\ 2R + 1 - t & \text{for } R < t - 1 \leq 2R - 1 \\ 0 & \text{for } t - 1 > 2R - 1 \end{cases} \quad \text{by lemma 5.13.} \end{aligned}$$

Hence result.

Proof of (b)

$$\begin{aligned} \sum_{t=1}^{2R} |\phi_u^t(a, b)| &= \sum_{t=1}^R t + \sum_{t=R+1}^{2R} (2R + 1 - t) \\ &= \frac{1}{2}R(R + 1) + R(2R + 1) - R^2 - \frac{1}{2}R(R + 1) \\ &= R^2 + R = \frac{P - 1}{4} \end{aligned}$$

PARALLEL VALUE ITERATION

Proposition 5.15

If $g = h = 2R + 1$ for some integer R then all channels are assigned an equal load throughout BROADT without dividing or repeatedly sending messages.

Proof

It is clear from the statement of the algorithm that divided messages are not used.

During the t^{th} step of the algorithm, $|\phi_u^t(a, b)|$ messages pass along every link in both directions. It has already been noted that $|\phi_u^t(a, b)|$ is independent of the values of a and b and lemma 5.14 (a) shows that if $g = h = 2R + 1$, $|\phi_u^t(a, b)|$ is also independent of u . So all channels are assigned an equal load throughout the t^{th} step and hence throughout the procedure.

Proposition 5.12 shows that $P - 1$ messages are read by each processor and lemma 5.14 (b) shows that if $g = h = 2R + 1$ each processor reads exactly $\frac{P-1}{4}$ messages from each channel. Hence no processor reads any message more than once.

Note that it is possible to show (by verifying that $\phi_1^t(a, b+1)$, $\phi_2^t(a+1, b)$, $\phi_3^t(a, b-1)$ and $\phi_4^t(a-1, b)$ where $1 \leq t \leq \left\lfloor \frac{g}{2} \right\rfloor + \left\lfloor \frac{h}{2} \right\rfloor$ are disjoint sets) that when using BROADT for general tori no processor reads any message more than once.

To understand the way in which channels have to be synchronised during BROADT, and therefore to develop a timing model for the procedure, it is necessary to understand the relationships between the sets $\phi_1^{t-1}(a, b+1)$, $\phi_2^{t-1}(a+1, b)$, $\phi_3^{t-1}(a, b-1)$ and $\phi_4^{t-1}(a-1, b)$ and the sets $\phi_u^t(a, b)$ for $u = 1, 2, 3$ and 4 . By examining the definition of $\phi_u^t(a, b)$ for $u = 1, 2, 3$ and 4 it is easy to verify that when $g = h = 2R + 1$ for some integer R the following relationships hold.

$$\phi_1^t(a, b) = \begin{cases} \{\theta_{a,b}\} & \text{if } t = 1 \\ \phi_1^{t-1}(a, b+1) \cup \{\theta_{a-t+1,b}\} & \text{if } 1 < t \leq R \\ \phi_1^{t-1}(a, b+1) \setminus \{\theta_{a,b+R}\} \cup \{\theta_{a-R,b}\} & \text{if } t = R+1 \\ \phi_1^{t-1}(a, b+1) \setminus \{\theta_{a-t+R+1,b+R}\} & \text{if } R+1 < t \leq 2R \end{cases}$$

Note that $\theta_{a-t,b} \in \phi_4^t(a-1, b)$ for $1 \leq t \leq R$.

This relationship is illustrated in figure 5.13 for the case $R = 3$.

PARALLEL VALUE ITERATION

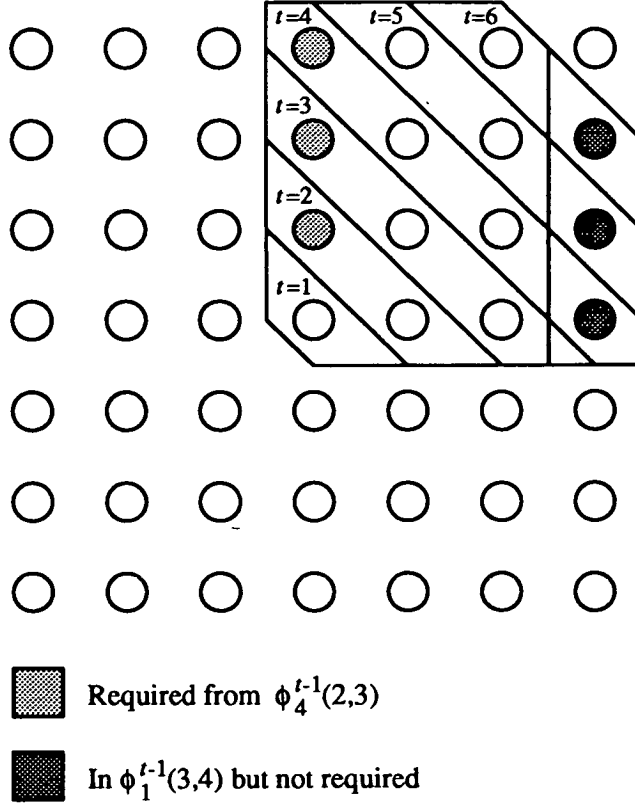


Figure 5.13: The relationship between $\phi_1^t(3,3)$ and the sets $\phi_1^{t-1}(3,4)$ and $\phi_4^{t-1}(2,3)$ when $P = 7^2$.

$$\phi_2^t(a, b) = \begin{cases} \{\theta_{a,b}\} & \text{if } t = 1 \\ \phi_2^{t-1}(a+1, b) \cup \{\theta_{a,b+t-1}\} & \text{if } 1 < t \leq R \\ \phi_2^{t-1}(a+1, b) \setminus \{\theta_{a+R,b}\} \cup \{\theta_{a,b+R}\} & \text{if } t = R+1 \\ \phi_2^{t-1}(a+1, b) \setminus \{\theta_{a+R,b+t-R-1}\} & \text{if } R+1 < t \leq 2R \end{cases}$$

Note that $\theta_{a,b+t} \in \phi_1^t(a, b+1)$ for $1 \leq t \leq R$.

$$\phi_3^t(a, b) = \begin{cases} \{\theta_{a,b}\} & \text{if } t = 1 \\ \phi_3^{t-1}(a, b-1) \cup \{\theta_{a+t-1,b}\} & \text{if } 1 < t \leq R \\ \phi_3^{t-1}(a, b-1) \setminus \{\theta_{a,b-R}\} \cup \{\theta_{a+R,b}\} & \text{if } t = R+1 \\ \phi_3^{t-1}(a, b-1) \setminus \{\theta_{a+t-R-1,b-R}\} & \text{if } R+1 < t \leq 2R \end{cases}$$

Note that $\theta_{a+t,b} \in \phi_2^t(a+1, b)$ for $1 \leq t \leq R$.

$$\phi_4^t(a, b) = \begin{cases} \{\theta_{a,b}\} & \text{if } t = 1 \\ \phi_4^{t-1}(a-1, b) \cup \{\theta_{a,b-t+1}\} & \text{if } 1 < t \leq R \\ \phi_4^{t-1}(a-1, b) \setminus \{\theta_{a-R,b}\} \cup \{\theta_{a,b-R}\} & \text{if } t = R+1 \\ \phi_4^{t-1}(a-1, b) \setminus \{\theta_{a-R,b-t+R+1}\} & \text{if } R+1 < t \leq 2R \end{cases}$$

Note that $\theta_{a,b-t} \in \phi_3^t(a, b-1)$ for $1 \leq t \leq R$.

PARALLEL VALUE ITERATION

These relations show that for $1 < t \leq R+1$, $\phi_1^t(a, b)$ depends upon $\phi_1^{t-1}(a, b+1)$ and $\phi_4^{t-1}(a-1, b)$; $\phi_2^t(a, b)$ depends upon $\phi_2^{t-1}(a+1, b)$ and $\phi_1^{t-1}(a, b+1)$; $\phi_3^t(a, b)$ depends upon $\phi_3^{t-1}(a, b-1)$ and $\phi_2^{t-1}(a+1, b)$ and $\phi_4^t(a, b)$ depends upon $\phi_4^{t-1}(a-1, b)$ and $\phi_3^{t-1}(a, b-1)$. Hence for the first R steps of BROADT it is necessary to keep all eight channels synchronised. However for $R+1 < t \leq 2R$, $\phi_1^t(a, b)$ depends upon $\phi_1^{t-1}(a, b+1)$ only, $\phi_2^t(a, b)$ depends upon $\phi_2^{t-1}(a+1, b)$ only, $\phi_3^t(a, b)$ depends upon $\phi_3^{t-1}(a, b-1)$ only and $\phi_4^t(a, b)$ depends upon $\phi_4^{t-1}(a-1, b)$ only. Hence during the final R steps of BROADT four pairs of synchronised channels can operate independently. Therefore if channel set-ups on neighbouring transputers occur in the sequence which minimises delays, the time required to perform a multinode broadcast on a torus consisting of $(2R+1)^2$ transputers using BROADT is given by

$$\begin{aligned} & \sum_{t=1}^R (8c + |\phi_1^t(a, b)|\ell\rho) + 6c + \sum_{t=R+1}^{2R} (2c + |\phi_1^t(a, b)|\ell\rho) \\ &= 6c + 10Rc + \frac{P-1}{4}\ell\rho = (5\sqrt{P} + 1)c + \frac{P-1}{4}\ell\rho \end{aligned}$$

where c is the set-up time for a channel and ρ is the transmission time for a 64bit value along a link which is used bidirectionally.

When the messages consist of components of a vector, the overheads involved with the use of internal copying of data to prepare the packets of messages to be sent and to split the packets of messages received far outweigh the benefits of reduced total channel set-up time. This is the case in parallel value iteration where the messages consist of estimates of the values of some states. Note that when $P = 9$ no messages can be combined, so internal copying of data is not an issue. In other cases it is possible to avoid internal copying of data by having a different ordering of the state space on each transputer. Within each message states are placed in the order in which they are updated. Within packets $\phi_1^t(a, b)$ and $\phi_2^t(a, b)$ messages are placed in order of ascending first subscript (for example when $P = 7^2$, $\phi_1^4(3, 3) = \{\theta_{1,4}, \theta_{2,5}, \theta_{3,6}\}$ see figure 5.13). Within packets $\phi_3^t(a, b)$ and $\phi_4^t(a, b)$ messages are placed in order of descending first subscript. On transputer (a, b) packets are ordered as follows.

$$\theta_{a,b}, \phi_1^1(a, b+1), \phi_2^1(a+1, b), \phi_3^1(a, b-1), \phi_4^1(a-1, b), \dots, \\ \phi_1^{2R}(a, b+1), \phi_2^{2R}(a+1, b), \phi_3^{2R}(a, b-1), \phi_4^{2R}(a-1, b)$$

This allows each transputer to place messages in packets received straight into the correct locations in memory with no internal copying. Further transputer (a, b) stores the messages comprising packets $\phi_2^t(a, b)$, $\phi_3^t(a, b)$ and $\phi_4^t(a, b)$ in contiguous memory locations, so internal copying is not required to prepare these packets of messages for transmission. When $2 \leq t \leq R+1$ messages in $\phi_1^t(a, b)$ are not stored in contiguous memory locations on transputer (a, b) , so to avoid internal copying transmitting this packet requires two channel set-ups — one for all or part of $\phi_1^{t-1}(a, b+1)$ and one for $\theta_{a-t+1, b}$. The additional channel set-up is overlapped with transmission on other channels and prolongs the time required to communicate $\phi_1^t(a, b)$ and $\phi_1^t(a, b+1)$ by one channel set-up. If transputers initiate the communication of $\phi_1^t(a, b)$ and $\phi_1^t(a, b+1)$ before the communication of other packets then, when $2 \leq t \leq R$, the additional time is less than the delay which is necessary to synchronise channels and so will not affect the overall time. However when $t = R+1$ four independent pairs of channels are used concurrently on each transputer and the synchronisation of channels does not require a delay. Consequently the additional channel set-up affects the overall communication time. For $t > R+1$, messages in $\phi_1^t(a, b)$ are stored in contiguous memory locations on transputer (a, b) . Hence assuming the communication of $\phi_1^t(a, b)$ and $\phi_1^t(a, b+1)$ is initiated before the communication of other packets, the time required to perform a multinode broadcast on a torus consisting of $P = (2R+1)^2$ transputers using this implementation of BROADT is given by

$$(5\sqrt{P} + 2)c + \frac{P-1}{4}\ell\rho \quad (5.4)$$

where c is the set-up time for a channel and ρ is the transmission time for a 64bit value along a link which is used bidirectionally. If messages consist of components of a vector and a multinode broadcast has to be performed many times (as in an iterative algorithm for example) then, because the reordering need only be performed

PARALLEL VALUE ITERATION

once, the second implementation of BROADT is more efficient. Figure 5.14 shows how the communication tasks in BROADT might be scheduled on each transputer in the torus assuming that communication tasks are initiated in the order in which they are presented in the statement of BROADT.

When $P = (2R + 1)^2$ the time required to perform a multinode broadcast on a torus using BROADT approaches the rate bound as the number of values in each message increases. As before when messages are very short improvements can be obtained by delaying the communication of certain packets so that even longer packets can be created.

Least squares estimates of c and ρ were obtained for the torus by measuring the time required for a multinode broadcast using BROADT with different numbers of transputers and different values of ℓ and applying (5.4). These estimates are denoted by c_4 and ρ_4 in table 5.1. The estimates of channel set-up time during a multinode broadcast on a torus are approximately twice the estimate of channel set-up time during less complicated tasks. As before this is because the model does not allow for delays in the scheduling of communication tasks. Despite using optimal wiring instructions in the benchmarks, the estimates of transmission time increase with P because of the increase in the number of switching chips which have to be used. The effect of this is negligible for small P , but as P increases the degradation in the link speed becomes more of a factor. For a torus with 121 transputers the transmission time is $8.50\mu s$ per 64bit value, about 10% greater than for a ring of the same size. Hence for large P a ring will be more efficient than a torus when message lengths are large.

5.3.3 Procedures for global maximisations and minimisations

During a global maximisation (or minimisation) the value of $\alpha = \max_t\{\alpha_t\}$ (or $\alpha = \min_t\{\alpha_t\}$) is calculated from the values $\alpha_1, \dots, \alpha_P$, where α_t is local to processor t . A global maximisation is required in all the parallel iteration algorithms described in section 5.2, but only parallel PJ & PB requires a global minimisation. As the two problems are obviously closely related, only the maximisation problem will be addressed explicitly in this section.

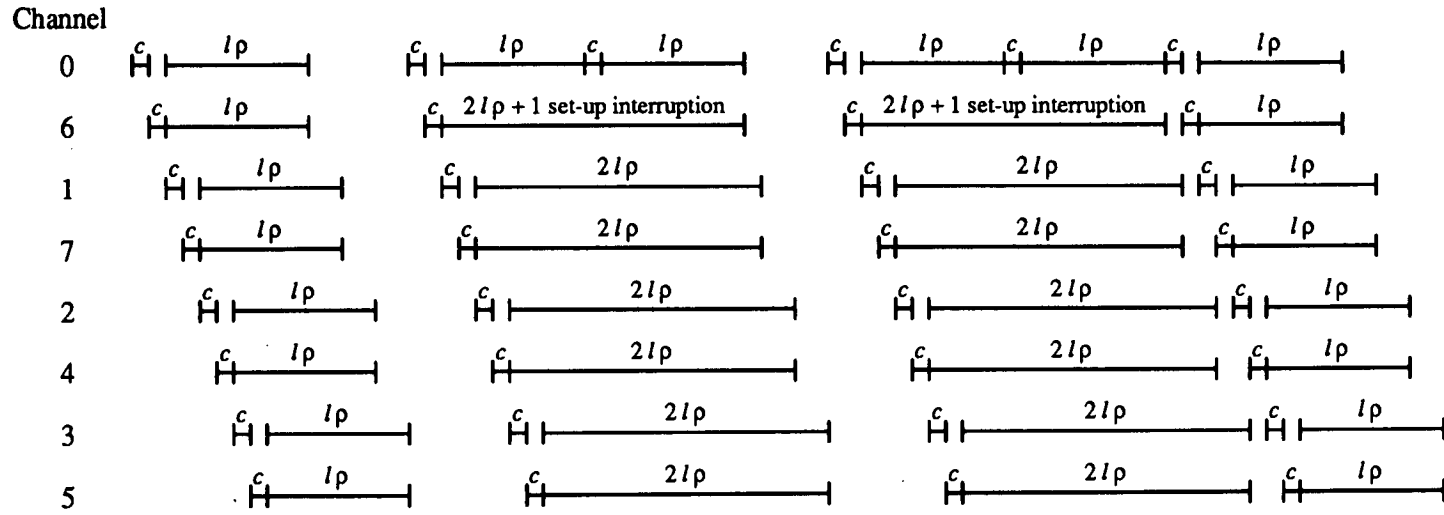


Figure 5.14: Scheduling the communication tasks in BROADT for the case of a 25 transputer torus.

PARALLEL VALUE ITERATION

A simple way to achieve a global maximisation is to perform a multinode broadcast so that the values $\alpha_1, \dots, \alpha_P$ are known by all the processors, and then to perform the maximisation on each processor. The calculation performed on each processor during this procedure requires $P - 1$ comparisons. Channel set-ups dominate the communication time because each processor has only one value to broadcast. Hence to perform the multinode broadcast efficiently on a chain or a ring, messages should not be split and the communication of some messages should be delayed so that longer packets of messages can be created. Although in a torus when BROADT is used messages are combined into longer packets, it is more efficient to delay the communication of some packets so that even longer packets can be created.

In a global maximisation two values α_r and α_t can be combined into one value equal to $\max\{\alpha_r, \alpha_t\}$, so in this case creating longer packets is not necessary. For all three topologies it is more efficient to accumulate the maximum value into one processor and then to broadcast it from that processor, because this technique reduces the number of channel set-ups required and distributes the work required to calculate the maximum value. On a chain or a ring the following procedure can be used.

GMAX: an algorithm for a global maximisation on a chain or a ring

Let t denote $\left\lfloor \frac{P}{2} \right\rfloor$.

Let processor 1 perform the following steps sequentially.

- a₁ send $\omega_1^1 = \alpha_1$ to processor 2
- b₁ read α from processor 2

Let processor r , where $2 \leq r < t + 1$, perform the following steps sequentially.

- a₁ read ω_{r-1}^1 from processor $r - 1$
- a₂ calculate $\omega_r^1 = \max\{\omega_{r-1}^1, \alpha_r\}$
- a₃ send ω_r^1 to processor $r + 1$
- b₁ read α from processor $r + 1$
- b₂ send α to processor $r - 1$

Let processor $t + 1$ perform the following steps sequentially.

- a₁ read ω_t^1 from processor t and read ω_{t+2}^2 from processor $t + 2$ concurrently
- a₂ calculate $\alpha = \max\{\omega_t^1, \alpha_{t+1}, \omega_{t+2}^2\}$
- b₁ send α to processor t and processor $t + 2$ concurrently

PARALLEL VALUE ITERATION

Let processor r , where $t + 2 \leq r < P$, perform the following steps sequentially.

- a₁ read ω_{r+1}^2 from processor $r + 1$
- a₂ calculate $\omega_r^2 = \max\{\omega_{r+1}^2, \alpha_r\}$
- a₃ send ω_r^2 to processor $r - 1$
- b₁ read α from processor $r - 1$
- b₂ send α to processor $r + 1$

Let processor P perform the following steps sequentially.

- a₁ send $\omega_P^2 = \alpha_P$ to processor $P - 1$
- b₁ read α from processor $P - 1$

It is easy to verify that $\omega_r^1 = \max\{\alpha_1, \dots, \alpha_r\}$ and $\omega_r^2 = \max\{\alpha_r, \dots, \alpha_P\}$ and hence that the steps in GMAX identified by a_i comprise a procedure which accumulates the maximum value into processor $\lfloor \frac{P}{2} \rfloor + 1$. The steps in GMAX identified by b_i comprise a procedure which broadcasts α from processor $\lfloor \frac{P}{2} \rfloor + 1$. The entire procedure is illustrated in figure 5.15.

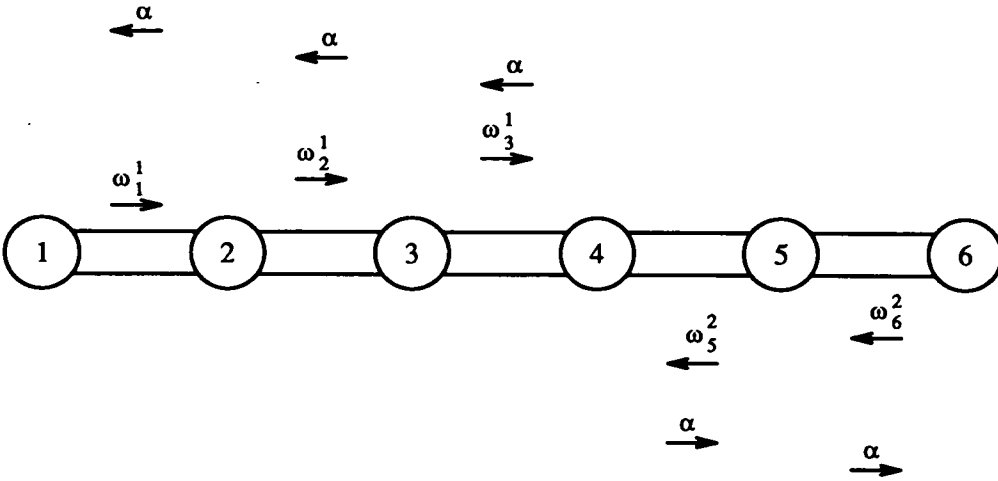


Figure 5.15: The communication required during GMAX for the case of a 6 processor chain or ring.

Proposition 5.16

If GMAX is implemented on a transputer based computer and tasks are initiated in the order in which they are presented in the statement of GMAX, the communication required takes $4c + \frac{3}{2}\rho$ if $P = 3$ and $Pc + \frac{3}{2}\lfloor \frac{P}{2} \rfloor \rho$ otherwise, while the calculation required is 1 comparison if $P = 2$ and $\lfloor \frac{P}{2} \rfloor + 1$ comparisons otherwise.

PARALLEL VALUE ITERATION

Proof

During GMAX two tasks are performed concurrently — a message is relayed from processor 1 to processor $\lfloor \frac{P}{2} \rfloor + 1$ and back to processor 1 and a message is relayed from processor P to processor $\lfloor \frac{P}{2} \rfloor + 1$ and back to processor P . Since no processor sends and reads simultaneously, no link is used bidirectionally.

Case 1: P even

If P is even then processor $\lfloor \frac{P}{2} \rfloor + 1$ is closer to processor P than to processor 1. Hence the time taken by GMAX is given by the time taken on processor 1. The message relayed from processor 1 to processor $\lfloor \frac{P}{2} \rfloor + 1$ reaches processor $\lfloor \frac{P}{2} \rfloor + 1$ after $\frac{P}{2} \left(c + \frac{3}{4}\rho \right)$ and $\frac{P}{2} - 1$ comparisons. There is then a delay of one comparison if $P = 2$ and two comparisons if $P \geq 4$ (while α is calculated) before the message relayed from processor $\lfloor \frac{P}{2} \rfloor + 1$ to processor 1 is ready to be sent. Assuming communication tasks are initiated in the order in which they are presented in the statement of GMAX, α is read by processor 1 after a further $\frac{P}{2} \left(c + \frac{3}{4}\rho \right)$.

Case 2: P odd

If P is odd then processors 1 and P are equidistant from processor $\lfloor \frac{P}{2} \rfloor + 1$ and, if communication tasks are initiated in the order in which they are presented in the statement of the algorithm, the time taken by GMAX is given by the time taken on processor P . The message relayed from processor P to processor $\lfloor \frac{P}{2} \rfloor + 1$ reaches processor $\lfloor \frac{P}{2} \rfloor + 1$ after $2c + \frac{3}{4}\rho$ if $P = 3$ and $\frac{P-1}{2} \left(c + \frac{3}{4}\rho \right)$ and $\frac{P-3}{2}$ comparisons if $P \geq 5$. There is then a delay of two comparisons (while α is calculated) and c (while the channel used to send α to processor $\lfloor \frac{P}{2} \rfloor$ is set up) before the message relayed from processor $\lfloor \frac{P}{2} \rfloor + 1$ to processor P is ready to be sent. This message is read by processor P after a further $\frac{P-1}{2} \left(c + \frac{3}{4}\rho \right)$.

GMAX can be incorporated into a procedure for a global maximisation on a general $g \times h$ torus as follows. Firstly use the steps of GMAX identified by a_i to accumulate $\max_r \{ \alpha_{i,r} \}$ into processor $(t, 0)$ for $1 \leq t \leq g$. Secondly use GMAX to accumulate and broadcast $\max_t \{ \max_r \{ \alpha_{i,r} \} \} = \alpha$ in column 0 of the torus. Finally use the steps of

PARALLEL VALUE ITERATION

GMAX identified by b_i to broadcast α in each row of the torus. The time required by this procedure is equal to the time required for GMAX on a g processor chain plus the time required for GMAX on an h processor chain.

If a global maximisation is to be performed at the same time as a multinode broadcast then it is possible to combine the two tasks.

In a chain it is possible to employ channels which are not used throughout BROADCAST by adding the following two steps to task 1 in BROADCAST

- c⁺ if $r = 1$ let $\omega_1^1 = \alpha_1$, otherwise read ω_{r-1}^1 from processor $r - 1$ and then calculate $\omega_r^1 = \max\{\omega_{r-1}^1, \alpha_r\}$
- d send ω_r^1 to processor $r + 1$ (provided $r \neq P$)

and adding the following two steps to task 2 in BROADCAST

- c⁺ if $r = P$ let $\omega_P^2 = \alpha_P$, otherwise read ω_{r+1}^2 from processor $r + 1$ and then calculate $\omega_r^2 = \max\{\omega_{r+1}^2, \alpha_r\}$
- d send ω_r^2 to processor $r - 1$ (provided $r \neq 1$)

In tasks 1 and 2, steps c and c⁺ can be performed concurrently. Since $\alpha = \max\{\omega_r^1, \omega_r^2\}$ each processor can calculate α via one further comparison.

The following argument shows that on a transputer based computer the global maximisation can be performed on the fly. On processors 1 and P step c⁺ requires no communication or calculation. On all other processors one value is read and one comparison is performed during step c⁺ and one half message is sent during step c. If the messages being broadcast consist of ten or more values, step c⁺ can be overlapped with step c without affecting the time required by step c. If tasks are initiated in the order in which they are presented in the statement of BROADCAST then, without the above modifications, on every processor task 1 is completed $4c$ before task 3 and task 2 is completed $4c$ before task 4, see figure 5.8. Since step d only requires one value to be read (which can be achieved in less than $4c$) and step d is added to tasks 1 and 2 only, the above modifications will not affect the time required for BROADCAST on any processor.

If P is odd every channel is used at every step of BROADCAST and if P is even there is very little redundancy, see figure 5.10. If $P = (2R + 1)^2$ for some integer R every

PARALLEL VALUE ITERATION

channel is used at every step of BROADT. Hence the above approach cannot be used for a ring or a torus. However processor r can combine θ_r and α_r into one packet to broadcast during the multinode broadcast. Each processor can then calculate α via $P - 1$ comparisons. For a ring the packet needs to have $\ell + 2$ values so that it can be halved and therefore, on a transputer based computer, the communication time is increased by $\frac{P-1}{2}\rho$ if P is odd and $\frac{2P-1}{4}\rho$ if P is even, see (5.3). If $P = (2R + 1)^2$ for some integer R then, on a transputer based computer, the communication time for a torus is increased by $\frac{P-1}{4}\rho$, see (5.4). If internal copying is used to prepare the packets to be sent and to split the packets received, the additional overheads outweigh the reduction in communication time which can be obtained. However in a Markov decision process if α_i is stored immediately before the estimate of the value of state n_i on each processor, the need for internal copying of data is removed. This can be achieved by relabelling the state space.

Benchmarks have shown that for the three topologies considered, combining a global maximisation with a multinode broadcast is more efficient than performing the two tasks separately.

The PB require a global maximisation and a global minimisation to be performed at the same time. The method of GMAX can be used to accumulate the maximum and minimum values into one processor. Compared with GMAX, the length of the messages relayed during this procedure is doubled and twice as many comparisons are performed on each processor. This additional load can be reduced by accumulating the maximum value and the minimum value in different processors. However on a chain using two processors which are far apart greatly increases the distance over which messages have to be relayed and so increases the time required by the procedure. On a ring (or a row or column of a large torus) this is not the case and using processors 1 and $\left\lfloor \frac{P}{2} \right\rfloor + 1$ leads to a highly efficient procedure. Conflict arises when two values reach a processor at the same time. On a transputer based computer this conflict results in a delay of c and possibly one comparison, and for $P > 8$ the total delay is at most $2c$ and one comparison.

PARALLEL VALUE ITERATION

On a chain a global maximisation and minimisation can be performed concurrently with a multinode broadcast and, when a transputer based computer is used, the overall time will not be affected if the messages being broadcast are long. On a ring the two tasks can be combined with a multinode broadcast without increasing the communication time further, but the calculation required is doubled. On a torus both the increase in communication time and the calculation required are doubled. Benchmarks have shown that on a chain combining a global maximisation and minimisation with a multinode broadcast is more efficient than performing the tasks separately, but on a ring or a torus this is not the case.

5.3.4 Conclusions

Since the problems to be considered involve broadcasting long messages and it is often efficient to reduce the frequency of the bound calculation, the communication overheads in the algorithms described in section 5.2 will be dominated by the time required by the multinode broadcast procedure. Therefore it is important to choose a topology on which a multinode broadcast can be performed quickly. Comparing the timing models for the multinode broadcast procedures on a chain and a ring ((5.2) and (5.3) respectively), it is apparent that the chain is never competitive for a multinode broadcast. Although it has been shown that a multinode broadcast on a torus requires fewer channel set-ups than on a ring, there are a number of reasons for preferring a ring. Firstly when the messages being broadcast are long, the improvement resulting from the reduced total channel set-up time will be negligible. Secondly achieving an efficient multinode broadcast on a non-square torus is a complex task, so using a torus severely restricts the choice of the number of processors used. Also it is not possible to wire a large torus efficiently on a Meiko computing surface with electronic switching, so the improvement due to fewer channel set-ups must be balanced against longer transmission times.

The benchmarks on which the least squares estimates of channel set-up time and transmission time for a ring are based suggest that, when messages consist of 20 or more values, transmission time dominates channel set-up time during a multinode broadcast.

PARALLEL VALUE ITERATION

Since in GS2 the messages broadcast consist of only one value, it is not efficient to use half messages. In this case BROADR is modified by removing tasks 3 and 4 and letting $\theta_r^1 = \theta_r$ for $r = 1, \dots, P$. This modification means that the number of channel set-ups required is halved and no link need ever be used bidirectionally.

GMAX will be used whenever a bound calculation is performed. This technique is more efficient for the PB which require both a global maximisation and a global minimisation and, since combining the global maximisation required for the $L_\infty B$ with the multinode broadcast performed at the end of each iteration is a complex task and reduces the benefits of less frequent bound calculation, GMAX is preferred for the $L_\infty B$.

5.4 Implementing synchronous non-overlapped parallel value iteration

5.4.1 Introduction

This section compares the performance of the parallel value iteration algorithms described in section 5.2 using the six small test problems (the properties of which are summarised in table 3.3). The algorithms are implemented on a ring of transputers using the procedures BROADR and GMAX described in section 5.3 to perform the communication tasks required. The reasons for the choices of topology and procedures are set out in section 5.3.4.

5.4.2 Partitioning the state space

The choice of partition will affect the time required for the communication in the algorithms, the load balancing and the convergence of GS and SOR. If each processor is allocated an equal number of states then communication overheads will be minimised. However by dividing messages it is possible to organise communication so that any imbalance in the number of states per processor has a small effect on communication time. It is therefore more important to choose a partition which ensures good load balancing and, where appropriate, good convergence. The form of the variations in the number of actions per state and the number of transitions per action affects how easy it is to select a partition which distributes the work evenly among the processors.

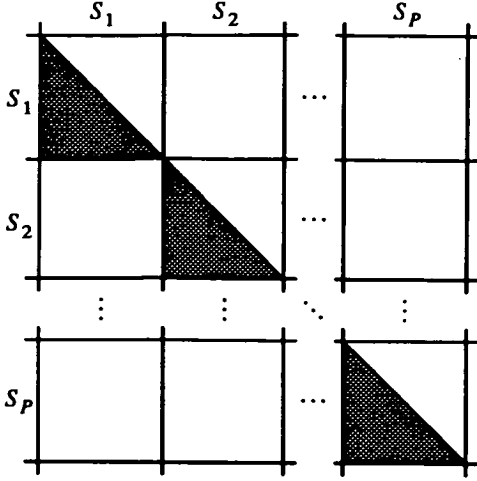


Figure 5.16: Area 1, the critical area for GS1.

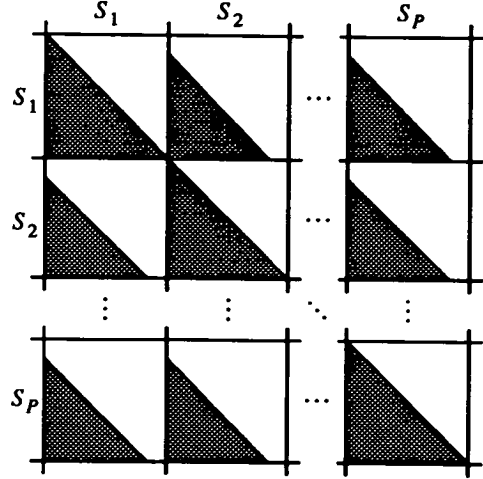


Figure 5.17: Area 2, the critical area for GS2.

Systematic changes with the order of the states make it difficult to maintain load balancing as actions are eliminated.

In the case of PJ, calculating v_i^n requires $|K_i| - 1$ comparisons and $2 \sum_{k \in K_i} |J_i^k|$ additions and multiplications, where $J_i^k = \{p_{i,j}^k : p_{i,j}^k > 0, j \in S\}$, while calculating the PB requires two comparisons and one addition for each state. Thus a processor which is allocated the states in $A \subset S$ must perform $C(A) = \sum_{i \in A} (|K_i| + 2 \sum_{k \in K_i} |J_i^k| + 2)$ operations at each iteration. In structured problems it is often possible to derive expressions for $|K_i|$ and $\sum_{k \in K_i} |J_i^k|$ in terms of i and k .

Satisfactory load balancing can usually be achieved by choosing n_r to minimise $\left| C(S_{r-1}) - \frac{1}{P-r+2} C(S \setminus \bigcup_{i=1}^{r-2} S_i) \right|$ for $r > 1$. With action elimination the operation count falls between iterations, but this method will ensure approximate load balancing when most of the actions are present and this is the stage at which most of the work is done.

Given a partition $\{S_i\}_{i=1}^N$ of the state space, let area 1 of a transition matrix be the shaded area in figure 5.16 and let area 2 be the shaded area in figure 5.17. Areas 1 and 2 correspond to transitions from a state i to states in T_i (see (2.4)) for GS1 and GS2 respectively.

For high efficiency the choice of the partition and the ordering of the states within

PARALLEL VALUE ITERATION

each set of the partition must give good load balancing and maximise the weight in area 1, for GS1, or area 2, for GS2, of the transition matrices used. For GS1 an ordering of the state space is chosen and then, using the method above, a partition which gives good load balancing is chosen. For GS2 an ordering of the state space is chosen and then each processor is allocated (as far as possible) an equal number of states. This approach will not result in good load balancing if states which are updated on different processors at the same step take different times to update.

Recalling (5.1), the amount of data required by a processor which has been allocated the states in $A \subset S$ is equal to $\sum_{i \in A} (|K_i| + \sum_{k \in K_i} |J_i^k|) + |S|$. Comparing this with the form of $C(A)$ above shows that balancing the computational load will usually give a reasonably even distribution of data. This is important for very large problems where processor memory is a constraint.

5.4.3 Results

The times presented in the result tables below are in seconds and were obtained using a Meiko computing surface consisting of T800-20MHz transputers. As before the solution time does not include the time required to obtain the data initially. In HOWARD and INVENT1, the data is generated locally on each transputer and the time required for this is small compared to the solution time. In the four generated problems, the data is read from disk and the time required is highly variable due to multiuser access. Details of the notation used in the tables can be found on page xiv. The solution time for $N = 1$ refers to the serial implementation of that algorithm and the efficiency for a particular algorithm is relative to this solution time. Note that serial GS1 and GS2 are equivalent to GS. Communication overheads and efficiency do not add to 100%. This is due to imperfect load balancing and (for GS1 and GS2) poorer convergence, both of which tend to become worse as the number of processors used increases. There is also an effect due to the increase in the proportion of data stored in the transputers' on-chip memory (which allows faster access) as the number of transputers used increases. In several cases the effect of this gives an efficiency of greater than 100% and in many other cases $\xi + \xi_{comm}$ is greater than 100%.

Table 5.2: Synchronous non-overlapped parallel pre-Jacobi

		PJ & PB			PJ & PB with PAE2		
P	N_{iter}	T_{soln}	ξ	ξ_{comm}	T_{soln}	ξ	ξ_{comm}
HOWARD							
1	141	6.33	100.0	0.0	1.59	100.0	0.0
5	141	1.17	108.3	3.5	0.35	90.1	11.5
11	141	0.64	89.3	12.9	0.24	60.4	34.6
25	141	0.47	54.2	40.1	0.28	22.6	66.4
INVENT1							
1	93	57.41	100.0	0.0	13.97	100.0	0.0
5	93	11.34	101.2	0.3	2.99	93.5	1.2
11	93	5.32	98.1	1.2	1.54	82.2	4.0
25	93	2.63	87.3	4.9	0.87	64.5	14.8
49	93	1.74	67.3	13.2	0.76	37.6	30.4
GEN1FF							
1	613	595	100.0	0.0	220	100.0	0.0
5	613	116	102.9	1.1	43	102.8	3.0
11	613	54	100.9	2.8	21	97.3	7.2
25	613	25	95.5	7.4	10	85.5	17.9
49	613	14	84.3	15.5	7	64.8	32.3
GEN1CV							
1	230	223	100.0	0.0	223	100.0	0.0
5	230	43	103.0	1.1	44	102.4	1.1
11	230	20	100.9	2.8	20	99.8	2.7
25	230	9	95.5	7.4	9	94.1	7.3
49	230	5	84.0	15.5	6	81.0	14.9
GEN1CS							
1	1508	1464	100.0	0.0	1148	100.0	0.0
5	1508	284	103.0	1.1	219	105.0	1.4
11	1508	132	100.9	2.8	102	102.3	3.6
25	1508	61	95.5	7.4	48	95.4	9.4
49	1508	35	84.4	15.5	29	79.9	18.8
GEN2							
1	26	32.52	100.0	0.0	34.78	100.0	0.0
5	26	6.33	102.8	0.8	6.62	105.0	0.8
11	26	2.94	100.7	2.1	3.08	102.8	2.0
25	26	1.36	95.9	5.7	1.42	98.2	5.5
49	26	0.76	87.0	12.5	0.79	89.7	12.0

Random orderings of the state space are used in INVENT1 and GEN2 and the WLT orderings are used in the other four cases.

PARALLEL VALUE ITERATION

Table 5.3: Synchronous non-overlapped parallel Gauss-Seidel using the WLT orderings of the state space

P	LGS1 & $L_\infty B$ with $L_\infty AE2$					LGS2 & $L_\infty B$ with $L_\infty AE2$				
	N_{iter}	T_{soln}	t_{aver}	ξ	ξ_{comm}	N_{iter}	T_{soln}	t_{aver}	ξ	ξ_{comm}
HOWARD										
1	77	2.19	0.028	100.0	0.0	77	2.19	0.028	100.0	0.0
5	179	0.78	0.004	56.2	6.3	130	0.69	0.005	63.9	11.5
11	230	0.55	0.002	36.1	24.0	200	0.63	0.003	31.5	23.5
25	411	0.86	0.002	10.2	62.7	387	0.98	0.003	9.0	41.6
INVENT1										
1	29	7.87	0.271	100.0	0.0	29	7.87	0.271	100.0	0.0
5	68	3.02	0.044	52.1	1.0	46	3.58	0.078	43.9	2.2
11	114	2.60	0.023	27.5	3.0	72	2.66	0.037	26.9	5.1
25	174	1.98	0.011	15.9	12.3	118	2.06	0.017	15.2	13.1
49	205	1.61	0.008	10.0	31.6	148	1.76	0.012	9.1	22.4
GENIFF										
1	198	151	0.76	100.0	0.0	198	151	0.76	100.0	0.0
5	272	37	0.14	81.1	1.5	204	40	0.20	74.9	8.5
11	289	19	0.06	74.2	3.8	215	25	0.12	54.9	15.3
25	314	10	0.03	62.7	9.7	235	17	0.07	36.7	28.6
49	357	7	0.02	45.5	19.2	279	15	0.05	20.6	35.3
GENICS										
1	230	184	0.80	100.0	0.0	230	184	0.80	100.0	0.0
5	318	46	0.14	80.0	1.4	244	51	0.21	72.4	8.1
11	340	23	0.07	72.6	3.6	259	32	0.12	53.0	14.6
25	378	12	0.03	59.8	9.2	299	22	0.07	33.6	27.4
49	441	9	0.02	42.7	18.2	361	20	0.06	18.8	34.1
P	LGS1 & $L_\infty B$					LGS2 & $L_\infty B$				
	N_{iter}	T_{soln}	t_{aver}	ξ	ξ_{comm}	N_{iter}	T_{soln}	t_{aver}	ξ	ξ_{comm}
GENICV										
1	145	142	0.98	100.0	0.0	145	142	0.98	100.0	0.0
5	188	35	0.19	80.4	1.1	147	38	0.26	75.4	6.5
11	201	17	0.09	73.9	2.8	158	23	0.15	55.6	12.1
25	222	9	0.04	63.0	7.3	171	14	0.08	39.4	23.8
49	256	6	0.02	48.0	15.4	204	12	0.06	24.0	32.0

PARALLEL VALUE ITERATION

Table 5.4: Synchronous non-overlapped parallel Gauss-Seidel using random orderings of the state space

	RGS1 & $L_\infty B$ with $L_\infty AE2$					RGS2 & $L_\infty B$ with $L_\infty AE2$				
P	N_{iter}	T_{soln}	t_{aver}	ξ	ξ_{comm}	N_{iter}	T_{soln}	t_{aver}	ξ	ξ_{comm}

HOWARD

1	322	5.57	0.017	100.0	0.0	322	5.57	0.017	100.0	0.0
5	497	1.49	0.003	74.7	9.1	323	1.34	0.004	83.1	14.6
11	530	1.01	0.002	50.2	30.2	373	1.03	0.003	49.3	27.0
25	530	1.06	0.002	21.0	65.3	413	1.04	0.003	21.5	42.1

INVENT1

1	118	18.4	0.156	100.0	0.0	118	18.4	0.156	100.0	0.0
5	194	5.1	0.026	72.5	1.4	126	6.4	0.051	57.6	3.4
11	206	2.7	0.013	62.8	5.1	122	3.7	0.030	45.4	6.2
25	215	1.6	0.007	47.1	18.8	142	2.2	0.016	33.0	14.6
49	215	1.4	0.007	26.1	36.8	152	1.8	0.012	20.8	22.4

GEN1FF

1	977	679	0.70	100.0	0.0	977	679	0.70	100.0	0.0
5	1714	223	0.13	61.0	1.5	1007	189	0.19	72.0	9.0
11	1839	112	0.06	54.9	3.9	1025	108	0.11	57.1	16.9
25	1904	55	0.03	49.0	10.2	1018	66	0.07	41.1	30.9
49	1933	34	0.02	40.6	20.5	1022	52	0.05	26.6	37.2

GEN1CS

1	1382	1093	0.79	100.0	0.0	1382	1093	0.79	100.0	0.0
5	2572	390	0.15	56.1	1.3	1368	284	0.21	77.0	8.1
11	2786	197	0.07	50.3	3.4	1448	169	0.12	58.8	15.3
25	2893	97	0.03	45.3	8.9	1431	100	0.07	43.6	28.6
49	2950	59	0.02	37.9	18.1	1459	78	0.05	28.5	35.3

	RGS1 & $L_\infty B$					RGS2 & $L_\infty B$				
P	N_{iter}	T_{soln}	t_{aver}	ξ	ξ_{comm}	N_{iter}	T_{soln}	t_{aver}	ξ	ξ_{comm}

GEN1CV

1	640	627	0.98	100.0	0.0	640	627	0.98	100.0	0.0
5	1107	208	0.19	60.2	1.1	666	172	0.26	72.9	6.5
11	1203	104	0.09	54.8	2.8	670	95	0.14	60.1	12.6
25	1253	51	0.04	49.7	7.3	672	55	0.08	45.6	24.5
49	1274	30	0.02	42.9	15.4	667	38	0.06	33.4	33.0

GEN2

1	1148	1449	1.26	100.0	0.0	1148	1449	1.26	100.0	0.0
5	2209	538	0.24	53.9	0.8	1160	399	0.34	72.7	4.9
11	2476	279	0.11	47.2	2.1	1161	212	0.18	62.0	9.7
25	2639	137	0.05	42.2	5.7	1167	114	0.10	51.1	20.6
49	2737	79	0.03	37.5	12.6	1209	79	0.07	37.3	28.9

PARALLEL VALUE ITERATION

For each problem and for most algorithms considered, figure 5.18 shows a plot of $(\text{number of processors used}) \times (\text{solution time})$ against number of processors used. For a given problem and algorithm this quantity is inversely proportional to efficiency. A plot with gradient zero corresponds to the ideal case of 100% efficiency for all numbers of processors. The results show that $(\text{number of processors used}) \times (\text{solution time})$ tends to increase as the number of processors used increases. This is due to the effect of increasing communication overheads, increasing load imbalance and, in the case of GS, increasing number of iterations. As the behaviour of RGS is similar in all three GEN1 problems, RGS1 and RGS2 have been omitted from the graphs for GEN1FF and GEN1CV so that a larger scale can be used. The GS algorithms plotted in the graphs for GEN1CV and GEN2 do not use action elimination because this technique is not worthwhile for these problems.

5.4.4 Analysis

Synchronous non-overlapped parallel pre-Jacobi

PJ & PB works well on all six problems. In HOWARD and the four generated problems the choice of partition ensures good load balancing and the drop in efficiency is almost entirely due to the communication overheads. In INVENT1 the number of actions per state ranges from 1 to 120, so some states involve much more work than others. As a consequence of this, as the number of processors used increases it becomes harder to achieve good load balancing without reordering the state space. The poor load balance is the major factor limiting the efficiency of PJ & PB when $P > 11$. Achieving a good load balance is harder when the WLT ordering of the state space is used because the number of actions then falls from 120 in state 0 to 1 in state 119. Hence, since the ordering of the state space does not affect the convergence of PJ, using a random ordering is more efficient. Using the WLT ordering increases solution times for PJ & PB by as much as 12.5%.

The choice of partition ensures good load balancing for PJ & PB with PAE2 initially, but as the algorithm proceeds this load balance is lost due to the elimination of actions. Ultimately all but one of the actions in each state would be eliminated.

PARALLEL VALUE ITERATION

○ PJ & PB
□ PJ & PB with PAE2

△ LGS1 & L_∞B (with L_∞AE2)
▽ LGS2 & L_∞B (with L_∞AE2)

◇ RGS1 & L_∞B (with L_∞AE2)
× RGS2 & L_∞B (with L_∞AE2)

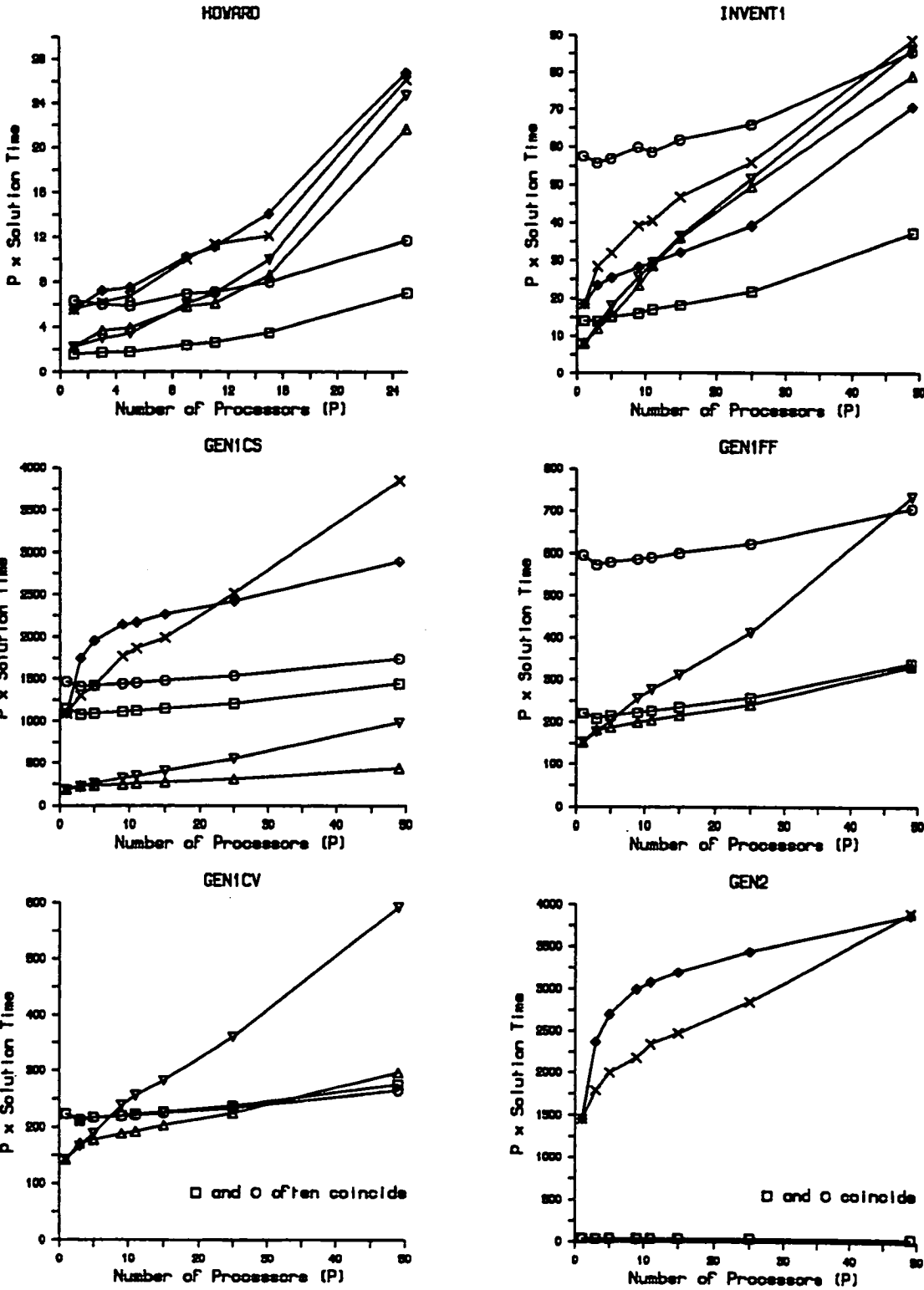


Figure 5.18: A graphical presentation of the results.

PARALLEL VALUE ITERATION

In HOWARD good load balancing is regained after all but one of the actions in each state have been eliminated. The drop in efficiency is largely due to communication overheads, which are expected to be high as the workload per state is very small.

In INVENT1 communication overheads account for only a small part of the drop in efficiency. The major part is due to poor load balancing which in this problem remains even when there is only one action in each state. Since the workload per action is almost constant, load balancing with one action per state can be achieved by allocating an equal number of states to each processor. However when this was tried, there was a slight increase in solution times due to the poor initial load balancing. A random ordering of the state space results in a better load balance when there is only one action per state because the maximum number of states allocated to a processor is smaller. In this case using the WLT ordering increases solution times by as much as 48.5%. As the algorithm progresses the partitioning could be modified to reflect the pattern of action elimination. It is likely that this will work particularly well in problems where all the data can be stored on every processor, but not so well if data has to be passed between processors in order to modify the partition.

Load balancing is not a problem in any of the four generated problems when PJ & PB with PAE2 is used. The drop in efficiency is almost entirely due to the increase in the communication overheads. This is what is expected when the state space is large compared to the number of processors used and the number of actions per state and the work per action do not depend in a systematic way on the order of the state space.

The efficiency of PJ & PB with PAE2 is generally less than that of PJ & PB because communication overheads are more significant and in some cases the load balance is poorer. Despite this whenever action elimination is successful in the serial case, it remains successful in the parallel case (see figure 5.18).

Synchronous non-overlapped parallel Gauss-Seidel

With GS1 and GS2, as with PJ, the average time per iteration falls as the number of processors used increases, but unlike PJ, the number of iterations required rises. This means that the efficiency on more than one processor is low. For example using

PARALLEL VALUE ITERATION

11 processors reduces the average time per iteration in LGS1 & $L_\infty B$ with $L_\infty AE2$ by a factor of 11.8 in HOWARD and by a factor of 11.9 in INVENT1, but increases the number of iterations required by 3.0 times and 3.9 times respectively. The resulting efficiency is 36.1% in HOWARD and 27.5% in INVENT1. (Note that in both these cases, although communication overheads and load imbalance are significant when 11 processors are used, the average time per iteration is reduced by a factor which is greater than the number of processors used. This is because the additional iterations performed involve little work due to the success of action elimination.) Eventually the rate of convergence of GS1 and GS2 will be as poor as that of Jacobi. For example in HOWARD when 11 or more processors are used and in INVENT1 when 25 or more processors are used, RGS1 & $L_\infty B$ requires as many iterations as J & $L_\infty B$.

In GS2 the increase in the number of iterations is not as severe as in GS1 but, because the communication time per iteration is higher and the load balance is generally poorer, the average time per iteration does not fall as sharply as in GS1. For example using 11 processors reduces the average time per iteration in LGS2 & $L_\infty B$ with $L_\infty AE2$ by a factor of 9.0 in HOWARD and by a factor of 7.3 in INVENT1, but increases the number of iterations required by 2.6 times and 2.5 times respectively.

As the load balance in GS2 is generally poorer than in GS1 and the additional iterations required by GS1 involve little work when action elimination is successful, the average time per iteration is less in GS1. Consequently communication overheads can be greater in GS1 than in GS2 even though the communication time per iteration is less.

In HOWARD every state requires approximately the same amount of work to update when all actions are feasible and after all but one action in each state has been eliminated, so the load balance in GS2 is good. Hence the performance of LGS2 relative to that of LGS1 is better than in most other cases.

As mentioned above, with the WLT ordering of the state space the work per state in INVENT1 varies systematically. Consequently the load balance in LGS1 is poor and in LGS2 every state updated by processor P involves less work than every state updated

PARALLEL VALUE ITERATION

by processor 1, so processor P must wait prior to each synchronised communication. As the work per state is highly variable, the load balance in RGS2 is also poor. Only in RGS1 can approximate load balancing be achieved and so RGS1 & $L_{\infty}B$ with $L_{\infty}AE2$ becomes the best GS algorithm as the number of processors used increases (see figure 5.18). In HOWARD and the three GEN1 problems the rates of convergence of LGS1 and LGS2 are dramatically better than those of RGS1 and RGS2, and for these problems LGS1 is the best GS variant. In GEN2 no WLT ordering of the state space exists and RGS2 is the best GS variant.

In GEN1FF using 49 processors increases the number of iterations required by 80% for GS1 and 41% for GS2 when the WLT ordering of the state space is used and by 98% for GS1 and 5% for GS2 when a random ordering is used. This is typical of the behaviour of all the test problems. When the WLT ordering of the state space is used most of the weight in the transition matrix corresponding to the optimal policy is concentrated just below the diagonal and therefore lies in areas 1 and 2 of this matrix (see figures 5.16 and 5.17). Although the additional weight in area 2 improves the convergence, it is not sufficient to compensate for the additional overheads of GS2. Hence GS1 is generally more efficient. This point is illustrated by the graphs for the three GEN1 problems in figure 5.18. When a random ordering of the state space is used there is more weight in area 2 of the transition matrices than in area 1, because area 2 is larger. Consequently the convergence of GS2 is much better than GS1 and as a result GS2 is frequently better than GS1, see the graph for GEN2 in figure 5.18.

The multinode broadcast procedure used in GS2 can be made more efficient by combining the messages broadcast into longer packets, but this improvement would not make GS2 competitive with GS1 when a WLT ordering of the state space is used. In conclusion although GS2 requires far fewer iterations than GS1, the increase in the number of synchronisation points, resulting in poorer load balancing, and the increase in the communication time per iteration, mean that GS1 is generally better than GS2 unless no WLT ordering of the state space exists.

Comparing synchronous non-overlapped parallel pre-Jacobi and Gauss-Seidel

As already mentioned, the efficiency of GS is low because the number of iterations required increases as the number of processors used increases, whereas with PJ the number of iterations required remains constant. As a result PJ & PB with PAE2 is more efficient than any GS algorithm.

In HOWARD the best serial algorithm is PJ & PB with PAE2 and this advantage is retained as the number of processors used increases. In INVENT1 although LGS & $L_\infty B$ with $L_\infty AE2$ is the best serial algorithm, PJ & PB with PAE2 becomes the best algorithm to use as the number of processors used increases because of the poor load balance in LGS1 and LGS2.

In GEN1FF and GEN1CS the best serial algorithm is LGS & $L_\infty B$ with $L_\infty AE2$ and LGS1 & $L_\infty B$ with $L_\infty AE$ remains the best algorithm as the number of processors used increases. Due to the slow mixing and strong WLT structure in every transition matrix in GEN1CS, LGS2 & $L_\infty B$ with $L_\infty AE$ is also better than PJ & PB with PAE2. In GEN1CV PJ & PB becomes better than LGS1 & $L_\infty B$ as the number of processors used increases because, since action elimination is not successful, the additional iterations performed impose a significant workload.

GEN2 demonstrates that serial PJ & PB is far better than serial RGS & $L_\infty B$ when the transition matrices are not WLT but allow a lot of mixing and that the relative advantage of PJ becomes greater as the number of processors used increases.

All these points are illustrated in figure 5.18.

Synchronous non-overlapped parallel successive over-relaxation

As in the serial case trial and error was used to determine a good choice for the relaxation parameter for SOR1 & heuristic $L_\infty B$ for each problem. This choice is denoted by ω^* in table 5.5. These results show that even with good choices for the relaxation parameter, the number of iterations required generally increases significantly with the number of processors used. In GEN1CV the number of iterations required actually falls as the number of processors used increases. However in this case using

Table 5.5: Synchronous non-overlapped parallel SOR1 & heuristic $L_\infty B$

P	1	3	5	9	11	15	25	49
HOWARD								
ω^*	1.05	1.00 [†]	1.02 [‡]	1.03	1.03	1.03	1.04	—
N_{iter}	42	162	157	211	202	242	379	—
F_{iter}	1.00	3.86	3.74	5.02	4.81	5.76	9.02	—
INVENT1								
ω^*	1.04	1.00 [†]	1.02	1.00 [†]	1.00 [†]	1.06	1.14	1.26
N_{iter}	14	46	57	88	106	124	127	136
F_{iter}	1.00	3.29	4.07	6.29	7.57	8.86	9.07	9.71
GEN1FF								
ω^*	1.02	1.03 [‡]	1.03 [‡]	1.04	1.04	1.04	1.05	1.04
N_{iter}	112	147	153	139	143	151	141	213
F_{iter}	1.00	1.31	1.37	1.24	1.28	1.35	1.26	1.90
GEN1CV								
ω^*	1.02 [‡]	1.03	1.04	1.04	1.04	1.04	1.05	1.04
N_{iter}	72	62	39	51	55	66	53	118
F_{iter}	1.00	0.86	0.54	0.71	0.76	0.92	0.74	1.64
GEN1CS								
ω^*	1.02	1.02 [‡]	1.02 [‡]	1.03	1.03 [‡]	1.03 [‡]	1.03 [‡]	1.03 [‡]
N_{iter}	128	200	206	191	194	206	234	295
F_{iter}	1.00	1.56	1.61	1.49	1.52	1.61	1.83	2.30
GEN2								
ω^*	1.24	1.31	1.32	1.31	1.26	1.30	1.31	1.31
N_{iter}	419	1144	1134	1587	1825	1817	1822	2004
F_{iter}	1.00	2.73	2.71	3.79	4.36	4.34	4.35	4.78

[†] in these cases the algorithm does not converge for $\omega > 1.02$.

[‡] in these cases the algorithm does not converge for $\omega > 1.03$.

The case $\omega^* = 1$ does not correspond to GS1 & $L_\infty B$ because, as noted in section 2.2, a heuristic stopping rule is used for SOR.

In GEN2 a random ordering of the state space is used, but in all other problems the WLT orderings are used.

$\omega^* + 0.01$ instead of ω^* increases the number of iterations required by more than 18 times for six of the eight values of P shown. GEN1CS is another interesting example because for six of the eight values of P shown, the algorithm failed to converge when $\omega > 1.03$. It is also worth noting that in four cases the optimal value of ω is 1. Furthermore as in the serial case, the resulting approximation to v^* is not guaranteed to satisfy the convergence criterion in (1.3) and action elimination cannot be employed. Hence SOR1

PARALLEL VALUE ITERATION

is not a suitable solution method for Markov decision processes because its behaviour cannot be predicted and failure to use the optimal value for ω can lead to very poor convergence.

5.4.5 Conclusions

The analysis in section 5.4.4 shows that PJ and GS have greatly improved performance when suitably implemented on a parallel computer. Using 49 processors the solution times are reduced by a factor of more than 40 in some cases. The corresponding increase in memory means that much larger problems can be solved than previously.

In terms of efficiency it is clear that PJ is superior to GS. Although parallel algorithms using action elimination do not tend to be as efficient as those without, action elimination is still a worthwhile technique. In PJ without action elimination the only problem is communication overheads, but the introduction of action elimination causes a load balancing problem as well because the distribution of work then changes from iteration to iteration. The only way to deal with this problem is by some form of dynamic load balancing. However in general performance is satisfactory because most work is done during the early iterations when the initial load balancing works reasonably well.

Both parallel GS algorithms exhibit slower convergence as the number of processors used increases. However despite the low efficiency GS remains competitive with PJ for slowly mixing problems in which a WLT ordering of the state space can be used.

The results for the generated problems suggest that for large problems with few positive transition probabilities per action, communication overheads will be the major limiting factor for PJ.

5.5 Asynchronous parallel value iteration

Section 5.4 concludes that high communication overheads are the major factor limiting the speed-up obtainable with synchronous non-overlapping parallel PJ, while high communication overheads and slow convergence are responsible for the low

PARALLEL VALUE ITERATION

efficiency of synchronous non-overlapping parallel GS1.

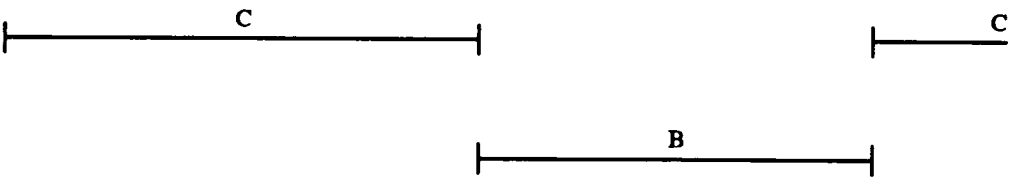
Bertsekas [10] suggests modifying the communication procedure for the multinode broadcast so that a processor only sends the latest estimate of the value of a state it updates to processors which require that estimate in their calculation. While this modification may be useful for problems with special structure, it is unlikely to reduce solution times for parallel value iteration algorithms on transputer networks in general for the following reasons. Firstly if the number of actions per state is large then, even when many transputers are used to solve highly sparse problems, the average number of transitions with positive probabilities per transputer is greater than the number of states in the problem and so, unless the problem has special structure, all transputers will require estimates of the values of most states in their calculation. For example in GEN1FF the average number of transitions with positive probabilities per state is 96, so only when more than 96 transputers are used does the average number of transitions with positive probabilities per transputer fall below the number of states. Secondly since each transputer can only communicate directly with at most four other transputers, many messages will have to be routed through intermediate transputers and so, in general, a transputer will have to handle many more estimates than it requires in its calculation. Finally with this approach it will be difficult to avoid using internal copying of data to prepare the messages to be sent and to split the messages received.

For problems in which action elimination is successful, the number of transitions with positive probabilities per transputer falls as the value iteration algorithms progress. However if the modification suggested above is not efficient initially, solution times still may not be improved because detecting when it is worthwhile switching communication procedures will involve a significant overhead.

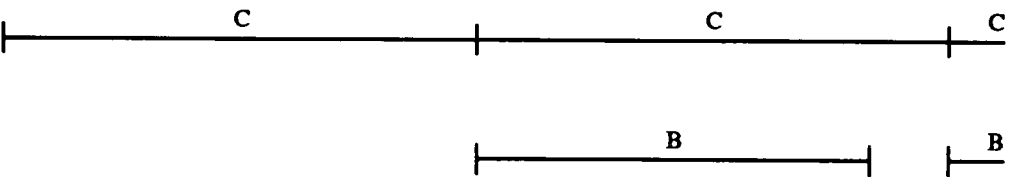
Figure 5.19(a) illustrates how synchronous non-overlapped parallel PJ and GS1 are scheduled on a processor. Updating the states is represented by the operation C and this must be complete before the communication, represented by the operation B, can commence. Subsequent calculation cannot take place until the communication is complete and this delay means that communication overheads can be significant. This

PARALLEL VALUE ITERATION

(a) Synchronous case with no overlap



(b) Asynchronous case with overlap



(c) Phased Pipeline Algorithm

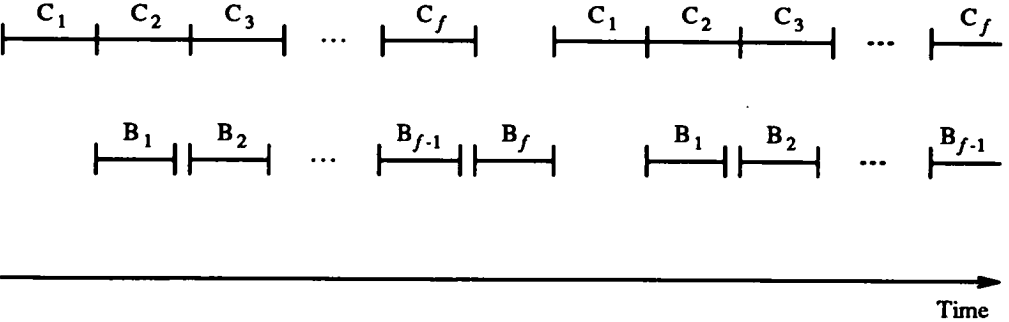


Figure 5.19: A comparison of the three schedules.

effect might be reduced by applying an asynchronous algorithm which performs less frequent communication or overlaps calculation and communication.

One such algorithm would allow the calculation at an iteration to start as soon as the calculation required at the previous iteration is complete, with each processor using the most recent updates available to it for states not updated locally. Figure 5.19(b) shows how this algorithm might be scheduled on a processor for the case in which calculation and communication can be overlapped without penalty. As before

PARALLEL VALUE ITERATION

updating the states is represented by the operation C and communication is represented by the operation B. Comparing this with figure 5.19(a) it is apparent that the average time per update is far less, due to the reduced communication overheads. As there are no known bounds relating the latest approximation in an asynchronous algorithm to the optimal solution, one iteration of a synchronous algorithm must be performed periodically to test for convergence. Details of two asynchronous algorithms which adopt this approach are given below. The first is based on PJ & PB and the second on GS1 & $L_\infty B$. The mappings E and F , defined in (2.1) and (2.2) respectively, are used to simplify the description of the calculation performed.

Asynchronous PJ & PB

Let processor r perform the following steps sequentially.

Step 1 Initialisation

Let $v_i = 0 \forall i \in S$, let $d = 1$ and calculate $u_i = (Ev)_i \forall i \in S_r$

Step 2 Perform $D - 1$ multinode broadcasts and $D - 1$ sets of updates

While $d < D$

Let $v_i = u_i \forall i \in S_r$, increase d by 1 and then concurrently send $u_i \forall i \in S_r$ to all other processors, read $u_i \forall i \notin S_r$ (overwriting v_i with u_i in the process) and calculate $u_i = (Ev)_i \forall i \in S_r$ sequentially in the order of increasing state number

Step 3 Synchronise processors

Let $w_i = u_i \forall i \in S_r$ and send $w_i \forall i \in S_r$ to all other processors and read $w_i \forall i \notin S_r$

Step 4 Perform a synchronous PJ & PB calculation

Calculate $u_i = (Ew)_i \forall i \in S_r$, $U_r = \max_{i \in S_r} \{u_i - w_i\}$ and $L_r = \min_{i \in S_r} \{u_i - w_i\}$

Step 5 Complete the bound calculation

Perform a global maximisation and minimisation to calculate $U = \max_{i \in S} \{u_i - w_i\}$ and $L = \min_{i \in S} \{u_i - w_i\}$ from the values $U_1, \dots, U_P$ and $L_1, \dots, L_P$ respectively

Step 6 Check for convergence

If $\frac{\beta}{1 - \beta}(U - L) > \epsilon$, then let $d = 1$ and repeat from step 2

PARALLEL VALUE ITERATION

Asynchronous GS1 & $L_\infty B$

Let processor r perform the following steps sequentially.

Step 1 Initialisation

Let $v_i = 0 \forall i \in S$, let $d = 1$ and calculate $v_i = (Fv)_i \forall i \in S_r$ sequentially in the order of increasing state number

Step 2 Perform $D - 1$ multinode broadcasts and $D - 1$ sets of updates

While $d < D$

 Increase d by 1 and then concurrently send $v_i \forall i \in S_r$ to all other processors, read $v_i \forall i \notin S_r$ and calculate $v_i = (Fv)_i \forall i \in S_r$ sequentially in the order of increasing state number

Step 3 Synchronise processors

Send $v_i \forall i \in S_r$ to all other processors and read $v_i \forall i \notin S_r$

Step 4 Perform a synchronous GS1 & $L_\infty B$ calculation

Calculate $u_i = (Gv)_i \forall i \in S_r$, with $T_i = \{n_r, \dots, i-1\}$, and $N_r = \max_{i \in S_r} |u_i - v_i|$

Step 5 Complete the bound calculation

Perform a global maximisation to calculate $\|u - v\|_\infty$ from the values $N_1, \dots, N_P$

Step 6 Check for convergence

If $\frac{\beta}{1 - \beta} \|u - v\|_\infty > \frac{\epsilon}{2}$, then let $d = 1$, let $v_i = u_i \forall i \in S$ and repeat from step 2

As bounds can only be calculated after the state updates performed in step 4, an action elimination test can only be applied during the first set of updates performed in step 2. Since two successive synchronous iterations are never performed, only PAE1 and L_∞ AE1 are valid. With $D = 1$ the convergence properties of these algorithms are equivalent to those of the synchronous PJ and GS1 algorithms described in section 5.2. However with $D = 2$ the rate of convergence of these algorithms is much slower. Experiments have shown that with $D > 1$, increasing the value of D reduces the average time required per set of state updates at the expense of slightly poorer convergence. The results shown in table 5.6 were obtained using a ring of transputers with $D = 25$. Details of the notation used in the tables can be found on page xiv.

Table 5.6: Convergence of the asynchronous algorithms with $D = 25$

P	Asynchronous PJ & PB				Asynchronous GS1 & $L_\infty B$			
	5	11	25	49	5	11	25	49
HOWARD								
N_{upd}	451	626	676	—	226	351	601	—
F_{upd}	3.20	4.44	4.79	—	1.26	1.53	1.46	—
INVENT1								
N_{upd}	276	276	326	326	101	151	226	326
F_{upd}	2.97	2.97	3.51	3.51	1.49	1.32	1.30	1.59
GEN1FF								
N_{upd}	3076	3126	3076	3076	301	351	376	476
F_{upd}	5.02	5.10	5.02	5.02	1.11	1.21	1.20	1.33
GEN1CV								
N_{upd}	3051	3076	2976	2976	226	251	276	351
F_{upd}	13.3	13.4	12.9	12.9	1.20	1.25	1.24	1.37
GEN1CS								
N_{upd}	3076	3151	3126	3126	351	401	451	601
F_{upd}	2.04	2.09	2.07	2.07	1.10	1.18	1.19	1.36
GEN2								
N_{upd}	3001	3076	3376	3426	2351	2726	3176	3376
F_{upd}	115	118	130	132	1.06	1.10	1.20	1.23

With PJ random orderings of the state space are used in INVENT1 and GEN2, but the WLT orderings are used in the other four cases. With GS1 a random ordering of the state space is used in GEN2 and the WLT orderings are used in the other five cases.

The results in table 5.6 show that the asynchronous version of PJ exhibits a much slower rate of convergence than synchronous PJ. This is because the good convergence properties of PJ depend upon the iteration matrices being stochastic² and asynchronous versions of PJ do not have this property. Asynchronous PJ performs best with GEN1CS (where a two fold increase in the number of updates required is recorded), because the eigen-structure of this problem is not well suited to PJ. In GEN2, which has a small 2nd eigenvalue, the asynchronous algorithm performs more than 115 times as many updates as synchronous PJ. In the other problems the increase in the number of updates performed is between 3 and 14 times. In no case could a reduction in communication overheads compensate for the increase in the calculation required.

The results in table 5.6 show that asynchronous GS1 exhibits a slower rate of convergence than synchronous GS1, although this effect is not as great as for PJ. The

PARALLEL VALUE ITERATION

overheads involved with overlapping calculation and communication on a transputer prevent asynchronous LGS1 out performing synchronous LGS1 for HOWARD using 25 transputers and INVENT1 using 49 transputers. In all other cases a reduction in communication overheads could not be expected to compensate for the increase in the number of updates performed.

The convergence of an asynchronous algorithm in which processors communicate less frequently will exhibit even poorer convergence and therefore it is likely that the effect of the increase in the number of updates performed will outweigh the benefits of reduced communication overheads in this case as well.

In conclusion the slow rate of convergence exhibited by asynchronous parallel PJ and GS outweighs the benefits of reduced communication overheads and therefore these algorithms are not efficient solution methods for Markov decision processes.

5.6 The phased pipeline algorithm

It has been shown that for PJ separate calculation and communication results in good convergence at the expense of high communication overheads, while with totally overlapped calculation and communication, communication overheads are low, but convergence is poor. Splitting the calculation performed on each processor into phases creates the opportunity to overlap calculation and communication without affecting convergence. This is the basis of the phased pipeline algorithm described below.

Let f denote the number of phases. As before the state space is partitioned into P sets denoted by $S_1, \dots, S_P$, but unlike before each set S_i in this partition is itself partitioned into f sets denoted by $S_{i,1}, \dots, S_{i,f}$.

The phased pipeline algorithm applied to PJ & PB

Let processor r perform the following steps sequentially.

Step 1 Initialisation

Let $v_i^0 = 0 \forall i \in S$ and let $n = 1$

PARALLEL VALUE ITERATION

Step 2 Initial calculation

Calculate $v_i^n = (Ev^{n-1})_i \forall i \in S_{r,1}$ and let $b = 1$

Step 3 Overlap remaining calculation with communication of new estimates

While $b < f$

Calculate $v_i^n = (Ev^{n-1})_i \forall i \in S_{r,b+1}$, send $v_i^n \forall i \in S_{r,b}$ to all other processors and read $v_i^n \forall i \in S_{t,b}, \forall t \neq r$ concurrently and then increase b by 1

Step 4 Communicate the remaining new estimates

Send $v_i^n \forall i \in S_{r,f}$ to all other processors and read $v_i^n \forall i \in S_{t,f}, \forall t \neq r$

Step 5 Initiate the bound calculation

Calculate $U_r^n = \max_{i \in S_r} \{v_i^n - v_i^{n-1}\}$ and $L_r^n = \min_{i \in S_r} \{v_i^n - v_i^{n-1}\}$

Step 6 Complete the bound calculation

Perform a global maximisation and minimisation to calculate U^n and L^n from the values $U_1^n, \dots, U_p^n$ and $L_1^n, \dots, L_p^n$ respectively

Step 7 Check for convergence

If $\frac{\beta}{1-\beta}(U^n - L^n) > \epsilon$, increase n by 1 and repeat from step 2

Figure 5.19(c) shows how the phased pipeline algorithm might be scheduled on a processor for the case in which calculation and communication can be overlapped without penalty. For processor r , C_t represents updating the states in $S_{r,t}$ and B_t represents broadcasting the new updates for states in $S_{r,t}$ and receiving the latest updates for states in $S_{t,t} \forall t \neq r$. In the time required to update all the states assigned to a processor, almost all of the communication for that iteration is complete. In terms of time per iteration, this algorithm is far superior to the algorithm in figure 5.19(a) and approaches the time per iteration achieved by the asynchronous algorithm in figure 5.19(b) as the number of phases increases.

In order to preserve the convergence properties of serial PJ & PB, the new estimates received by a processor during an iteration are not used in that processor's calculation until the following iteration. As a result implementing this algorithm requires some

PARALLEL VALUE ITERATION

additional storage, but the reward and transition data can still be distributed as before. Any action elimination test can be added to the calculation performed at every iteration.

If the phased pipeline algorithm is applied to generalised GS and a processor uses the most recent estimates for states local to that processor and estimates from the previous iteration for all other states, then the resulting algorithm will have the same convergence properties as GS1, but will have greatly reduced communication overheads. However since the convergence of a parallel Gauss-Seidel algorithm is generally improved by the use of a greater number of up-to-date estimates, it is sensible for a processor to use the new estimates in its calculation as soon as they are received. This also eliminates the need for extra storage. Since the pattern of communication, and hence the availability of recent estimates on each processor, varies from iteration to iteration, the resulting algorithm (which will be referred to as GS3) is equivalent to some instance of variable generalised GS.

GS3 & $L_\infty B$: the phased pipeline algorithm applied to GS

Let processor r perform the following steps sequentially.

Step 1 Initialisation

Let $v_i^0 = 0 \forall i \in S$ and let $n = 1$

Step 2 Initial calculation

Let $v^n = v^{n-1}$, calculate $v_i^n = (Fv^n)_i \forall i \in S_{r,1}$ in the order of increasing state number and let $b = 1$

Step 3 Overlap remaining calculation with communication of new estimates

While $b < f$

Calculate $v_i^n = (Fv^n)_i \forall i \in S_{r,b+1}$ sequentially in the order of increasing state, send $v_i^n \forall i \in S_{r,b}$ to all other processors and read $v_i^n \forall i \in S_{t,b}, \forall t \neq r$ concurrently and then increase b by 1

Step 4 Communicate the remaining new estimates

Send $v_i^n \forall i \in S_{r,f}$ to all other processors and read $v_i^n \forall i \in S_{t,f}, \forall t \neq r$

Step 5 Initiate the bound calculation

Calculate $N_r^n = \max_{i \in S_r} |v_i^n - v_i^{n-1}|$

PARALLEL VALUE ITERATION

Step 6 Complete the bound calculation

Perform a global maximisation to calculate $\|v^n - v^{n-1}\|_\infty$ from the values $N_1^n, \dots, N_P^n$

Step 7 Check for convergence

If $\frac{\beta}{1-\beta} \|v^n - v^{n-1}\|_\infty > \frac{\epsilon}{2}$, increase n by 1 and repeat from step 2

During each phase when a processor updates a state, it uses the most recent data for states local to that processor and for states in some subset of the set of all states updated on other processors during previous phases. Therefore the critical area for GS3 is some subset of area 2 (see figure 5.17). Figure 5.20 shows one possible candidate for this area. The critical area in off-diagonal partially filled squares is determined by the scheduling of the communication and will vary from iteration to iteration. It is apparent from the figures that in terms of convergence GS3 is somewhere between GS1 and GS2. As the number of phases increases the convergence of GS3 approaches that of GS2 while, for the case in which calculation and communication can be overlapped

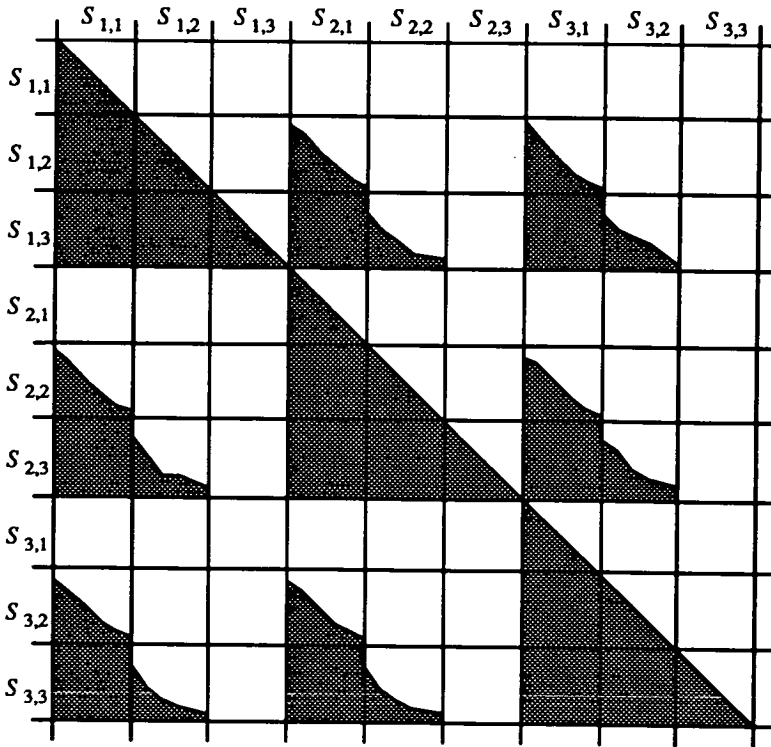


Figure 5.20: An example of the critical area for GS3 with $P = 3$ and $f = 3$.

PARALLEL VALUE ITERATION

without penalty, the time per update approaches that of the asynchronous scheme in figure 5.19(b).

It is possible to obtain a further reduction in the communication overheads by overlapping the final communication phase at one iteration with the first calculation phase at the next iteration. However the resulting algorithm is asynchronous and will exhibit slower convergence. For PJ this modification will destroy the stochastic² property of the iteration matrices and, as observed in section 5.5, this leads to an increase in the number of iterations. The net effect is likely to be a significant increase in solution time. However for GS the results of section 5.5 show that the number of additional iterations required is not as significant and so the benefits of lower communication overheads may result in a reduction in solution time. The following algorithm will be used to determine the potential of this modification.

Asynchronous GS3 & $L_\infty B$

In the description of this algorithm arithmetic modulo f is used on the second subscript of S .

Let processor r perform the following steps sequentially.

Step 1 Initialisation

Let $v_i = 0 \forall i \in S$, let $n = 0$ and calculate $v_i = (Fv)_i \forall i \in S_{r,f}$ sequentially in the order of increasing state

Step 2 Initialisation for the asynchronous calculation

Let $d = 1$ and let $b = 1$

Step 3 Perform a multinode broadcast and a set of updates asynchronously

While $b \leq f$

Calculate $v_i = (Fv)_i \forall i \in S_{r,b}$ sequentially in the order of increasing state, send $v_i \forall i \in S_{r,b-1}$ to all other processors and read $v_i \forall i \in S_{t,b-1}, \forall t \neq r$ concurrently and then increase b by 1

Step 4 Interrupt the asynchronous calculation to check for convergence

If $d < D - 2$, increase d by 1 and repeat from step 3

PARALLEL VALUE ITERATION

Step 5 Synchronise processors

Let $v^n = v$, increase n by 1 and perform steps 2, 3 and 4 of GS3 & $L_\infty B$

Step 6 Perform one synchronous iteration

Increase n by 1 and perform steps 2 and 3 of GS3 & $L_\infty B$

Step 7 Perform a bound calculation

Perform steps 5 and 6 of GS3 & $L_\infty B$

Step 8 Check for convergence

If $\frac{\beta}{1-\beta} \|v^n - v^{n-1}\|_\infty > \frac{\epsilon}{2}$, let $v = v^n$, increase n by 1 and repeat from step 2

5.7 Modelling overlapped calculation and communication on a transputer

In this section a model for overlapped calculation and communication on a T800 transputer is developed. This model is used in section 5.8 where a detailed analysis of the performance of the phased pipeline algorithm can be found. The basis for the model is the results of two experiments in which a transputer performs some calculation and communicates using all eight of its channels simultaneously. Timings are made of the transputer performing the calculation only, the communication only and both tasks concurrently. The calculation involves updating some states in a Markov decision process. The communication involves repeated multinode broadcasting on a ring topology. Let K be the number of multinode broadcasts performed and let ℓ be the number of values in each message broadcast.

In the first experiment the number of values in the messages broadcast is varied, while the number of multinode broadcasts performed and the number of states updated are held constant. The results are shown in figure 5.21 and can be interpreted as follows.

Initially, when messages are very short, little is gained by overlapping because communication time is dominated by channel set-ups which require the central processing unit (cpu), and consequently block the calculation.

As message length increases, the time during which all channels are transmitting increases, and hence the potential for overlapping increases. However the fact that

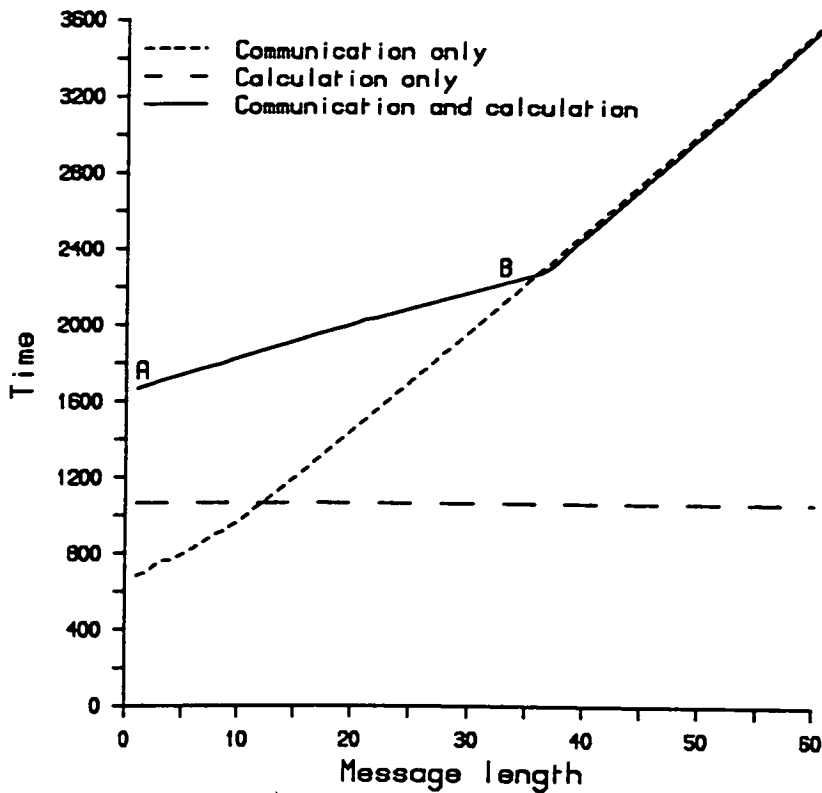


Figure 5.21: A plot of the time required (in ms) for overlapped calculation and communication against message length.

between the points A and B the calculation and communication curve is increasing indicates that not only is the calculation blocked during channel set-ups, but it is also slowed down during transmission. This effect is due to the conflict between link buffers and the cpu for memory access. If the cpu is denied memory access then the calculation will come to a halt. The amount of interference depends upon the rate at which the calculation generates memory accesses and the number of channels in use.

For long messages the time during which all channels are transmitting is large enough to allow all the calculation to be overlapped, and the communication only and the calculation and communication curves coincide. When a communication process requires memory access, it may have to wait while another communication process or the cpu accesses the memory but, since the link buffer allows communication to continue, this conflict does not lead to an increase in the time required for the communication.

PARALLEL VALUE ITERATION

Assuming that the conditions required for (5.3) are satisfied and P is odd, figure 5.21 suggests the following model for elapsed time.

$\max\{\text{communication time, slowed calculation time}\}$

$$= \max \left\{ K \left((P+5)c + \frac{(P-1)\ell}{4}\rho \right), \right. \\ \left. \tau + 4K(P-1)b + \left(K \left((P+5)c + \frac{(P-1)\ell}{4}\rho \right) - 4K(P-1)b \right) (1-\eta) \right\} \quad (5.5)$$

where c is the set-up time for a channel,

ρ is the transmission time for a 64bit value when the links are being used bidirectionally,

b is the time for which a channel set-up blocks the calculation,

η is the ratio of the amount of calculation performed in unit time when overlapped with 8 channel transmission to the amount performed in unit time with no transmission,

and τ is the calculation time.

The second term only represents slowed calculation time when slowed calculation time is at least as great as communication time. When calculation can be completely overlapped with communication then the second time is an upper bound on the slowed calculation time but, as the following argument shows, it never exceeds the communication time.

The time during which channel set-ups do not block calculation is given by

$$K \left((P+5)c + \frac{(P-1)\ell}{4}\rho \right) - 4K(P-1)b$$

and the amount of calculation which can be performed during this time is given by

$$\left(K \left((P+5)c + \frac{(P-1)\ell}{4}\rho \right) - 4K(P-1)b \right) \eta.$$

So when calculation can be completely overlapped with communication

$$\tau < \left(K \left((P+5)c + \frac{(P-1)\ell}{4}\rho \right) - 4K(P-1)b \right) \eta$$

$$\Leftrightarrow \tau + 4K(P-1)b + \left(K \left((P+5)c + \frac{(P-1)\ell}{4}\rho \right) - 4K(P-1)b \right) (1-\eta)$$

$$< K \left((P+5)c + \frac{(P-1)\ell}{4}\rho \right).$$

PARALLEL VALUE ITERATION

Applying (5.3) to the straight line section of the communication only curve yields least squares estimates of $18.93\mu s$ for c and $7.59\mu s$ for ρ . These are consistent with the estimates in table 5.1. Using (5.5) least squares estimates of the gradient and intercept of the calculation and communication curve between A and B can be obtained which, when combined with the above estimates of c and ρ , give estimates of $12.32\mu s$ for b and 0.67 for η . The estimate of b implies that the calculation is not blocked for the full duration of a channel set-up. This is because calculation can proceed during some of the delays introduced in the scheduling of communication tasks (see section 5.3).

In the second experiment K , the number of multinode broadcasts performed, and ℓ , the number of values in the messages broadcast, are varied but $K\ell$ and the number of states updated are held constant. The results are shown in figure 5.22 and they yield

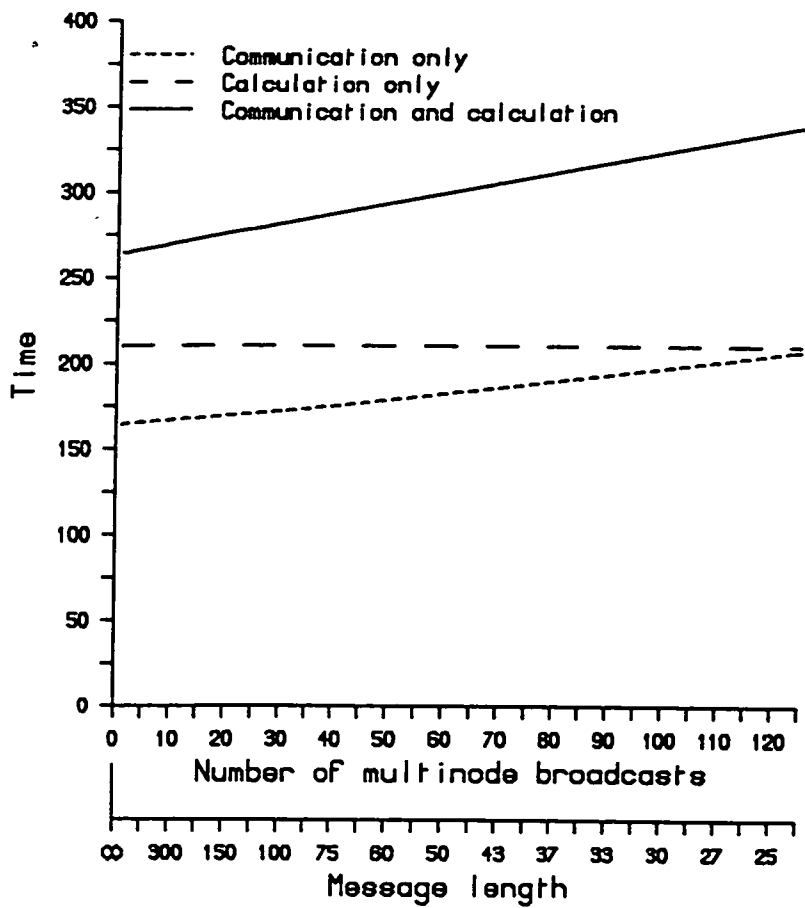


Figure 5.22: A plot of the time required (in ms) for overlapped calculation and communication against the number of multinode broadcasts.

PARALLEL VALUE ITERATION

the following estimates.

The communication only curve gives least squares estimates of $17.11\mu s$ for c and $7.76\mu s$ for ρ . Rewriting the communication time in (5.3) as $K(P + 5)c + \text{constant}$ shows that the gradient of this curve as K varies gives an estimate of c . Similarly from (5.5) the calculation and communication curve has a gradient as K varies of $(P + 5)(1 - \eta)c + 4(P - 1)b\eta$ which in figure 5.22 is a steeper gradient than the $(P + 5)c$ of the communication only curve. Taking c to be $17.11\mu s$ and ρ to be $7.76\mu s$ and using the gradient of the calculation and communication curve yields least square estimates of $12.97\mu s$ for b and 0.67 for η . These are in good agreement with the estimates obtained from the first experiment.

Further experiments support the general model for overlapped calculation and communication on a transputer described in (5.5) and confirm that the values of c , b and η depend upon the nature of the communication and calculation tasks performed.

5.8 Implementing the phased pipeline algorithm

The model developed in section 5.7 can now be applied to the phased pipeline algorithm described in section 5.6. A multinode broadcast on a torus requires fewer channel set-ups than on a ring, so there will be less interruption of the calculation during overlapping, and consequently increased efficiency. However for the reasons given in section 5.3.4, this section concentrates on a ring using the procedures BROADR and GMAX described in section 5.3. It would be an easy task to rework the analysis for a torus or any other connection topology.

Assume that the number of transputers used is odd, all the sets in the partition of the state space contain the same number of states and that the load balance is perfect. The PJ and GS updates require approximately the same number of operations so, using (5.3) and (5.5), the time required for an iteration of the phased pipeline algorithm applied to PJ or GS1 using f phases and a ring of P transputers is given by

$$T(P, f) = \frac{\tau}{Pf} + (f-1) \max \left\{ (P+5)c + \frac{(P-1)m}{4Pf} \rho, \right. \\ \left. \frac{\tau}{Pf} + 4(P-1)b + \left((P+5)c + \frac{(P-1)m}{4Pf} \rho - 4(P-1)b \right) (1-\eta) \right\} + \\ (P+5)c + \frac{(P-1)m}{4Pf} \rho \quad (5.6)$$

where m is the number of states,

τ is time required to update all the states once using one transputer
and c , ρ , b and η are as defined in (5.5).

(5.6) does not allow for the bound calculation because, as it is not necessary to calculate bounds at every iteration, the time required to perform a bound calculation need never be a significant part of the solution time. However if required the expressions derived in section 5.3.3 can be used.

Due to the overheads involved with overlapped calculation and communication, the model predicts that the number of phases which minimise the time per iteration for all six of the small test problems is 1 and this has been found to be correct. Consequently to observe the benefits of the phased pipeline algorithm it is necessary to consider larger problems.

Figure 5.23 shows a contour plot of the speed-up in time per iteration (which is equal to $\frac{\tau}{T(P,f)}$) for the phased pipeline algorithm on a 60000 state, 3 action problem (GEN3 from table 3.5) when action elimination is not used. Since the convergence properties of serial PJ are preserved when the phased pipeline algorithm is applied to PJ, this quantity is equal to the speed up for PJ & PB. The contours join points of equal speed-up. The plane is divided into two regions — one in which communication time per phase is greater than the slowed calculation time per phase and another in which the opposite is true. The point at which speed-up is maximised is marked by a circle. For the example shown the optimal point lies on the boundary between the two regions, but this is not always the case. The cross marks the point at which speed-up is maximised when the calculation is not phased. Compared with the phased pipeline algorithm, twice as many transputers are required to obtain a speed-up which is 12%

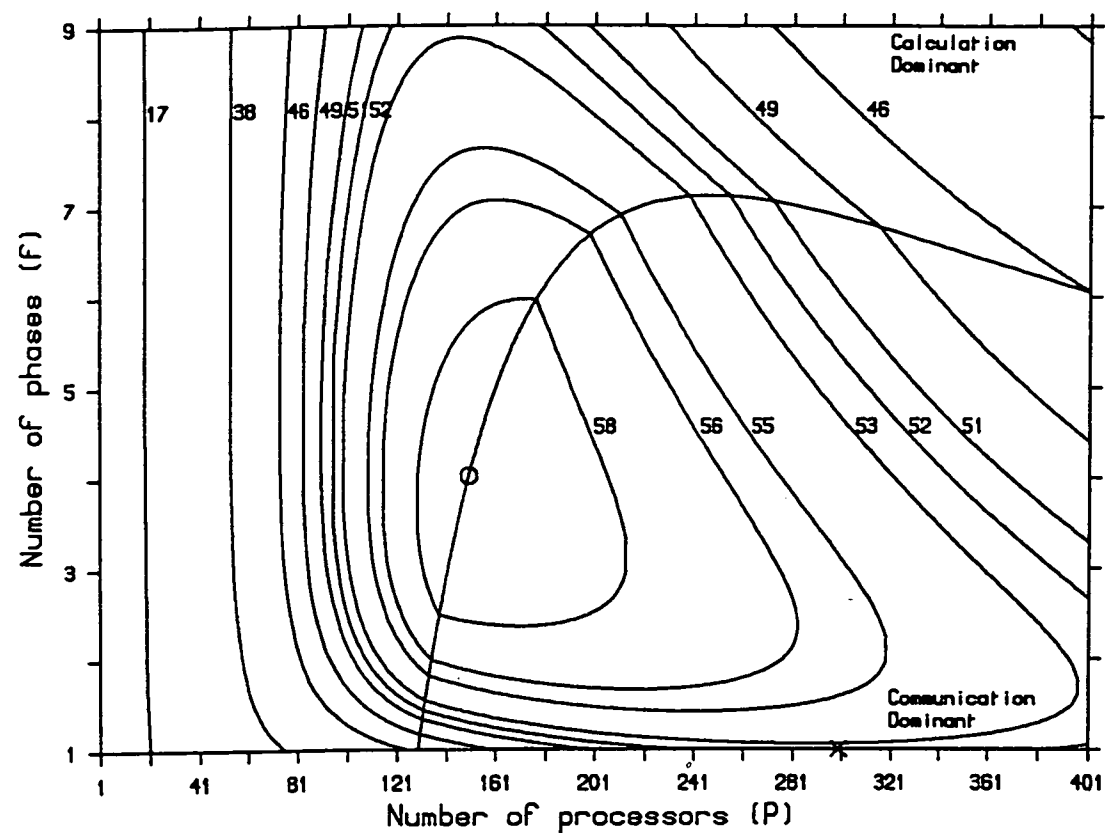


Figure 5.23: Contour plot showing speed-up for the phased pipeline algorithm.

less than the optimal. As the number of transputers used changes from the optimal number in figure 5.23, the drop-off in speed-up is more severe for a reduction than for an increase. Hence it is preferable to err on the side of too many transputers.

Table 5.7 compares the model's predictions of the number of transputers P^* and the number of phases f^* which maximise speed-up with the performance obtained on a Meiko computing surface for two problems with different properties (GEN3 and GEN4 from table 3.5). The number of transputers and phases actually used are denoted by P and f in the table. A series of 30000 state problems which differ only in the number of actions per state (GEN5 from table 3.5) were solved using a ring of 111 transputers and 8 calculation phases. Table 5.8 compares the actual speed-up obtained with that predicted by the model.

In tables 5.7 and 5.8 there is a good agreement between the model's predictions and the performances actually observed. In the worst cases the predicted times are around

Table 5.7: Actual versus predicted optimal speed-up

Number of states	15000	60000
Number of actions per state	1	3
P^*, f^*	51.11, 3.25	150.03, 4.03
Predicted speed-up	21×	60×
P, f	49, 4	121, 4
Speed-up obtained	18×	58×

Table 5.8: Actual versus predicted speed-up for some 30000 state problems

Number of actions per state	1	2	3	4	5	6	7	8	9	10
Predicted speed-up	18×	32×	42×	50×	56×	61×	65×	69×	72×	74×
Actual speed-up	17×	30×	38×	45×	50×	54×	58×	61×	64×	69×

10% less than the actual times. The partition chosen satisfies the assumption of load balancing required by the model, but does not satisfy the condition of equal number of states per transputer because there is a trend in the work per state (see section 3.3.3). Therefore the model underestimates communication time. However the results show that the model can adequately predict the performance of the phased pipeline algorithm and so can be used to maximise speed-up or efficiency.

Table 5.9 uses the model to compare the performance of the phased pipeline algorithm and an algorithm which does not overlap calculation and communication on larger versions of the 60000 state problem from table 5.7. The results demonstrate that the phased pipeline algorithm can achieve greater speed-ups using a fraction of the transputers. In the case of the 6000K state problem, the phased pipeline algorithm achieves a slightly greater speed-up using about $\frac{1}{20}$ of the transputers.

The results of sections 4.2 and 5.4.3 have shown that action elimination procedures can lead to significant reductions in solution times in serial and parallel value iteration algorithms. With action elimination the calculation time per iteration is no longer constant, but decreases as the algorithms progress. Consequently with the phased pipeline approach the ideal number of transputers and phases to use will change as the algorithms progress. However the original regime will remain within the communication

PARALLEL VALUE ITERATION

Table 5.9: Predicted performance on large scale problems

Number of states	60K	600K	6000K
P^*, f^*	150.03, 4.03	132.35, 15.63	127.08, 51.73
Optimal Speed-up	60×	75×	79×
P^-	299.76	947.94	2997.65
Speed-up	53×	69×	76×

P^* and f^* are the model's estimates of the optimal number of transputers and phases to use in the phased pipeline algorithm.

P^- is the optimal number of transputers to use when the calculation is performed in one phase.

dominant region of the relevant contour plot and these regions are all reasonably flat. Hence the degradation from the optimum will be slight. In problems with a large number of actions per state it might be worthwhile estimating the ideal number of transputers and phases to use after most actions have been eliminated and switching to this regime at a suitable point in the calculation. However the improvements which can be obtained by adopting such a strategy appear to be fairly limited.

The results above show that the average time per iteration can be reduced by applying the phased pipeline algorithm to PJ and GS1, but for GS the rate of convergence of parallel algorithms is also an important consideration. In table 5.10 the six GEN6 problems (see section 3.5 and table 3.6) are used to investigate the convergence properties of the two GS algorithms based on the phased pipeline algorithm — GS3 & $L_\infty B$ and asynchronous GS3 & $L_\infty B$.

In the first three problems 90% of the weight in the transition matrix lies in a band just below the diagonal and so is in the critical areas for GS1 and GS3. However the additional weight in the critical area for GS3 leads to a 33% reduction in the number of iterations required. In GEN6D 90% of the weight in the transition matrix lies in a band well below the diagonal and is therefore not in the critical area for GS1. Consequently the convergence of GS1 in this case is even worse than for the completely random structure of GEN6F. However much of the weight does lie in the critical area for GS3 and so the convergence of GS3 is far better. In GEN6E the positive transitions are distributed at random below the diagonal and again the convergence of GS1 is slow.

Table 5.10: The effect of the phased pipeline algorithm on the convergence of GS

Problem	GEN6A	GEN6B	GEN6C	GEN6D	GEN6E	GEN6F
GS1 & $L_{\infty}B$						
N_{iter}	378	374	370	3038	1628	3028
GS3 & $L_{\infty}B$						
N_{iter}	254	248	246	733	1366	2019
Asynchronous GS3 & $L_{\infty}B$ with $D = 25$						
N_{upd}	275	275	275	775	1375	2075

In all six cases a ring of 111 transputers and 8 calculation phases were used.

For the two problems with random structure the convergence of GS3 is considerably better because the critical area for this problem is larger.

Although the average time per update is less in asynchronous GS3 & $L_{\infty}B$, the increase in the number of updates performed results in an increase in solution time for every problem except GEN6E. However in no case is the change in solution time greater than 5%.

In conclusion when used to solve large scale Markov decision processes, value iteration algorithms based on the phased pipeline algorithm have low communication overheads and good convergence properties. With PJ the convergence properties of the serial algorithm are preserved and with GS the rate of convergence is generally faster than in a non-overlapping parallel algorithm.

Footnotes

- 1. A parallel algorithm is said to deadlock when one or more processors attempt communications which will never be successfully completed.
- 2. To be precise, the iteration matrices in PJ are not stochastic, but are equal to a scalar multiple of a stochastic matrix, see section 2.2.

Chapter 6

Reward Revision

6.1 Introduction

The reward revision algorithm for infinite horizon time invariant discounted Markov decision processes was first proposed by White *et al* [54]. The objective is to construct a series of Markov decision processes which are computationally easier than the original problem but have the similar optimal values. This is achieved by modifying the transition structure of the original problem, so that a reduction in the number of transitions with positive probabilities or an increase in the rate of convergence is achieved, and revising the discount factor and the immediate rewards to compensate for this modification. Since reward revision is based on value iteration methods, the knowledge built-up in chapter 5 can be employed in the development of parallel reward revision algorithms.

White *et al* [54] concentrate on the PJ & PB value iteration method, although it is noted that it would be straightforward to add an action elimination test to the procedure. Since the results of chapters 4 and 5 show that action elimination tests and Gauss-Seidel based algorithms are often successful, this study considers a broad class of value iteration methods in the evaluation of parallel reward revision.

The discount factor, the transition probabilities and the immediate rewards for the revised problems are denoted by $\bar{\beta}$, $\bar{p}_{i,j}^k$ and $\bar{r}_i^{n,k}$ respectively.

The names given to the reward revision algorithms described in this chapter have the form A1:A2 where A1 refers to the value iteration algorithm used to revise the rewards and A2 refers to the value iteration algorithm used to iterate on the revised problems.

6.2 Serial reward revision algorithms

The algorithms developed in this chapter are based on the serial reward revision algorithm proposed in [54]. The algorithm below is essentially the same as the one proposed in [54], the only difference being the stopping rule used.

PJ:PJ & PB

Step 1 Modify the discount factor and the transition structure

Choose $\bar{\beta}$ and $\bar{p}_{i,j}^k$ to satisfy $0 \leq \bar{\beta} < 1$, $0 \leq \bar{p}_{i,j}^k \leq 1 \forall k \in K_i, \forall i, j \in S$ and $\sum_{j \in S} \bar{p}_{i,j}^k = 1 \forall k \in K_i, \forall i \in S$

Step 2 Initialisation

Choose D such that $2 \leq D < \infty$, let $v^{0,0} = 0$ and let $n = 0$.

Step 3 Revise rewards

Calculate $v_i^{1,n} = \max_{k \in K_i} \{R_i^k\} \forall i \in S$ and $\bar{r}_i^{n,k} = R_i^k - \bar{\beta} \sum_{j \in S} \bar{p}_{i,j}^k v_j^{0,n} \forall k \in K_i, \forall i \in S$

where $R_i^k = r_i^k + \beta \sum_{j \in S} p_{i,j}^k v_j^{0,n}$

Step 4 Check for convergence

If $\frac{\beta}{1-\beta} \left(\max_{i \in S} \{v_i^{1,n} - v_i^{0,n}\} - \min_{i \in S} \{v_i^{1,n} - v_i^{0,n}\} \right) \leq \epsilon$ then (1.3) is satisfied

Step 5 Iterate on the current revised problem

Let $d = 2$

While $d \leq D$

Calculate $v_i^{d,n} = \max_{k \in K_i} \left\{ \bar{r}_i^{n,k} + \bar{\beta} \sum_{j \in S} \bar{p}_{i,j}^k v_j^{d-1,n} \right\} \forall i \in S$ and then increase d by 1

Step 6 Initialisation for another iteration

Increase n by 1, let $v^{0,n} = v^{D,n-1}$ and repeat from step 3

The transition structure and the discount factor are only modified once, but the rewards are revised periodically resulting in a series of problems which differ only in their immediate rewards.

REWARD REVISION

Since step 3 of PJ:PJ involves a pre-Jacobi iteration on the original problem (observe that $v^{1,n} = Ev^{0,n}$ where E is as in (2.1)), the PB can be used to terminate the iterative procedure in step 4 and MAE can be applied at the beginning of step 5. It is efficient to combine the application of MAE with the first iteration performed on the revised problem during step 5. PAE2 is not appropriate because two consecutive iterations on the original problem are never performed.

Step 5 of PJ:PJ involves $D - 1$ pre-Jacobi iterations on a revised problem. White *et al* [54] propose bounds which relate $v^{D,n}$ to v^* and this enables a stopping rule and an action elimination test to be applied at the end of this step. However these bounds were not used as a stopping rule in [54] and, since chapter 4 concludes that the less frequent application of stopping rules and action elimination tests often reduces solution time, they will not be considered in this study.

White *et al* [54] show that if the method used to modify the discount factor and the transition structure has the property that

$$\max_{i \in S, k \in K_i} \left\{ \sum_{j \in S} |\beta p_{i,j}^k - \bar{\beta} \bar{p}_{i,j}^k| \right\} = \beta - \bar{\beta} \quad (6.1)$$

then

$$\|v^{0,n} - v^*\|_\infty \leq \beta(D) \|v^{0,n-1} - v^*\|_\infty \text{ where } \beta(D) = 1 - \frac{(1 - \bar{\beta}^D)(1 - \beta)}{1 - \bar{\beta}} \quad (6.2)$$

Since $\beta(D) \leq \beta$ for $D \geq 1$, the reward revision algorithm converges.

White *et al* [54] suggest the following procedure for modifying the discount factor and the transition structure, because it guarantees that the condition in (6.1) is satisfied. For each $i \in S$ and $k \in K_i$ choose $S_{i,k} \subset S$ such that $\sum_{j \in S_{i,k}} p_{i,j}^k \neq 0$ and let

$$\bar{p}_{i,j}^k = \begin{cases} 0 & \text{if } j \notin S_{i,k} \\ \frac{p_{i,j}^k}{\sum_{l \in S_{i,k}} p_{i,l}^k} & \text{if } j \in S_{i,k} \end{cases} \text{ and } \bar{\beta} = \beta \min_{i \in S, k \in K_i} \left\{ \sum_{j \in S_{i,k}} p_{i,j}^k \right\} \quad (6.3)$$

White *et al* [54] do not suggest any method for selecting the sets $S_{i,k}$, but this area is identified as being worthy of further research.

The results of chapters 4 and 5 have shown that for slowly mixing problems which have a known WLT structure GS performs far better than PJ. In such cases it is

REWARD REVISION

unlikely that any algorithm based on PJ will be competitive with GS. Therefore ways of incorporating GS within the framework of reward revision are worth exploring.

One method would be to use GS to iterate on the revised problems resulting in a hybrid scheme (hybrid iterative solution algorithms for Markov decision processes were proposed by Puterman and Shin [44]). This is the approach adopted in the following algorithm.

PJ:GS & PB

Steps 1 to 4 are identical to steps 1 to 4 of PJ:PJ & PB

Step 5 Iterate on the current revised problem

Let $d = 2$

While $d \leq D$

$$\text{Calculate } v_i^{d,n} = \max_{k \in K_i} \left\{ \frac{\bar{r}_i^{n,k} + \bar{\beta} \sum_{j < i} \bar{p}_{i,j}^k v_j^{d,n} + \bar{\beta} \sum_{j > i} \bar{p}_{i,j}^k v_j^{d-1,n}}{1 - \bar{\beta} \bar{p}_{i,i}^k} \right\} \quad \forall i \in S \text{ and}$$

then increase d by 1

Step 6 Initialisation for another iteration

Increase n by 1, let $v^{0,n} = v^{D,n-1}$ and repeat from step 3

As with PJ:PJ & PB, MAE can be applied at the beginning of step 5. A more radical method of incorporating GS would be to use GS to revise the rewards as well as to iterate on the revised problems. This is the approach adopted in the following algorithm.

GS:GS & L_∞ B

Steps 1 and 2 are identical to steps 1 and 2 of PJ:PJ & PB

Step 3 Revise rewards

$$\text{Calculate } v_i^{1,n} = \max_{k \in K_i} \{R_i^k\} \quad \forall i \in S \text{ and } \bar{r}_i^{n,k} = R_i^k - \frac{\bar{\beta} \sum_{j < i} \bar{p}_{i,j}^k v_j^{1,n} - \bar{\beta} \sum_{j > i} \bar{p}_{i,j}^k v_j^{0,n}}{1 - \bar{\beta} \bar{p}_{i,i}^k}$$

$$\forall k \in K_i, \forall i \in S \text{ where } R_i^k = \frac{r_i^k + \beta \sum_{j < i} p_{i,j}^k v_j^{1,n} - \beta \sum_{j > i} p_{i,j}^k v_j^{0,n}}{1 - \beta p_{i,i}^k}$$

REWARD REVISION

Step 4 Check for convergence

If $\frac{\beta}{1-\beta} \|v^{1,n} - v^{0,n}\|_{\infty} \leq \frac{\epsilon}{2}$ then (1.3) is satisfied

Step 5 Iterate on the current revised problem

Let $d = 2$

While $d \leq D$

Calculate $v_i^{d,n} = \max_{k \in K_i} \left\{ \bar{r}_i^{n,k} + \frac{\bar{\beta} \sum_{j < i} \bar{p}_{i,j}^k v_j^{d,n} + \bar{\beta} \sum_{j > i} \bar{p}_{i,j}^k v_j^{d-1,n}}{1 - \bar{\beta} \bar{p}_{i,i}^k} \right\} \quad \forall i \in S$ and

then increase d by 1

Step 6 Initialisation for another iteration

Increase n by 1, let $v^{0,n} = v^{D,n-1}$ and repeat from step 3

The PB are not used in GS:GS because the results of chapter 4 show that the $L_{\infty}B$ are generally more effective than the PB for GS. $L_{\infty}AE1$ can be applied at the beginning of step 5, but $L_{\infty}AE2$ is not valid.

The proof of proposition 1 of [54] is valid for PJ:GS and GS:GS so, if the condition in (6.1) is satisfied, PJ:GS & PB and GS:GS & $L_{\infty}B$ converge.

Modifying the transition structure to promote the rate of convergence of the reward revision algorithms will involve considerable overheads because, since the policy used changes from iteration to iteration and the structures of transition matrices for different policies can differ greatly, the transition structure must be modified to suit each policy selected. Consequently this technique has not been addressed explicitly in this study. As with the reordering algorithms of section 2.4, this technique is likely to be more useful within the framework of a policy iteration approach.

6.3 Computational aspects of serial reward revision

Let A be the total number of actions in the original problem (and hence in the revised problems also).

Let t_p be the number of transitions which have non-zero probability in the original problem and the revised problems.

REWARD REVISION

Let t_z be the number of transitions which have non-zero probability in the original problem, but have zero probability in the revised problems.

Let t_a be the number of transitions which have zero probability in the original problem, but have non-zero probability in the revised problems.

Note that with the method of modifying the discount factor and the transition structure described in (6.3), $t_a = 0$.

An iteration of PJ on the original problem requires $A - m + 2(t_z + t_p)$ operations and the requirement for GS is similar.

Step 3 of PJ:PJ and PJ:GS requires $A - m + 2(t_z + 2t_p + t_a)$ operations and the requirement for GS:GS is similar.

Step 5 of PJ:PJ requires $(D-1)(A - m + 2(t_p + t_a))$ operations and the requirement for PJ:GS and GS:GS is similar.

Using β and $\beta(D)$ as approximations to the rates of convergence of PJ and PJ:PJ, White *et al* [54] derive the following method for estimating the optimal value of D .

Let n_{PJ} and n_{RR} denote the number of iterations required by PJ & PB and PJ:PJ & PB respectively, then $n_{RR} \simeq \frac{\log \beta}{\log \beta(D)} n_{PJ}$. Therefore for a given method for modifying the discount factor and the transition structure, the ratio of the solution time for PJ:PJ to the solution time for PJ is approximately equal to

$$\frac{\log \beta}{\log \beta(D)} \left(\frac{D(A - m + 2(t_p + t_a)) + 2(t_z + t_p)}{A - m + 2(t_z + t_p)} \right)$$

and the value of D used can be chosen to minimise this expression. Since A , t_p , t_z and t_a can easily be calculated when the transition structure is modified, this minimisation involves little effort.

In this study the PB are used with PJ and PJ:PJ, so the rates of convergence in these algorithms will generally be faster than the approximations used above. Consequently the estimator may not be successful. If the structure of the problem is suited to GS then the rates of convergence of PJ:GS and GS:GS will not be determined by $\beta(D)$. Hence the estimator above is not appropriate for PJ:GS or GS:GS. When an action elimination test is used the values of A , t_p , t_z and t_a will vary from iteration to iteration in a way which cannot be accurately predicted. However in cases where these variations have

a significant effect on the performance of the algorithms, the estimate of the optimal value of D can be recalculated after each application of the action elimination test.

It can be shown that when $\bar{\beta}$ satisfies $0 \leq \bar{\beta} \leq \beta$ and $D \geq 1$, $\beta(D)$ is an increasing function of $\bar{\beta}$ and $\beta \leq \beta(D) \leq \beta^D$. Therefore PJ:PJ will be successful when t_z is much greater than $t_p + t_a$, so that iterations on the revised problem require a trivial amount of work; and $\bar{\beta}$ is close to β , so that $\frac{\log \beta}{\log \beta(D)}$ is close to $\frac{1}{D}$. However increasing t_z involves increasing the left hand side of (6.1) and hence decreasing $\bar{\beta}$, so for a convergent method these are conflicting goals.

6.4 Implementing serial reward revision

This section compares the performance of the three reward revision algorithms described in section 6.2 on a single transputer using the six small test problems from table 3.3. The transition structure is modified by discarding all transition probabilities less than a given threshold λ . This can be achieved by letting $S_{i,k} = \{j \in S : p_{i,j}^k \geq \lambda\}$ in (6.3) and has the effect of reducing the number of transitions with positive probabilities. In general this approach will not promote the rate of convergence of GS or PJ in the revised problems. However when a WLT ordering of the state space is used, since probabilities above the diagonal of the transition matrices are small and so are likely to be discarded, the convergence of GS in the revised problems will be rapid. The revised probabilities are well defined provided λ is small enough to ensure that $\forall i \in S$ and $\forall k \in K_i$, $\exists j \in S$ such that $p_{i,j}^k \geq \lambda$. A suitable choice for λ can be determined from the probability distributions of the transitions.

The programs used have been designed specifically for serial machines. The times presented in the result tables are in seconds and were obtained using a T800-20MHz transputer. As before the solution time does not include the time required to obtain the data initially (see section 4.2). An iteration of a reward revision algorithm consists of steps 3, 4 and 5 of the algorithm and the number of iterations required to solve a problem is taken to be the value of n when the convergence criterion in step 4 is first satisfied. Note that at this point $n + 1$ iterations have been performed on the original problem and $n(D - 1)$ iterations have been performed on revised problems. The results

REWARD REVISION

for PJ:PJ & PB compare the value of D predicted by the method proposed in section 6.3 with the optimal value of D , which is determined by trial and error. The tables corresponding to the two GS based algorithms record results for the optimal value of D only, because there is no known method for predicting the rate of convergence in these cases. See page xiv for details of the notation used in the tables.

HOWARD

In HOWARD the number of possible transitions resulting from an action is never more than two. Hence an iteration on a revised problem involves at least half as many operations as an iteration on the original problem. Further for many of the actions which result in two possible transitions, both probabilities are significant. This means that for a convergent method it is not possible to reduce the workload significantly and have $\bar{\beta}$ close to β . Therefore reward revision is never successful for HOWARD.

INVENT1

In INVENT1 there are usually twelve possible transitions resulting from an action and the sum of the probabilities of the six least likely of these is only 0.042. Hence it is possible to reduce the workload by approximately one half and keep the ratio of $\bar{\beta}$ to β as high as 0.958. Therefore the transition structure of INVENT1 is suited to reward revision.

Table 6.1 gives details of the performance of PJ:PJ & PB with INVENT1 for a range of thresholds. With $\lambda = 0.002$ three of the twelve possible transitions resulting from most actions are discarded. An additional transition is discarded with each increment of λ so that with $\lambda = 0.235$ only one of the twelve possible transitions remains.

The results verify that the value of $\bar{\beta}$ falls as the number of operations required in an iteration on the revised problem falls. As $\bar{\beta}$ falls the solutions of the revised problems become poorer approximations to v^* and the rate of convergence in the revised problems increases, so fewer iterations are performed between revisions of the rewards.

When the predicted values of D are used PJ:PJ & PB out performs PJ & PB in seven of the nine cases and the reduction in solution time is greater than 34% when $\lambda = 0.055$. The predicted optimal value of D is equal to the actual optimal value of D

Table 6.1: Serial PJ:PJ & PB and INVENT1

λ	0.002	0.008	0.015	0.055	0.066	0.105	0.171	0.197	0.235
F_{imp}	0.78	0.70	0.63	0.55	0.47	0.38	0.30	0.22	0.13
$\bar{\beta}$	0.95	0.95	0.94	0.91	0.85	0.77	0.64	0.45	0.24
Predicted values of D									
D	29	19	12	8	5	4	3	3	2
N_{iter}	4	5	8	10	22	23	38	39	66
T_{soln}	62.6	48.9	47.0	37.8	51.5	41.8	52.2	48.1	61.6
Optimal values of D									
D	23	11	10	9	7	4	4	3	2
N_{iter}	4	8	8	9	13	23	28	39	66
T_{soln}	50.6	47.5	40.1	37.6	39.7	41.8	45.6	48.1	61.6

PJ & PB requires 93 iterations and takes 57.4s to solve INVENT1 on a single transputer.

in three cases and in another two cases using the predicted value instead of the optimal value increases the solution time by less than 3%. Unfortunately with $\lambda = 0.002$ and 0.066 the solution time is increased by more than 20% when the predicted optimal value is used. However overall the method for predicting the optimal value of D is accurate enough to make PJ:PJ & PB a practical solution method.

It is interesting to note that in two cases ($\lambda = 0.0015$ and $\lambda = 0.055$) the total number of iterations performed on the original and revised problems in PJ:PJ & PB is less than the number of iterations required by PJ & PB. This is counter intuitive as it implies that iterating on the revised problems actually improves the rate of convergence. This behaviour is presumably spurious.

Table 6.2 gives details of the performance of GS:GS & $L_{\infty}B$ with INVENT1 for the range of thresholds described above. These results are in sharp contrast to those in table 6.1. In all cases the solution times for GS:GS & $L_{\infty}B$ are greater than those for LGS & $L_{\infty}B$. The best solution time is obtained using a very small threshold ($\lambda = 0.002$) and even this is more than 17% greater than the solution time for LGS & $L_{\infty}B$. When $\lambda \geq 0.066$ the optimal value of D is 1 indicating that during iterations on the revised problem little or no progress is made towards v^* .

Table 6.3 gives details of the performance of PJ:GS & PB with INVENT1 for the range of thresholds used in table 6.1. It is apparent from these results that PJ:GS &

REWARD REVISION

Table 6.2: Serial GS:GS & $L_\infty B$ and INVENT1 using the WLT ordering of the state space and the optimal values of D

λ	0.002	0.008	0.015	0.055	0.066	0.105	0.171	0.197	0.235
D	6	4	3	2	1	1	1	1	1
N_{iter}	5	8	11	17	28	28	28	28	28
T_{soln}	21.0	22.4	23.1	25.6	28.7	27.3	25.9	24.5	23.0

Note that F_{imp} and $\bar{\beta}$ are the same as for serial PJ:PJ & PB.
LGS & $L_\infty B$ requires 29 iterations and takes 17.8s to solve INVENT1 on a single transputer.

Table 6.3: Serial PJ:GS & PB and INVENT1 using the WLT ordering of the state space and the optimal values of D

λ	0.002	0.008	0.015	0.055	0.066	0.105	0.171	0.197	0.235
D	9	7	5	4	6	5	4	3	3
N_{iter}	4	6	10	16	13	19	25	38	55
T_{soln}	23.8	26.3	30.3	37.3	37.6	42.7	43.6	50.0	64.3

Note that F_{imp} and $\bar{\beta}$ are the same as for serial PJ:PJ & PB.
LGS & $L_\infty B$ requires 29 iterations and takes 17.8s to solve INVENT1 on a single transputer.

PB is not competitive with LGS & $L_\infty B$. This is because in PJ:GS & $L_\infty B$ the total number of iterations performed on the revised and original problems is far greater than the number of iterations performed by LGS & $L_\infty B$. In fact PJ:GS & PB does not even match the performance of PJ:PJ & PB in all cases. This is due to the relative effectiveness of the PB with the two algorithms. The PB are not effective in the hybrid case because, during iterations on the revised problem, the iteration matrices are not stochastic¹. Although PJ:PJ switches between problems, the optimal values are similar in all cases and the iteration matrices are all stochastic¹. Consequently the PB work well in this case. This effect can be observed in table 6.4 which shows the number of iterations required by PJ:PJ & $L_\infty B$ and PJ:GS & $L_\infty B$ for the range of thresholds used above. For example with $\lambda = 0.055$ replacing the PB with the $L_\infty B$ increases the number of iterations required by PJ:PJ and PJ:GS by 244% and 12% respectively. As λ increases the GS iterations in PJ:GS have less effect and hence the PB become more effective.

REWARD REVISION

Table 6.4: Number of iterations required in the solution of INVENT1 by PJ:PJ & $L_\infty B$ and PJ:GS & $L_\infty B$ using the optimal values of D

λ	0.002	0.008	0.015	0.055	0.066	0.105	0.171	0.197	0.235
PJ:PJ & $L_\infty B$	11	22	25	31	47	78	96	137	183
PJ:GS & $L_\infty B$	5	6	11	18	30	48	78	124	172

GEN1

For the three GEN1 problems there are on average six possible transitions resulting from each action. With $\lambda = 0.02$ the workload is reduced by a factor of 2 and $\bar{\beta}$ equals 0.868. With values of λ less than 0.02 the reduction in the workload is very slight and if the workload is reduced further (by increasing λ) the value of $\bar{\beta}$ falls sharply.

Table 6.5 gives details of the performance of PJ:PJ & PB with the three GEN1 problems for the case $\lambda = 0.02$. For GEN1FF and GEN1CS the solution times for PJ:PJ & PB are greater than those for PJ & PB, while for GEN1CV the improvement is only 10%. The poor performance of PJ:PJ & PB is due to insufficient progress being made during iterations on the revised problems. For example with GEN1FF the total number of iterations performed on the original and revised problems by PJ:PJ & PB is 36% greater than the number of iterations required by PJ & PB.

LGS & $L_\infty B$ solves the three GEN1 problems much more quickly than GS:GS & $L_\infty B$ with $\lambda = 0.02$. As in INVENT1 GS:GS & $L_\infty B$ makes little progress towards v^* during iterations on the revised problem (the optimal value of D for each problem is 1). PJ:GS & PB is far worse than LGS & $L_\infty B$ for all three problems and is even worse than PJ & PB for GEN1FF and GEN1CV.

Table 6.5: PJ:PJ & PB and GEN1 with $\lambda = 0.02$

Problem	GEN1FF	GEN1CV	GEN1CS
N_{iter}	166	49	379
T_{soln}	675	201	1538

In all cases $F_{imp} = 0.54$, $\bar{\beta} = 0.87$ and the predicted value of D and the optimal value of D are equal to 5.

REWARD REVISION

GEN2

There are no dominant transition probabilities in GEN2, so whenever a significant number of probabilities are discarded $\bar{\beta}$ is low. For example with $\lambda = 0.05$ the workload is approximately halved and $\bar{\beta}$ equals 0.79. For this scenario PJ:PJ & PB is slightly worse than PJ & PB and GS:GS & $L_{\infty}B$ is far worse than RGS & $L_{\infty}B$. Further the number of iterations required by PJ:GS & PB rises from 26 for $D = 1$ to 1142 for $D = 2$, so that in the latter case PJ:GS & PB takes twice as long as RGS & $L_{\infty}B$ to solve GEN2. Although the solution time is less for $D = 3$ and 4, PJ:GS & PB is never competitive with PJ or RGS.

Action elimination

The only two successful instances of reward revision have both involved PJ:PJ & PB. Therefore PJ:GS & PB with MAE and GS:GS & $L_{\infty}B$ with $L_{\infty}AE1$ are not considered. The two problems to benefit from reward revision are INVENT1 and GEN1CV but, since action elimination is not successful in the case of GEN1CV, PJ:PJ & PB with MAE is only considered with INVENT1. Table 6.6 gives details of the performance of PJ:PJ & PB with MAE and INVENT1 for the range of thresholds used in the earlier tables. In this implementation the value of D used does not vary during the solution procedure.

Table 6.6: Serial PJ:PJ & PB with MAE and INVENT1

λ	0.002	0.008	0.015	0.055	0.066	0.105	0.171	0.197	0.235
Predicted values of D									
D	29	19	12	8	5	4	3	3	2
N_{iter}	4	5	8	10	22	23	38	39	66
T_{soln}	21.5	16.9	15.1	13.5	14.8	13.4	15.1	15.0	16.8
Optimal values of D									
D	9	11	10	8	7	5	4	3	2
N_{iter}	11	8	8	10	13	21	28	39	66
T_{soln}	16.4	15.3	14.1	13.5	14.4	13.2	14.9	15.0	16.8

Note that F_{imp} and $\bar{\beta}$ are the same as for serial PJ:PJ & PB.

PJ & PB with PAE2 requires 93 iterations and takes 14.0s to solve INVENT1 on a single transputer.

REWARD REVISION

PJ:PJ & PB with MAE only out performs PJ & PB with PAE2 in two instances (although when the optimal values of D are used PJ:PJ & PB with MAE is superior to PJ & PB with MAE on a further two occasions) and the reduction in solution time is less than 6%. This is surprising when the performance of PJ:PJ & PB for this problem is recalled. The difference is not due to changes in the ratio of the work required during an iteration on a revised problem to the work required during an iteration on the original problem as actions are eliminated, because the number of possible transitions resulting from an action is almost constant in the original and the revised problems. This also means that the optimal value of D will not vary greatly from iteration to iteration and hence the predicted values of D work reasonably well. In such cases it is not efficient to modify the value of D periodically.

In the case of INVENT1 action elimination has less impact on PJ:PJ than PJ for three reasons. Firstly MAE is a weaker test than PAE2, but is used because PAE2 is not valid for PJ:PJ. Secondly the rate of convergence is poorer initially and consequently action elimination is less effective during early iterations of PJ:PJ. Finally less frequent application of the action elimination test increases the solution time.

Conclusions

Reward revision is successful in improving the performance of serial PJ for Markov decision processes which have a small number of dominant transition probabilities and are not highly sparse. This is because when all but the dominant probabilities are discarded, PJ iterations on the revised problems involve little calculation but still make good progress towards v^* .

The two reward revision algorithms based on GS are never successful, so for slowly mixing problems with a known WLT structure LGS is likely to be the best serial algorithm to use.

In general action elimination will be less effective with PJ:PJ than with PJ because MAE is a weaker test than PAE2, the initial rate of convergence is slower in PJ:PJ than in PJ, the computational advantage of iterating on a revised problem may decrease as actions are eliminated and less frequent application of action elimination does not

always reduce solution times.

6.5 Parallel reward revision algorithms

Since serial PJ:GS & PB and serial GS:GS & $L_\infty B$ have proved unsuccessful and the performance of GS generally degrades as the number of processors used increases, this section only considers parallel versions of PJ:PJ & PB. Let P denote the number of processors used and let $\{S_i^1\}_{i=1}^P$ and $\{S_i^2\}_{i=1}^P$ denote partitions of the state space. The following algorithm uses the principle of synchronous non-overlapped parallel value iteration to distribute the calculation performed at steps 3 and 5 of PJ:PJ & PB and modifies the transition structure by discarding all probabilities less than a given threshold.

Synchronous non-overlapped parallel PJ:PJ & PB

Processor r performs the following steps.

Step 1 Modify the transition structure

$$\text{Choose } \lambda > 0 \text{ and let } \bar{p}_{i,j}^k = \begin{cases} 0 & \text{if } p_{i,j}^k < \lambda \\ \frac{p_{i,j}^k}{\sum_{l: p_{i,l}^k \geq \lambda} p_{i,l}^k} & \text{otherwise} \end{cases} \quad \forall j \in S, \forall k \in K_i, \forall i \in S_r^1$$

Step 2 Modify the discount factor

$$\text{Calculate } \bar{\beta}_r = \beta \min_{i \in S_r^1, k \in K_i} \left\{ \sum_{j: p_{i,j}^k \geq \lambda} p_{i,j}^k \right\}$$

Perform a global minimisation to calculate $\bar{\beta} = \beta \min_{i \in S, k \in K_i} \left\{ \sum_{j: p_{i,j}^k \geq \lambda} p_{i,j}^k \right\}$ from the values $\bar{\beta}_1, \dots, \bar{\beta}_P$

Step 3 Distribute the transition data for the revised problems

Send $\{\bar{p}_{i,j}^k : j \in S, k \in K_i, i \in S_r^1 \cap S_t^2\}$ to processor $t \forall t \neq r$ and receive $\{\bar{p}_{i,j}^k : j \in S, k \in K_i, i \in S_t^1 \cap S_r^2\} \forall t \neq r$

Step 4 Initialisation

Choose D such that $2 \leq D < \infty$, let $v^{0,0} = 0$ and let $n = 0$

REWARD REVISION

Step 5 Revise rewards

Calculate $\bar{r}_i^{n,k} = R_i^k - \bar{\beta} \sum_{j \in S} \bar{p}_{i,j}^k v_j^{0,n} \quad \forall k \in K_i, \forall i \in S_r^1$ and $v_i^{1,n} = \max_{k \in K_i} \{R_i^k\} \quad \forall i \in S_r^1$

where $R_i^k = r_i^k + \beta \sum_{j \in S} p_{i,j}^k v_j^{0,n}$

Step 6 Distribute the latest estimates

Send $v_i^{1,n} \quad \forall i \in S_r^1$ to all other processors and receive $v_i^{1,n} \quad \forall i \notin S_r^1$

Step 7 Distribute the latest revised rewards

Send $\{\bar{r}_i^{n,k} : k \in K_i, i \in S_r^1 \cap S_t^2\}$ to processor $t \quad \forall t \neq r$ and receive $\{\bar{r}_i^{n,k} : k \in K_i, i \in S_t^1 \cap S_r^2\} \quad \forall t \neq r$

Step 8 Perform a bound calculation

Calculate $U_r^n = \max_{i \in S_r^1} \{v_i^{1,n} - v_i^{0,n}\}$ and $L_r^n = \min_{i \in S_r^1} \{v_i^{1,n} - v_i^{0,n}\}$

Perform a global maximisation and minimisation to calculate U^n and L^n from the values $U_1^n, \dots, U_P^n$ and $L_1^n, \dots, L_P^n$ respectively

Step 9 Check for convergence

If $\frac{\beta}{1-\beta} (U^n - L^n) \leq \epsilon$ then (1.3) is satisfied

Step 10 Initialisation for iterations on the current revised problem

Let $d = 2$

Step 11 Perform a PJ iteration on the current revised problem

Calculate $v_i^{d,n} = \max_{k \in K_i} \left\{ \bar{r}_i^{n,k} + \bar{\beta} \sum_{j \in S} \bar{p}_{i,j}^k v_j^{d-1,n} \right\} \quad \forall i \in S_r^2$

Send $v_i^{d,n} \quad \forall i \in S_r^2$ to all other processors and receive $v_i^{d,n} \quad \forall i \notin S_r^2$

Step 12 Control the number of iterations performed on the current revised problem

Increase d by 1 and if $d \leq D$ repeat from step 11

Step 13 Initialisation for another iteration

Increase n by 1, let $v^{0,n} = v^{D,n-1}$ and repeat from step 5

REWARD REVISION

As a revision of the rewards requires more calculation than an iteration on a revised problem, the optimal number of processors to use during steps 5 and 6 will be greater than during step 11. However adjusting the number of processors used greatly increases the amount of data which has to be transmitted during steps 3 and 7 and, in most cases, the overheads of the additional communication will outweigh any benefits. Therefore in this study the same number of processors are used at every step of the algorithm. In general, using this strategy eliminates the need for values to be sent to remote processors during steps 3 and 7.

Let A , t_z , t_p and t_a be as defined in section 6.3, let ψ be the number of operations which can be performed on a single processor in unit time and let $C(P, \ell)$ be the time required to perform a multinode broadcast when P processors are used and all messages broadcast are of length ℓ . Note that with the method used to modify the transition structure in the algorithm above, $t_a = 0$. $C(P, \ell)$ will depend upon the connection topology and communication protocol used.

If the number of actions per state and the number of transition probabilities which are less than the threshold per action do not depend on a systematic way on the order of the state space, it is reasonable to assume that states; actions; transitions which have non-zero probabilities in the original problem, but have zero probability in the revised problems and transitions which have non-zero probability in the original problem and the revised problems are evenly distributed between the processors. Therefore the time required to perform an iteration of synchronous non-overlapped parallel PJ & PB on the original problem is approximately equal to

$$\frac{(A - m + 2(t_z + t_p))\psi}{P} + C\left(P, \frac{m}{P}\right)$$

and the time required to perform an iteration of synchronous non-overlapped parallel PJ: PJ & PB is approximately equal to

$$\frac{(A - m + 2t_z + 4t_p)\psi}{P} + \frac{(D - 1)(A - m + 2t_p)\psi}{P} + DC\left(P, \frac{m}{P}\right)$$

Since the convergence properties of serial PJ & PB and PJ: PJ & PB are preserved in these parallel versions, the ratio of solution time for synchronous non-overlapped

REWARD REVISION

parallel PJ:PJ to the solution time for synchronous non-overlapped parallel PJ is approximately equal to

$$\frac{\log \beta}{\log \beta(D)} \left(\frac{(D(A - m + 2t_p) + 2(t_z + t_p)) \psi + PDC \left(P, \frac{m}{P}\right)}{(A - m + 2(t_z + t_p)) \psi + PC \left(P, \frac{m}{P}\right)} \right)$$

and the value of D used can be chosen to minimise this expression.

The number of actions; the number of transitions which have non-zero probabilities in the original problem, but have zero probability in the revised problems and the number of transitions which have non-zero probabilities in the original problem and the revised problems on each processor can easily be calculated when the transition structure is modified. A , t_p and t_z can then be calculated by performing global summations. During a global summation the value of $\alpha = \sum_t \alpha_t$ is calculated from the values $\alpha_1, \dots, \alpha_P$ where α_t is local to processor t . This operation can be performed in the same manner as a global maximisation since two values α_t and α_r can be combined into one value equal to $\alpha_r + \alpha_t$ (see section 5.3.3).

The estimates above do not allow for the bound calculation because, since it is not necessary to calculate bounds at every iteration, the time required to perform a bound calculation need never be a significant part of the solution time. The estimates above also ignore the time required for step 7 because, when the number of processors used does not vary during the solution procedure, this step only requires a small amount of local communication.

The choice of partitions will affect the time required for the communication tasks, the load balance during revisions of the rewards and the load balance during iterations on the revised problems. Section 5.4.2 notes that since communication procedures can be modified to compensate for imbalances in the number of states per processor, it is more important to address the issue of load balancing. The question of load balancing in PJ:PJ must be dealt with in two cases, because good load balancing during revisions of the rewards and during iterations on the revised problems cannot generally be obtained with the same partition. There are two possible approaches to this problem. The first chooses a partition which ensures good load balancing for one of the cases and the

REWARD REVISION

second uses two distinct partitions.

During each iteration of PJ:PJ processors are synchronised once after a revision of the rewards and $D - 1$ times after the calculation required during an iteration on a revised problem. Hence at each iteration the effect of a load imbalance during a revision of the rewards is apparent once, but the effect of a load imbalance during an iteration on a revised problem is apparent $D - 1$ times. If a partition which achieves perfect load balancing during revisions of the rewards is used and the load imbalance during other tasks is assumed to be proportional to the average amount of work allocated to a processor during those tasks, then the total load imbalance is proportional to $\frac{F_{imp}(D-1)}{P}$. On the other hand, if a partition which achieves perfect load balancing during iterations on the revised problems is used and the same assumption is made, then the total load imbalance is proportional to $\frac{1+F_{imp}}{P}$. Therefore to minimise the total load imbalance, choose a partition which achieves good load balancing during iterations on the revised problems if $F_{imp}(D - 2) > 1$ and a partition which achieves good load balancing during a revision of the rewards otherwise. During a revision of the rewards, a processor which is allocated the states in $A \subset S$ must perform $\widetilde{W}(A) = \sum_{i \in A} (|K_i| + 2 \sum_{k \in K_i} (|J_i^k| + |\bar{J}_i^k|) - 1)$ operations where $\bar{J}_i^k = \{\bar{p}_{i,j}^k : \bar{p}_{i,j}^k > 0, j \in S\}$ and J_i^k is as defined in section 5.4.2. During an iteration on a revised problem, a processor which is allocated the states in $A \subset S$ must perform $\overline{W}(A) = \sum_{i \in A} (|K_i| + 2 \sum_{k \in K_i} |\bar{J}_i^k| - 1)$ operations. The method of partition selection described in section 5.4.2 can be used with $\widetilde{W}$ to ensure good load balancing during revisions of the rewards or with $\overline{W}$ to ensure good load balancing during iterations on the revised problems.

With the second approach the functions $\widetilde{W}$ and $\overline{W}$ defined above are used in conjunction with the method of section 5.4.2 to select two partitions — one which achieves good load balancing during revisions of the rewards and one which achieves good load balancing during iterations on the revised problems.

The load balance is better when the second approach is used but, since steps 3 and 7 are not required when the first approach is used, communication overheads are

REWARD REVISION

higher. Processor r uses the following data during the n^{th} iteration.

$$\{\{r_i^k : k \in K_i, i \in S_r^1\}, \{p_{i,j}^k : j \in S, k \in K_i, i \in S_r^1\}, \\ \{\bar{r}_i^{n,k} : k \in K_i, i \in S_r^2\}, \{\bar{p}_{i,j}^k : j \in S, k \in K_i, i \in S_r^1 \cup S_r^2\}, \{v^{d,n} : 0 \leq d \leq D-1\}\}$$

Since $\bar{W}$ does not involve $|J_i^k|$, the first approach can result in a very uneven distribution of the data when the partition is chosen to ensure a good load balance during iterations on the revised problem. When the first approach is used a processor requires less transition data for the revised problems because $S_r^1 = S_r^2$.

When the state space is large compared to the number of processors used communication overheads can be reduced by applying the phased pipeline algorithm (see section 5.6) when revising the rewards or iterating on the revised problems. Although it has already been noted that altering the number of processors used involves significant overheads, the number of phases used can easily be switched. The optimal number of processors to use during a revision of the rewards is greater than that during an iteration on a revised problem, because a revision of the rewards involves more work. Hence, since using too many processors is preferable to using too few (see section 5.8), it is generally better to consider a revision of the rewards when fixing the number of processors to use. (5.6) can be applied to a revision of the rewards and the resulting expression for the time per revision leads to estimates of the optimal number of processors and phases, P^* and f_1^* respectively, to use during this part of the algorithm. (5.6) can then be applied to an iteration on a revised problem and the resulting expression for the time per iteration gives rise to an estimate of the optimal number of phases, f_2^* , to use during this part of the algorithm when P^* processors are used. This approach might not yield satisfactory results when the solution time is dominated by the iterations on the revised problems. In such cases an expression for the time per iteration of the reward revision algorithm can be derived by combining the two expressions mentioned above. This expression can then be employed to estimate P^* , f_1^* and f_2^* .

Using $\bar{W}(A)$ from above and the method of section 5.4.2, each set in the partition $\{S_i^1\}_{i=1}^{P^*}$ can be partitioned into f_1^* sets in a way which ensures approximate load

balancing. Similarly each set in the partition $\{S_i^2\}_{i=1}^{P^*}$ can be partitioned into f_2^* sets.

In this study the problems for which reward revision is successful do not have large enough state spaces to make the use of the phased pipeline algorithm worthwhile and so the phased pipeline algorithm has not been implemented with reward revision.

For some problems it is possible to reduce communication overheads by revising the transition structure. For example if there exists an ordering of the state space for which all transitions with dominant probabilities are between states which are close together, it is possible to restrict the communication performed during iterations on the revised problems so that each processor sends its latest estimates to local processors only. If in addition the problem is highly sparse and the number of dominant transitions is relatively large, then it is likely that value iteration algorithms would also benefit from modified communication procedures. However if the problem is dense, this is not the case and an iteration on a revised problem would involve dramatically less calculation and communication than an iteration on the original problem. No Markov decision processes with this special structure could be found, so this technique has not been implemented. However the technique may be useful in the policy evaluation step of a policy iteration method.

6.6 Implementing synchronised non-overlapping parallel reward revision

This section compares the performance of the parallel reward revision algorithms proposed in section 6.5 with that of the best parallel value iteration algorithms. Section 6.4 concludes that reward revision is well suited to problems with transition structures which have a small number of dominant probabilities and are not highly sparse. This type of structure is found in single and multiple item inventory problems in which the probability distributions of the demands have long tails. Hence this section will concentrate on INVENT1 and the five small dense multiple item inventory problems from table 3.4. For these problems the proposed WLT orderings of the state spaces put states in the order of increasing number of items in stock. Consequently the workload per state increases with the state number. Random orderings of the state space are

REWARD REVISION

used for PJ:PJ and PJ, because this results in better load balancing and the order of the state space does not affect the convergence. The solution times for PJ:PJ in tables 6.7, 6.8 and 6.10 correspond to an implementation which uses the same partition during revisions of the rewards and iterations on the revised problems, because for every test problem this approach proved more efficient than using two distinct partitions. As $F_{imp}(D - 2) > 1$ for each problem, the partition used was chosen to ensure good load balancing during iterations on the revised problems. The algorithms are implemented on a ring of processors using the procedures BROADR and GMAX to perform the communication tasks required.

The times presented in the result tables are in seconds. The solution time does not include the time required to obtain the data initially. For every problem considered in this section the data is generated locally on each processor and the time required is small compared to the solution time. Details of the notation used in the tables can be found on page xiv.

INVENT1

Tables 6.7 and 6.8 give details of the performance of PJ:PJ & PB and PJ:PJ & PB with MAE using INVENT1 with $\lambda = 0.055$ (the threshold which gives the best performance in the serial case). Although the load balance for PJ:PJ & PB is poorer than the load balance for PJ & PB, the solution times for PJ:PJ & PB are consistently better than those for PJ & PB. Compared to PJ & PB, PJ:PJ & PB effects a 34% reduction in solution time when $P = 1$ and the reduction is still 19% when $P = 49$ and

Table 6.7: Synchronous non-overlapped parallel PJ:PJ & PB and INVENT1 with $\lambda = 0.055$

P	Predicted values of D					Optimal values of D				
	D	N_{iter}	T_{soln}	ξ	ξ_{comm}	D	N_{iter}	T_{soln}	ξ	ξ_{comm}
1	8	10	37.84	100.0	0.0	9	9	37.60	100.0	0.0
5	8	10	7.74	97.8	0.3	9	9	7.69	97.8	0.4
11	8	10	3.74	92.0	1.1	9	9	3.72	91.9	1.3
25	7	12	1.95	77.5	5.1	9	9	1.82	82.9	5.3
49	7	12	1.41	54.8	12.6	9	9	1.31	58.6	13.0

REWARD REVISION

Table 6.8: Synchronous non-overlapped parallel PJ:PJ & PB with MAE and INVENT1 with $\lambda = 0.055$

P	Predicted values of D					Optimal values of D				
	D	N_{iter}	T_{soln}	ξ	ξ_{comm}	D	N_{iter}	T_{soln}	ξ	ξ_{comm}
1	8	10	13.55	100.0	0.0	8	10	13.55	100.0	0.0
5	8	10	2.88	93.9	1.0	8	10	2.88	93.9	1.0
11	8	10	1.52	80.8	3.0	8	10	1.52	80.8	3.0
25	7	12	0.81	66.9	12.3	8	10	0.79	68.4	11.9
49	7	12	0.68	40.4	25.9	8	10	0.67	41.5	25.3

the predicted value of D is used. Surprisingly the load balance for PJ:PJ & PB with MAE is better than the load balance for PJ & PB with PAE2. This is an artefact of the pattern of action elimination in the two cases and will not be true in general. However as a result the advantage of PJ:PJ & PB with MAE over PJ & PB with PAE2 for INVENT1 increases with the number of processors used and the reduction in solution time reaches 10% when $P = 49$ and the predicted value of D is used.

Although not always exact, the method for predicting the optimal value of D proves accurate enough to make this parallel version of PJ:PJ & PB a practical solution method for Markov decision processes. In the worst case (PJ:PJ & PB using 49 processors) using the predicted value of D instead of the optimal value increases the solution time by 8%.

Communication overheads for PJ & PB and PJ:PJ & PB are similar because in PJ:PJ & PB, although on average less calculation is performed between multinode broadcasts, the bound calculation is not performed as frequently.

Multiple item inventory problems

Table 6.9 gives details of the revised transition structure of the five multiple item inventory problems (see table 3.4) and table 6.10 uses these problems to compare the performance of PJ:PJ & PB with that of PJ & PB and the best GS variant. The problems each have 540 states, 48 actions and between 1 and 5 items. The demand for an item has a Poisson distribution with mean equal to one tenth of the storage capacity for that item. F_{imp} increases as the dimension of the problem increases, because the

Table 6.9: Revised transition structure for five multiple item inventory problems

Problem	mi1	mi2	mi3	mi4	mi5
Number of items	1	2	3	4	5
Storage capacities	539	14 & 35	9, 8 & 5	9, 5, 2 & 2	8, 4, 2, 1 & 1
λ	0.0130	0.0074	0.0055	0.0038	0.0040
F_{imp}	0.09	0.20	0.30	0.38	0.38
$\bar{\beta}$	0.90	0.90	0.90	0.90	0.90

Table 6.10: A comparison of parallel iterative methods for five multiple item inventory problems

Algorithm		Problem				
		mi1	mi2	mi3	mi4	mi5
PJ:PJ & PB	D	13	9	8	7	7
	N_{iter}	25	13	9	13	21
	T_{soln}	30.1	7.60	2.85	2.94	3.13
	ξ_{comm}	5.9	8.4	13.9	17.2	25.9
PJ:PJ & PB with MAE	D	13	9	8	7	7
	N_{iter}	25	13	9	13	21
	T_{soln}	17.8	3.83	1.59	1.64	1.77
	ξ_{comm}	10.1	16.7	24.9	30.9	45.7
PJ & PB	N_{iter}	122	74	51	63	105
	T_{soln}	68.3	16.6	4.10	3.30	3.89
	ξ_{comm}	1.2	2.8	8.1	12.0	17.0
PJ & PB with PAE2	N_{iter}	122	74	51	63	105
	T_{soln}	51.9	5.70	1.88	1.77	2.06
	ξ_{comm}	1.5	8.2	17.7	22.2	32.1
Best GS variant	N_{iter}	629	1464	1400	1283	1234
	T_{soln}	167	116	58.4	36.9	26.7
	ξ_{comm}	2.6	8.0	15.6	21.7	29.1

All the results were obtained using 111 processors and the solution times for the reward revision algorithms were obtained using the predicted values of D . These problems cannot be implemented on less than 50 processors with the current data structures.

lengths of the tails of the probability distributions decrease with the storage capacity of the items and consequently the proportion of transitions with large probabilities increases. The discount factor in each case is 0.99 and λ is chosen so that $\bar{\beta} \simeq 0.90$. The test problems chosen are small because large scale multiple item inventory problems cannot be implemented without data structures which exploit the structure of the problems.

Table 6.11: Theoretical results for large scale multiple item inventory problems

Number of items	1	2	3	4
Storage capacities	9	9 & 9	9, 9 & 9	9, 9, 9 & 9
Maximum order quantity	3	3 & 3	3, 3 & 3	3, 3, 3 & 3
λ	0.1630	0.0200	0.0072	0.0012
F_{imp}	0.55	0.33	0.22	0.15
$\bar{\beta}$	0.93	0.87	0.88	0.87

Although reward revision is generally less effective when action elimination is employed, the results show PJ:PJ & PB with MAE to be the best algorithm to use for these five problems. The next best algorithm in each case is PJ & PB with PAE2, but this algorithm takes three times as long in one instance.

The results for GS show that, as indicated from the results for INVENT1, parallel GS is not an efficient solution algorithm for this type of problem. This is due to load balancing problems when the WLT orderings are used.

Communication overheads are high because, since the average number of states allocated to a processor is just five, channel set-up time is the dominant term in the time required for the multinode broadcast.

Although large scale multiple item inventory problems cannot be implemented with the current data structures, theoretical results suggest that the effectiveness of the reward revision technique increases as the dimension of the problems increases. This is illustrated in table 6.11 which shows the variation in F_{imp} and $\bar{\beta}$ as the dimension of the problem increases.

Conclusions

Although communication overheads in PJ:PJ are higher than in PJ and the load balance is generally poorer, the results of this section show that parallel synchronous PJ:PJ & PB is a useful solution method when the transition structure of the problem being solved is well suited to the reward revision technique.

The results in this section also verify that reward revision is a useful technique for problems which benefit from action elimination and that the poor performance of PJ:PJ & PB with MAE on INVENT1 is due to particular properties of that problem.

REWARD REVISION

With all the problems considered in this section, using two distinct partitions led to an increase in solution time because the additional communication overheads proved to be more significant than the improved load balance. This suggests that unless the variation in the work per state in the original problem differs greatly from that of the revised problems, using two distinct partitions is not efficient.

Footnotes

1. To be precise, the iteration matrices in PJ are not stochastic, but are equal to a scalar multiple of a stochastic matrix, see section 2.2.

Chapter 7

Summary of the main conclusions

The performance of the serial algorithms on the test problems shows that there is no algorithm which is best for all problems and also shows that the performance of an algorithm on a particular problem can be predicted from the properties of that problem. The speed of mixing and the existence of a known WLT ordering of the state space are the most important properties to use in the prediction, because PJ performs well if the problem is rapidly mixing or no WLT ordering is known and GS performs well if a WLT ordering is known.

The comparison of connection topologies and communication protocols shows that for the communication required in the iterative algorithms considered in this study, a ring of processors is as good as any other connection topology when the message length is large. Other connection topologies, for example the torus, require fewer channel set-ups and are therefore more efficient for small message lengths.

The success of the synchronous non-overlapped parallel value iteration algorithms shows that the parallelism inherent in Markov decision processes allows data to be distributed and faster solution times. Although the rate of convergence in parallel GS algorithms deteriorates as the number of processors used increases, these algorithms are still useful for slowly mixing problems with a known WLT structure. When there is no trend in the work per state, the major factor limiting the efficiency of synchronous non-overlapped parallel PJ is communication overheads.

Communication overheads can be reduced by using an asynchronous approach, but experience has shown that the resulting reduction in the rate of convergence more

SUMMARY OF THE MAIN CONCLUSIONS

than cancels any gain. This effect is most pronounced with PJ because the special convergence properties of synchronous PJ algorithms are destroyed.

The phased pipeline approach enables calculation and communication to be overlapped and improves the rate of distribution of the latest estimates of the values of states. This proves to be an efficient solution method for large sparse Markov decision processes, achieving higher speed-ups with drastically fewer processors than algorithms which do not phase the calculation. PJ algorithms based on the phased pipeline approach benefit from lower communication overheads, but GS algorithms based on the phased pipeline approach not only benefit from lower communication overheads, but also from faster convergence.

Action elimination is a useful addition to serial and parallel versions of both PJ and GS provided non-optimal policies do not have transition structures or vectors of values which are close to those of the optimal policy.

The reward revision technique has been shown to improve the performance of serial and parallel PJ for Markov decision processes which have a small number of dominant transition probabilities and are not highly sparse. This transition structure is typical of certain classes of single and multiple item inventory problems. The two methods of incorporating GS in the reward revision algorithm proved unsuccessful with all the test problems.

Bibliography

- [1] D. Al-Dabass (1980) "Two methods for the solution of the dynamic programming algorithm on a multiprocessor cluster". *Optimal Control Applications and Methods*, 1, pp 227–238.
- [2] D. Aldous (1983) "Random walks on finite groups and rapidly mixing Markov chains". In *Séminaire de probabilités XVII: proceedings* edited by A. Dold and B. Eckmann, pp 243–297. Springer-Verlag, Berlin.
- [3] T.W. Archibald (1991) "Parallel dynamic programming". To appear in *Advances in parallel algorithms* edited by L. Kronsjo and D. Shumsheruddin. Blackwell Scientific, Oxford.
- [4] T.W. Archibald, K.I.M. McKinnon and L.C. Thomas (1990) "Serial and parallel value iteration algorithms for discounted Markov decision processes". Working Paper Series 90/14, Department of Business Studies, University of Edinburgh, Scotland. To appear in *European Journal of Operational Research*.
- [5] T.W. Archibald, K.I.M. McKinnon and L.C. Thomas (1991) "On the random generation of Markov decision processes". Working Paper, Department of Mathematics, University of Edinburgh, Scotland.
- [6] T.W. Archibald, K.I.M. McKinnon and L.C. Thomas (1991) "Performance issues for the iterative solution of Markov decision processes on parallel computers". Working Paper Series 91/23, Department of Business Studies, University of Edinburgh, Scotland. Submitted to *ORSA Journal on Computing*.

BIBLIOGRAPHY

- [7] T.W. Archibald, K.I.M. McKinnon and L.C. Thomas (1992) "Parallel value iteration algorithms for discounted Markov decision processes". In *One-day workshop on parallel numerical analysis: proceedings* edited by D.B. Duncan, K.I.M. McKinnon and F. Plab, pp 14–22. Technical Report EPCC-TR92-05, Edinburgh Parallel Computing Centre, University of Edinburgh, Scotland.
- [8] R. Bellman (1957) *Dynamic programming*, Princeton University Press, Princeton, New Jersey.
- [9] R. Bellman and S. Dreyfus (1962) *Applied dynamic programming*, Princeton University Press, Princeton, New Jersey.
- [10] D.P. Bertsekas (1982) "Distributed dynamic programming". *IEEE Transactions on Automatic Control*, **AC-27**, pp 610–616.
- [11] D.P. Bertsekas (1987) *Dynamic programming: deterministic and stochastic models*, Prentice-Hall, Englewood Cliffs, New Jersey.
- [12] D.P. Bertsekas and J.N. Tsitsiklis (1989) *Parallel and distributed computation: numerical methods*, Prentice-Hall, Englewood Cliffs, New Jersey.
- [13] J. Boreddy and A. Paulraj (1990) "On the performance of transputer arrays for dense linear systems". *Parallel Computing*, **15**, pp 107–117.
- [14] L. Brochard (1989) "Efficiency of some parallel numerical algorithms on distributed systems". *Parallel Computing*, **12**, pp 21–44.
- [15] J. Casti, M. Richardson and R. Larson (1973) "Dynamic programming and parallel computers". *Journal of Optimization Theory and Applications*, **12**, pp 423–438.
- [16] V. Conrad and Y. Wallach (1977) "A faster SSOR algorithm". *Numerische Mathematik*, **27**, pp 371–372.
- [17] E.V. Denardo (1967) "Contraction mappings in the theory underlying dynamic programming". *SIAM Review*, **9**, pp 165–177.

BIBLIOGRAPHY

- [18] P. Edmonds, E. Chu and A. George (1990) "Dynamic programming on a shared-memory multiprocessor". Report CS-90-49, Department of Computer Science, University of Waterloo, Canada.
- [19] L.R. Ford and D.R. Fulkerson (1962) *Flows in networks*, Princeton University Press, Princeton, New Jersey.
- [20] E.S. Fraga and K.I.M. McKinnon (1991) "Parallel optimization techniques for optimal heat integrated separation sequence generation". Technical Report 1991-2, Department of Chemical Engineering, University of Edinburgh, Scotland.
- [21] T.L. Freeman (1989) "Calculating polynomial zeros on a local memory parallel computer". *Parallel Computing*, **12**, pp 351-358.
- [22] P. Gachet, B. Joannault and P. Quinton (1986) "Synthesizing systolic arrays using DIASTOL". In *Proceedings of the International Workshop on Systolic Arrays* edited by W. Moore, A. McCabe and R. Urquhart, pp 25-36. Adam Hilger, Bristol, UK.
- [23] R. Hartley, A.C. Lavercombe and L.C. Thomas (1986) "Computational comparison of policy iteration algorithms for discounted Markov decision processes". *Computers and Operations Research*, **13**, pp 411-420.
- [24] N.A.J. Hastings and J.M.C. Mello (1973) "Tests for suboptimal actions in discounted Markov programming". *Management Science*, **19**, pp 1019-1022.
- [25] M. Hendrikx, J. van Nunen and J. Wessels (1984) "On iterative optimization of structural Markov decision processes". *Mathematische Operationsforschung und Statistik*, **15**, pp 439-459.
- [26] J.E. Hendry and R.R. Hughes (1972) "Generating separation process flowsheets". *Chemical Engineering Progress*, **68**(6), pp 71-76.
- [27] R.W. Hockney and C.R. Jesshope (1988) *Parallel computers 2: architecture, programming and algorithms*, Adam Hilger, Bristol.

BIBLIOGRAPHY

- [28] R.A. Howard (1960) *Dynamic programming and Markov processes*, John Wiley, New York.
- [29] D.E. Knuth (1981) "Breaking paragraphs into lines". *Software Practice and Experience*, **11**, pp 1119–1184.
- [30] L. Kucera (1982) "Parallel computation and conflicts in memory access". *Information Processing Letters*, **14**, pp 93–96.
- [31] B. Louka and M. Tchunte (1988) "Dynamic programming on two-dimensional systolic arrays". *Information Processing Letters*, **29**, pp 97–104.
- [32] D.G. Luenberger (1969) *Optimization by vector space methods*, John Wiley, New York.
- [33] J. MacQueen (1966) "A modified dynamic programming method for Markovian decision problems". *Journal of Mathematical Analysis and Applications*, **14**, pp 38–43.
- [34] J. MacQueen (1967) "A test for suboptimal actions in Markovian decision problems". *Operations Research*, **15**, pp 559–561.
- [35] N. Matelan (1985) "The Flex/32 multicomputer". *Proceedings of the 12th International Symposium on Computer Architecture*, pp 209–213. IEEE Computer Society Press, Washington.
- [36] S. Miguet and Y. Robert (1989) "Dynamic programming on a ring of processors". In *Hypercube and distributed computers* edited by F. André and J.P. Verjus, pp 19–33. North Holland, Amsterdam.
- [37] T. Morton and W. Wecker (1977) "Discounting ergodicity and convergence for Markov decision processes". *Management Science*, **23**, pp 890–900.
- [38] D.M. Nicol (1989) "Parallel solution of sparse one-dimensional dynamic programming problems". ICASE report 89-17, NASA, Langley Research Center, Hampton, Virginia.

BIBLIOGRAPHY

- [39] E.L. Porteus (1971) "Some bounds for discounted sequential decision processes". *Management Science*, **18**, pp 7–11.
- [40] E.L. Porteus (1975) "Bounds and transformations for discounted finite Markov decision chains". *Operations Research*, **23**, pp 761–784.
- [41] E.L. Porteus (1980) "Improved iterative computation of the expected discounted return in Markov and semi-Markov chains". *Zeitschrift für Operations Research*, **24**, pp 155–170.
- [42] E.L. Porteus (1981) "Computing the discounted return in Markov and semi-Markov chains". *Naval Research Logistics Quarterly*, **28**, pp 567–577.
- [43] E.L. Porteus and J.C. Totten (1978) "Accelerated computation of the expected discounted return in a Markov chain". *Operations Research*, **26**, pp 350–358.
- [44] M.L. Puterman and M.C. Shin (1978) "Modified policy iteration algorithms for discounted Markov decision problems". *Management Science*, **24**, pp 1127–1137.
- [45] W. Rytter (1988) "On efficient parallel computations for some dynamic programming problems". *Theoretical Computer Science*, **59**, pp 297–307.
- [46] D. Tabak (1968) "Computational improvement of dynamic programming solutions by multiprocessing techniques". *IEEE Transactions on Automatic Control*, **AC-13**, p 596.
- [47] L.C. Thomas (1981) "Second order bounds for Markov decision processes". *Journal of Mathematical Analysis and Applications*, **80**, pp 294–297.
- [48] L.C. Thomas, R. Hartley and A.C. Lavercombe (1982) "Computational comparison for discounted Markov decision processes — value iteration". Notes in Decision Theory 100, Department of Decision Theory, University of Manchester, England.

BIBLIOGRAPHY

- [49] L.C. Thomas, R. Hartley and A.C. Lavercombe (1983) "Computational comparison of value iteration algorithms for discounted Markov decision processes". *Operations Research Letters*, 2, pp 72–76.
- [50] H.C. Tijms (1986) *Stochastic modelling and analysis: a computational approach*, John Wiley, Chichester.
- [51] A. Trew and G. Wilson (Editors) (1991) *Past, present, parallel: a survey of available parallel computing systems*, Springer-Verlag, Berlin.
- [52] R.S. Varga (1962) *Matrix iterative analysis*, Prentice-Hall, Englewood Cliffs, New Jersey.
- [53] J. Wexler (1989) *Concurrent programming in OCCAM 2*, Ellis Horwood, Chichester.
- [54] C.C. White, L.C. Thomas and W.T. Scherer (1985) "Reward revision for discounted Markov decision problems". *Operations Research*, 33, pp 1299–1315.
- [55] D.J. White (1985) "Real applications of Markov decision processes". *Interfaces*, 15(6), pp 73–78.
- [56] P.W. Williams (1972) *Numerical computation*, Thomas Nelson, London.