DUBLIN CITY UNIVERSITY

School of Electronic Engineering

Thesis Submitted for the Degree of

Master of Engineering

Aspects of Parallel Processing and Control Engineering

by:

Brendan McKittrick, B.Eng.

for:

Dr. John V. Ringwood.

September 1991

I hereby declare that the research herein
was completed by the undersigned

Signed: [illegible] Date: 23RD Sept. 1991.

To my parents, Seamus & Maureen, who made it all possible.

Acknowledgements

Sincere gratitude is expressed to my supervisor Dr. John Ringwood, whose interest in the project equalled my own. Thanks also to my colleagues Paddy Gibbs, Dara Murtagh, Niall Byrne, Greg Reilly and Liam O'Gradaigh who were always available for comment. A special thanks must be given to two of my sisters, Meadhbh and Fiona, who spent considerable time screening the text for my many typing errors and for my brothers Joseph and Tom, other sisters Marie, Eilish, Grainne and Siobhan, grandmother Maura Kennedy, and parents Seamus and Maureen, for being perpetually helpful and patient.

ABSTRACT

Aspects of Parallel Processing and Control Engineering

Submitted for the award of Degree of Master of Engineering

By: Brendan Mc Kittrick

The concept of parallel processing is not a new one, but the application of it to control engineering tasks is a relatively recent development, made possible by contemporary hardware and software innovation. It has long been accepted that, if properly orchestrated several processors/CPUs when combined can form a powerful processing entity. What prevented this from being implemented in commercial systems was the adequacy of the microprocessor for most tasks and hence the expense of a multi-processor system was not justified. With the advent of high demand systems, such as highly fault tolerant flight controllers and fast robotic controllers, parallel processing became a viable option.

Nonetheless, the software interfacing of control laws onto parallel systems has remained somewhat of an impasse. There are no software compilers at present which allow a programmer to specify a control law in pure mathematical terminology and then decompose it into a flow diagram of concurrent processes which may then be implemented on, say, a target Transputer system. There are several parallel programming languages with which a programmer can generate parallel processes but, generally, in order to realise a control algorithm in parallel the programmer must have intimate knowledge of the algorithm. Therefore, efficiency is based on the ability of the programmer to recognise inherent parellelism. Some attempts are being made to create intelligent partition and scheduling compilers but this usually means significantly extra overheads on the multiprocessor system. In the absence of an automated technique control algorithms must be decomposed by inspection.

The research presented in this thesis is founded upon the application of both parallel and pipelining techniques to particular control strategies. Parallelism is tackled objectively and by creating a tailored terminology it is defined mathematically, and consequently related concepts, such as bounded parallelism and algorithm speedup, are also quantified in a numerical sense. A pipelined explicit Self Tuning Regulator (STR) controller is developed and tested on systems of different order. Under the governance of the parallelism terminology the effectiveness of the parallel STR is evaluated and numerically quantified in terms of relevant performance indices.

A parallel simulator is presented for the Puma 560 robotic manipulator. By exploiting parallelism and pipelinability in the robot model a significant increase in execution speed is achieved over the sequential model. The use of Transputers is examined and graphical results obtained for several performance indices, including speedup, processor efficiency and bounded parallelism. By the same analytical technique a parallel computed torque feedforward controller incorporating proportional derivative feedback control for the Puma 560 manipulator is developed and appraised. The performance of a Transputer system in hosting the controller is graphically analysed and as in the case of the parallel simulator the more important performance indices are examined under both optimal conditions and conditions of varying hardware constraints.

CONTENTS

CHAPTER 1

Introduction

1.1 PARALLEL PROCESSING & CONTROL ENGINEERING

Presently, control systems are often required to perform a range of tasks, such as input/output handling, data logging, etc., on top of running the actual control algorithm. In many applications these tasks must be executed in real time. Ever since it's development as a commercial product the microprocessor has generally sufficed as a hardware tool in most areas of control and has been mostly used in a single processor capacity. The advent of new and more stringent demands for most control applications however, such as higher sampling rates, high performance industrial electro-mechanical systems and fault tolerant systems has led to experimentation with a variety of multiple processor systems. Nearly all of these systems involve processors operating on or generating data concurrently, or in parallel, hence the term *parallel processing*.

It may seem intuitive that by distributing the workload over a number of processors in parallel an improvement in execution time is achieved but this is not necessarily always the case. The success of a parallel implementation can depend on an efficient distribution of the workload, and the partitioning of an algorithm in an ad hoc or insouciant manner may yield a computationally more intensive application than the original sequential algorithm. It may also happen that the task is inherently sequential and may not be decomposable into parallel form. Furthermore, in some cases the communications overheads between the processors may outweigh any speedup gains achieved, and in other more desirable instances the algorithm may map smoothly onto an a tailored parallel hardware structure, for example the standard PID control algorithm may be decomposed onto an medium-grained array structure, where each node is a simple arithmetic operation, as described by Jones and Spray[82]. It is of paramount importance that an attempt is made to harmonise the granularity of the algorithmic decomposition with the granularity of the target multiprocessing system to optimise costs and maximise speedup. It would be a travesty of good hardware design to use a microprocessors for each of the processing elements in the last example. An array of medium grained processing elements, such as the PACE architecture[83], is optimal. A topology with relatively fewer processing elements that are capable of performing more complex data manipulations is known as a coarse grain approach. Alternatively, an array of processing elements that perform bit-wise operations on a data set that ripples through the array, similar to schemes proposed by Gaston and Irwin[21][23], is known as a fine grained parallel decomposition strategy.

Chapter 1: *Introduction*

Although the introduction of parallel processing techniques to control engineering is still at a relatively novel stage the union of the two fields has already manifested itself under several different research interests, the most prominent of which are:

- Parallel simulation of real systems.
- Development of parallel-specific control strategies and *fast* algorithms.
- Mapping of control laws onto tailored hardware topologies.

The second research topic listed above, which is concerned with the general formulation of algorithms and control strategies that are specifically suited to parallel implementation, has experienced somewhat of an impasse, mostly due to the lack of CACSD packages that support automated parallel algorithm design. Only parallel algorithms for specific applications have generally been postulated and very little successful work has be undertaken in the design of automated parallel algorithmic compilers, which, based on a user-defined control algorithm specification, automate the parallelisation and partitioning of the algorithm. In effect the parallelisation of most algorithms, including those presented in the research literature, has been the result of human inspection.

It is in the two other fields, however, that the research contained herein has been undertaken: *system simulation* and the *parallelisation of controllers*. The former represents the innovative study of the simulation of systems using the tools of parallel software and parallel hardware. The main objective is to create real-time, or faster, simulators and in some instances more fault tolerant simulators. In the research described by the thesis a validated parallel simulator for the Puma 560 robotic manipulator is presented. The latter field, the exploitation of parallelism in control strategies and structures, is perhaps the more common research interest since it covers both the innovations that originally spawned from the initial introduction of multiprocessors to control and the mainstream of research currently being undertaken worldwide. The technique of pipelining, a parallelisation method that has been in existence for quite some time, may be used to impose a concurrency scheme on an algorithm which may be inherently sequential. This is illustrated by the application of a multi-stage pipeline to an *explicit-self tuning regulator*, presented in Chapter 4. Contrasting to this pipelining of a sequential control algorithm an inherently parallel control structure is exemplified by the *parallel computed torque feedforward controller incorporating PD feedback control* for the Puma 560 robotic manipulator. This controller is presented and evaluated on a Transputer multiprocessing system in terms of specific performance indices, which are developed in Chapter 3.

1.2 MOTIVATION FOR RESEARCH

Since the early 1970's the importance of the robotic manipulator in industry has spiraled. This is due in no small way to the increased need for higher productivity, a need which is intensifying as world markets slow down and competition rises. Most manufacturing tasks are performed by highly mechanised equipment, but generally these machines lack adaptability and flexibility. The growth of assembly tasks requiring flexibility and machine intelligence, such as spot welding, spray painting, part assembly or materials handling, has made the *robot* a viable employee.

A further demand for mobile robot presence has arisen, not from solely commercial interests, but from modern research fields, including space exploration, sea exploration, experiments with toxic chemicals and/or radioactive substances and generally any interaction with hazardous or new environments. These often require the robot to perform tasks in proxy for a human and sometimes necessitate a limited ability for decision making. All these growing interests seem to indicate that the future holds greater demands and requirements for robotic vehicles and intelligent manipulators, and their controllers.

To meet the stringent margins necessary to perform in these environments robots need accurate control strategies, which are effective over the complete range of operation. In most instances only small errors in manipulator end-effector position are acceptable, thus requiring an error compensatory scheme to be incorporated into the controller, through the use of feedback or otherwise. This further intensifies the computational load of the controller and in some instances it may be necessary to modify the controller to execute within the sampling period constraint, possibly resulting in degraded performance. Whilst modification or redesign of an algorithm to meet the execution time requirements for the control of a system is merely treating the symptoms of a greater problem, parallel processing techniques may offer a more complete solution, without necessarily altering the integrity of the controller. It is on this premise that a parallel controller for the Puma 560 robotic manipulator is postulated. Combining both the computed torque feedforward and the PD feedback strategies the controller is implemented in parallel so as to meet shorter sampling times. The parallel realisation combines both a coarse grained and medium grained approach, and this is reflected in the choice of hardware and the parallel coding scheme.

One of the greatest challenges of robotic development, other than their control, is the testing and verification of their capacities and capabilities in real world situations, before assigning them to tasks which may, like many of the above applications, have little or no margin for error. Obviously, a traditional trial and error approach, whereby the machine's performance is tested and validated in its work environment, would be a travesty of experimental procedure, thus an alternative method is needed. This problem, which is so common, finds its solution in the use of *models* and the practice of *simulation*. This is especially true of the robot manipulator, which is generally an expensive and potentially self-damaging piece of machinery.

A software robot simulator is a piece of software written to mimic the behaviour, as closely as is possible, of a robot under a prescribed range of external conditions and inputs. This simulator includes a software coded model of the robot. A robot model is an equation, or several related equations, which describe the relationship between the inputs (e.g. joint voltages) to the internal states of the robot (e.g. forces on the joints) and the outputs of the robot (e.g. joint positions).

The procedure of modeling and simulation is no less important for the Puma 560 robot manipulator and the possession of a validated and complete model is necessary to evaluate its behaviour. This becomes more apparent when one considers the need to evaluate controllers under the full range of conditions, without actually putting the manipulator under the control of such a controller. This is probably one of the more important roles for a validated robot simulator.

One of the major drawbacks of most robot simulators is there inability to perform in real-time due to computationally intensive differential model equations, thus restricting their use in real-time controller evaluation. Naturally, as the trend for faster performances in control engineering more frequently finds expression through the use of parallelisation techniques it is apt that an attempt is made to develop a parallel simulator for the Puma 560 manipulator. This is in the spirit of the growing acceptance of parallel processing among the control community as the panacea for the problems caused by the computational intensity of many algorithms, including the dynamical models of robotic manipulators.

1.3 SUMMARY OF DISSERTATION

The following prologue briefly explains the purpose of each of the chapters in the thesis.

- □ **Chapter 2** contains an extensive investigation into field of parallel processing technology, including hardware and software analysis. It serves as a comprehensive introduction to the present state of parallel processing from an objective viewpoint and also contains a subsection on the application of this technology to general control practices.

- □ The concept of parallel processing is pondered in **Chapter 3**, and through the use of mathematical formulation is quantified. Other related concepts, such as speedup, maximum parallelism and bounded parallelism, are developed for defined hardware scenarios. These factors prove to be effective performance indicators and are necessary indices in the optimisation of a parallelisation scheme.

- □ Parallelism may be inherent in a control structure or it may be imposed. One method to impose concurrency upon the operations of an algorithm is by pipelining. **Chapter 4** illustrates the technique and effects of pipelining an explicit self tuning regulator (STR), a controller from the family of adaptive controllers. The STR is inherently sequential and offers practically no scope for parallel decomposition of its algorithmic operations. The development of a five stage pipeline increases the speed of the algorithm and allows this control strategy to be used under more stringent sampling period constraints.

- □ One of the main motivating factors for the research was the exciting prospect of developing a parallel model for the Puma 560 robotic manipulator, and the development of a parallel controller for the same. In **Chapter 5** the former is accomplished and is incorporated into a complete robotic simulator. The optimum Transputer hardware and Parallel C software configurations necessary to implement the simulator are presented. By the use of graphical analysis the simulator is validated. The eventuality of reduced numbers of processors is also considered and its effect on overall performance is analysed and graphically presented.

- □ In **Chapter 6** a computed torque controller for the Puma 560 robotic manipulator incorporating PD feedback is developed in parallel form, and just as for the simulator, a Transputer system is developed to host the controller, written in Parallel C. The performance of the controller under both optimal conditions and conditions of reduced processing power are considered and the results are grahically illustrated.

- The last chapter, **Chapter 7**, concludes the main observations and findings of the research.

- **Appendix A** contains an evaluation of several of the more prominent parallel processors available.

- **Appendix B** examines some parallel languages. Both popular and some seldom used language are assessed.

- **Appendix C** is a compendium of the graphical data cited in the seven chapters of the dissertation.

CHAPTER 2

PARALLEL PROCESSING - AN OVERVIEW

2.1 THE TECHNOLOGY OF PARALLEL PROCESSING

2.1.1 The Development of Parallel Processing

Before considering how parallel processing can be applied to aspects of control engineering it is wise to first look at the technological developments which have led to parallel architectures being considered for tasks previously implemented by single processor systems.

Figure 1 shows how the complexity of the monolithic integrated circuit has spiraled over the last thirty years to the point of over one million transistors on a single silicon die. This rapid advancement can be attributed mainly to ingenuity in circuit/layout design and decreasing feature size.

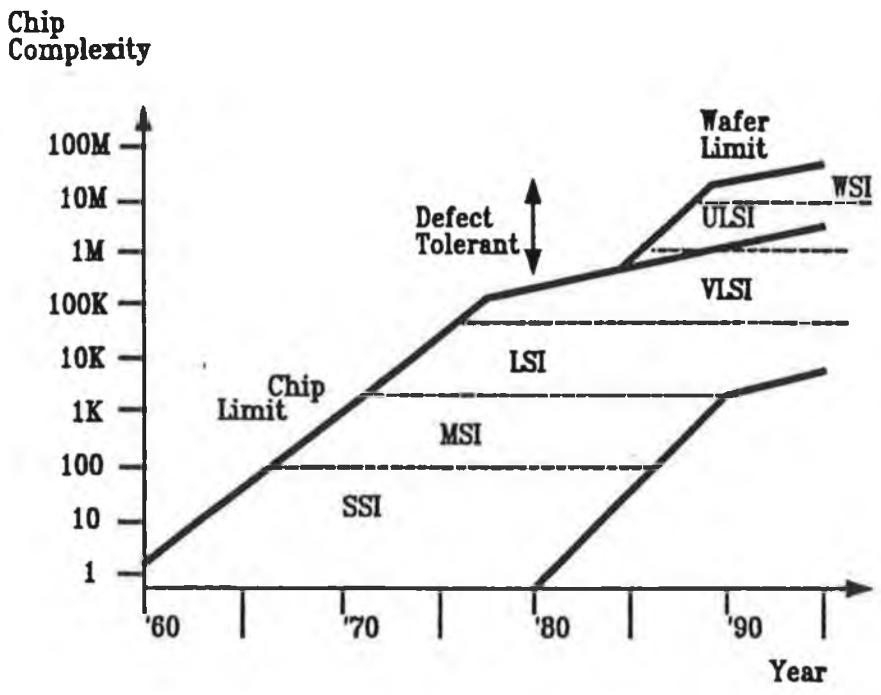


FIGURE 1

Despite this progress it is probable that VLSI packing density will continue to increase at a much more stunted rate [2] due to ever increasing processing costs and lower yields due to higher chip defects. Currently chip sizes are tending towards the 1 square centimetre milestone. But considering that typical defect densities are of the order of 2 to 5 per square centimetre in processed silicon further development is severely hampered. Thus as chip size is increased functional chip yield drops quite dramatically. These considerations bring into question the cost-effectiveness of developing more powerful stand alone processors and give fuel to the concept of concurrent processing.

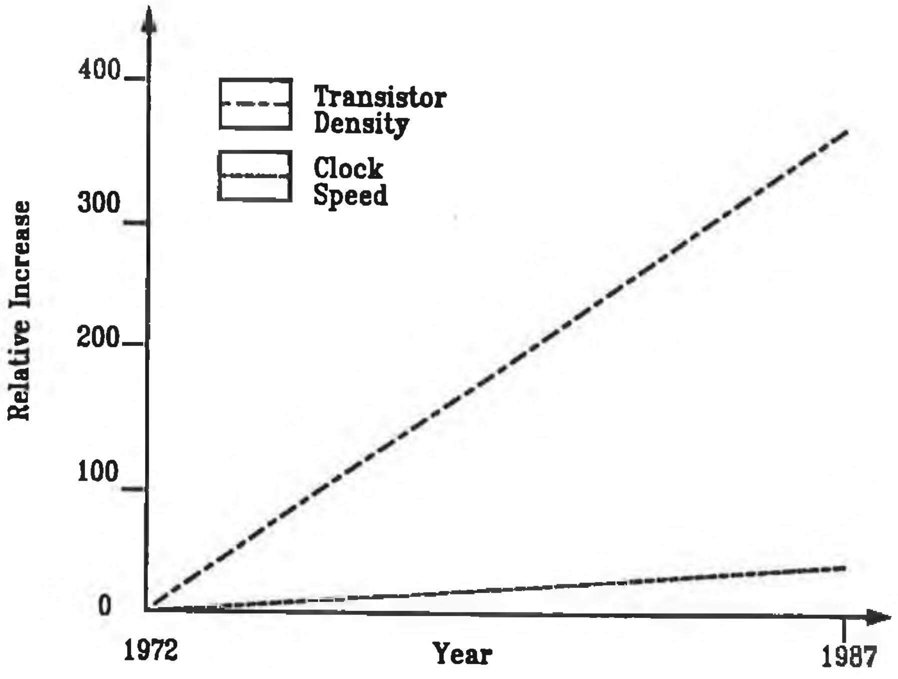


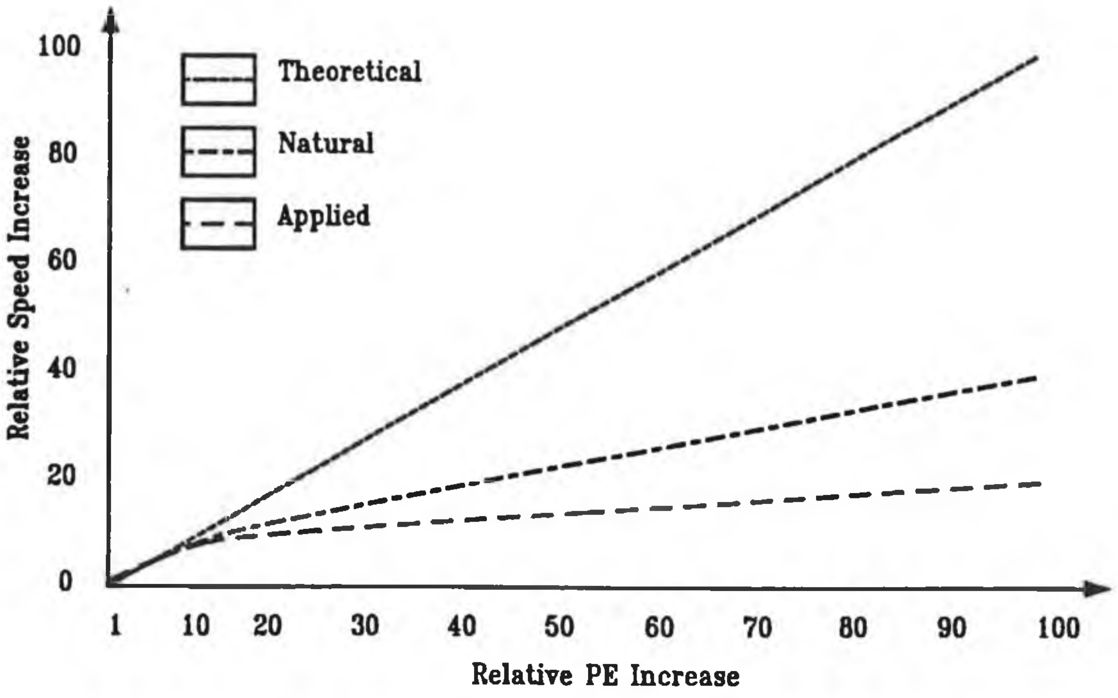
FIGURE 2

Perhaps more important than cost is the concept of performance and it is in this regard that parallel processing offers quite encouraging prospects. Figure 2 shows both the development of transistor packing densities and the increases in clock speeds over the 25 year period from 1972 to 1987. Despite a dramatic 350 fold increase in packing density clock speeds have only had an incomparable 25 fold increase. It would seem that by the law of diminishing returns VLSI/ULSI based single processor systems are reaching their point of expiration with regard to performance capabilities versus costs [4]. Parallel structured systems availing of VLSI/ULSI architectures, of moderate costs and relatively slow speeds operating concurrently can offer a solution to high performance applications.

'Parallel processor' is an umbrella term for a broad range of architectural configurations of processing elements, upon which applications containing some degree of parallelism may be implemented. It is vital that a particular task is realised on an appropriate parallel system so that maximum benefit is derived and that a significant speedup is achieved over the sequential implementation. This is an important point.

Consider a task such as a large matrix multiplication. This involves many multiplications which can essentially be executed separately. Compare this to most statistical problems where preceding data is necessary for current evaluations and likewise current values are needed for future evaluations. This type of problem is inherently sequential since only one evaluation can occur at a time whilst the matrix problem would be ideally suited to an array of simple processing elements. In the two preceding examples the contrast is quite explicit but what is needed is some method of identifying parallelism inherent in a task and how well matched it may be to a particular processor configuration.

Parallelism may be classified into 3 types [5]. The first type is 'theoretical' parallelism and denotes the maximum theoretical speed up of a process when applied to parallel architecture. The second type, 'natural' parallelism, arises from the fact that in a physical parallel system there will always be sections of the task to be executed sequentially and the speed up enjoyed by a process will not be a linear function of the number of processing elements N but rather $O(N/\log_2 N)$. A third and final type of parallelism, 'applied' parallelism, is due to possible mismatch between the architecture and the 'natural' parallelism of the process. This further reduces the speed up to $O(\log_2 N)$. A graph of speed up versus number of processing elements for the average of each type of parallelism is contained in Figure 3



Theoretical, Natural & Applied Parallelism.

FIGURE 3

Based on these observations several approaches may be adopted as regards the type of parallel processor to be chosen for a given task. Looking at the theoretical curve it is noted that by tailoring an array of parallel processors to a specific application an almost linear speed up may be achieved in theory. By using a low number of relatively powerful processors operation is confined to a region close to the origin where again the relationship is almost linear. This approach is termed medium\coarse-grain parallelism. Conversely the use of many cheap processors can be very cost effective per processing element and technically this is termed fine-grain parallelism.

The above approaches to processor organisation provide a good means of architecture classification but a more broad ranging classification can be derived from 'Flynn's taxonomy' [7]. It divides systems according to whether they have multiple(M) or singular(S) data(D) and instruction(I) streams as follows ;

SISD	...	Von Neumann Architecture
SIMD	...	Array Processors
MISD	...	Pipeline Processors
MIMD	...	Multiprocessors

These different forms of parallel processors are explained in more detail in Section 2.2 and appendix A contains an evaluation of various multiprocessor systems. In appendix B several parallel languages, including parallel forms of some conventional languages, are outlined and their methods of inter-process communication are given.

2.1.2 Technical Issues

2.1.2.1 Memory

The most prominent dilemma in designing memory structures for a parallel system is the dichotomy which arises between local and global memory. Should all variables be kept at a location accessible to all processors, should each processor possess its own private store or if both, how does one globally update a variable which appears in more than one location?

There are many other issues which arise in relation to the handling of variables and stored information which several processors may wish to access simultaneously, but the most important is ensuring memory coherence. To avoid large communications overheads and bus bottlenecks it is common in multiprocessor systems to assign each processor its own memory cache[8] for use in localised calculations. Aside from this localised memory there is generally a global memory, which is accessible from any of the system processors, although not necessarily all at once. Within the memory caches a copy of a variable may exist. The problem arises when a processor updates or modifies any such variable and this change needs to be reflected throughout the whole parallel processing system.

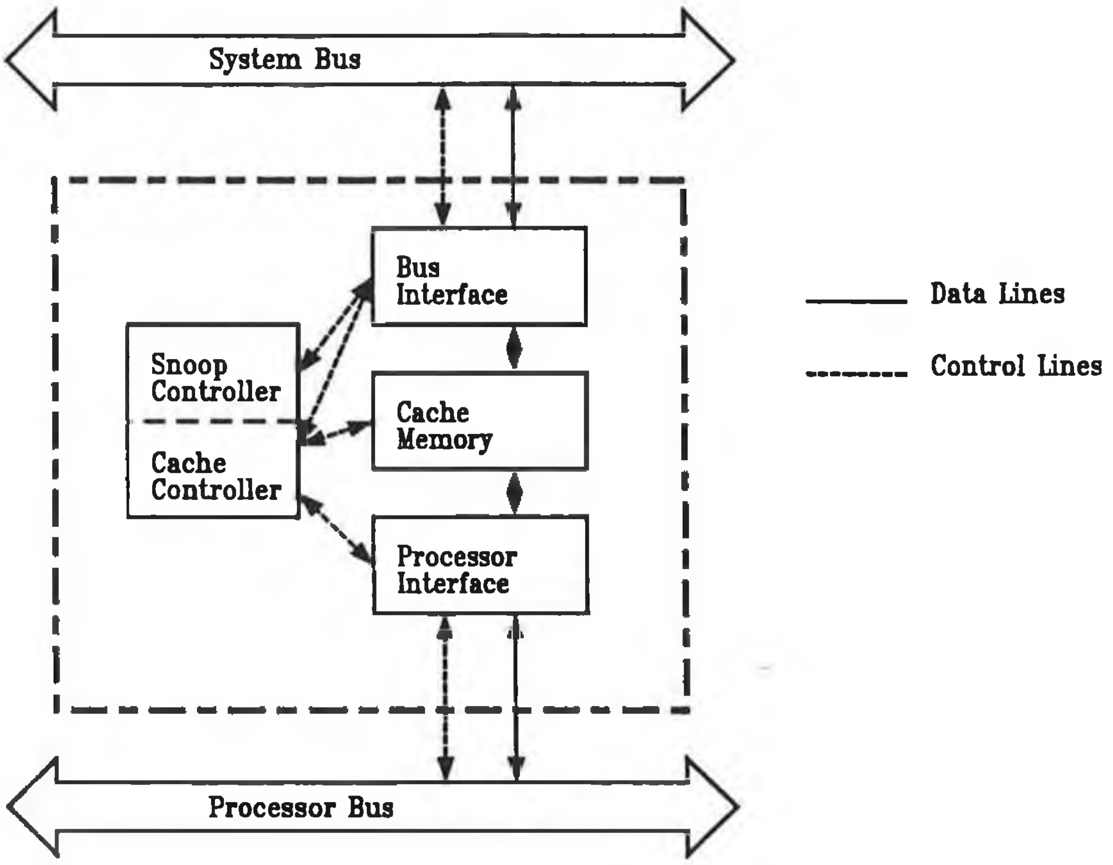
The first objective is to modify the copy of the variable stored in main memory. There are two better known methods for achieving this. Any changes made may be transmitted directly to memory, known as the *Write-Through* method or at a later stage the cache can send a batch of changes to the main memory, otherwise known as *Write-Back*. The actual method used depends on costings which consider main memory traffic, additional logic to cache, buffer space and reliability.

When a variable is updated every processor must have access to the new value and again there are several ways to achieve this. A direct approach is to simply broadcast the updated variable to all other processors, whereby a positive search for the variable results in the data being updated. Another solution is to designate the necessary variables as uncacheable and only available from main memory. This is a software solution to the problem and will be dealt with in later section. A third solution involves keeping a centralised directory of all main memory lines and ensuring that no lines are write-shared.

2.1.2.2 Buses for Parallel Processing

Although recent developments have seen the introduction of more powerful buses, such as the VME bus and the IEEE Fastbus in order to overcome deficiencies in previous buses, they are still not considered adequate for powerful parallel processing systems which utilise global memory as well as local caches. In response to this the IEEE, for instance have set up a new committee known as the P896 Standards Committee to develop the *Futurebus*[9] - a bus intended to address this imbalance. The Encore Computer company have also begun to tackle the problem and are currently developing a high performance bus known as the *Nanobus*.

The P896 working group of the IEEE have, based on the assumption that buses designed for parallel processing must incorporate features and facilities to support cache consistency protocols, begun work on defining a class of compatible protocols for this same cache consistency. The objective is therefore to allow boards from any vendor which adhere to the standard to be incorporated into an existing system with cache consistency guaranteed. The *Futurebus* will display the features of cache consistency. One aspect of the *Futurebus* is that all caches not accessing the bus will in fact be constantly monitoring it, which is termed *snooping*. Address cycles are broadcast to all nodes so that all caches participate in the address-time handshake. The address-time handshake involves a single bus master issuing an address and an address strobe. The address continues to be asserted until all nodes on the bus signal that they are no longer in need of the address. Before allowing the address cycle to continue to completion a cache must check the address to see whether it is in its directory or not. Figure 4 shows the outline of a data cache chip and how this interfaces to the bus. The chip is partitioned into sub-sections, bus interface, cache memory, processor interface, and the two controllers - snoop and cache. The cache controller controls the read and write on the cache and the snoop controller spends most of its time monitoring the bus and also implements most of the cache consistency protocol.



Data Cache Chip Block Diagram

FIGURE 4

As regards the write-through scheme described earlier only the *Futurebus* and the *Fastbus* have an adequate broadcast facility to support such a scheme, and only the *Futurebus* can support the write-back scheme.

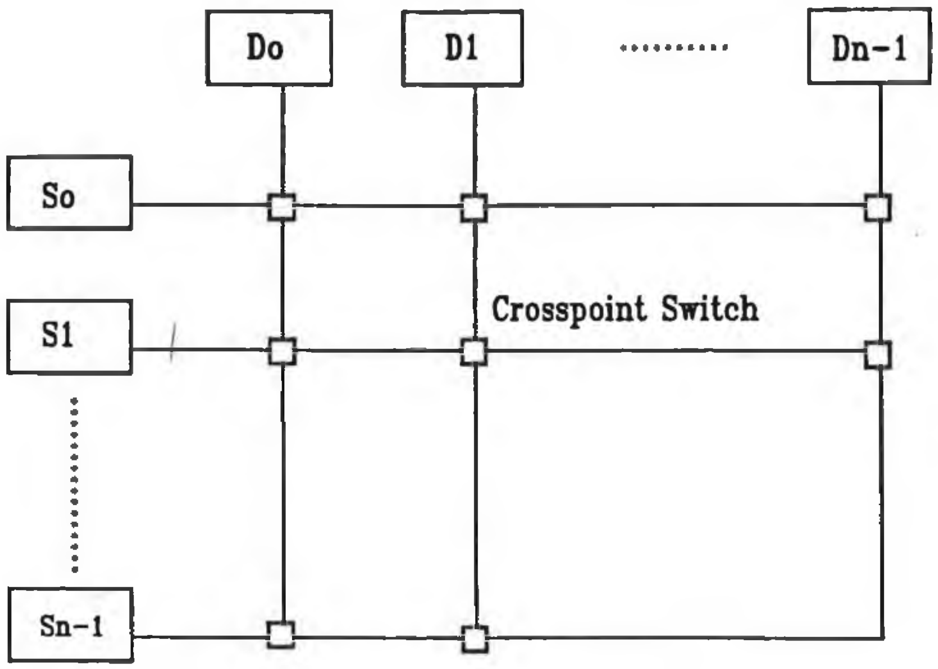
Another major concern in designing buses for parallel processing systems is known as the clock latency problem. Processors within a multiprocessing system generally spend most of their time accessing their local cache and relatively less time accessing the bus. Thus it is imperative that the processors be optimised to their local resources rather than to the system bus. An implication of this is that the processor clock speed is chosen to match that of the cache access time (typically 20/25 MHz), and will be different and asynchronous to the bus clock. Therefore when accessing the bus the processor must synchronise its clock. Two problems arise from this - namely, the delay in waiting for a valid clock pulse and the metastable state problems arising from the synchronizer circuits. This delay is defined as the clock latency problem and will be a half a cycle on average. Using an asynchronous bus solves this problem but synchronous buses are easier to design and are more reliable in the long run.

2.1.2.3 Architectural concerns

Section 2.2 illustrates and discusses the various categories of parallel architectures. It classifies the architectures based on the nature of the input data stream, data handling and more importantly on the relative processing power of the individual processing units. Conversely, this section looks, not at the classes of architectures but at the issues which arise in the development of a parallel architecture. Both memory and bus performance are two important architectural concerns, so much so that they are both treated separately above.

A vital part of a parallel processing system architecture is the communications interconnection network. Naturally enough the communications subsystem, consisting of links between processors, memory modules and I/O controllers is one of the fundamental blocks of an architecture, especially a parallel architecture and the efficiency of the system as a whole depends on an adequate communications implementation. It also has an influence on cost, capabilities and the size of the system. The communications requirements of an architecture can be solved in two ways: either a bus structure or a network.

If a system contains less than about 50 processors with each processor possessing its own cache memory then a high-bandwidth communications bus should prove adequate for inter-device communications. The presence of a local cache at each processor is necessary to reduce the number of bus access requests. Alternatively if the system contains a larger number of processors then a communications bus is no longer feasible due to unacceptable delays arising from an increased number of conflicting bus access requests. The solution to this bus bottleneck is to avoid using a bus altogether and to employ a communications network to implement the communications requirements. In the extreme case a cross-bar network can be used, but this method is quite complex and costly and is generally prohibitively so for very large systems. Figure 5 displays the cross-bar switching network technique for N sources and N destinations.



Crossbar Network Connecting N Sources
To N Destinations.

FIGURE 5

Several issues arise in the cost-effective implementation of a communications network. The first issue is the analysis and evaluation of the communications sub-system topologies of buses, cross-bars and multi-stage networks. In general all networks for parallel systems can be built from four basic types of simple networks. Further to this, interconnection network control is another issue to be considered. This involves communications protocols, modes of operation and techniques for routing messages, including switching. In order to effectively satisfy these requirements and for accurate analysis one must first understand the mathematics of formal specification of interconnection networks. Such an analysis is beyond the intended scope of this report.

2.2 PARALLEL PROCESSING STRUCTURES AND NETWORKS

2.2.1 Classification of Parallel Processing Architectures

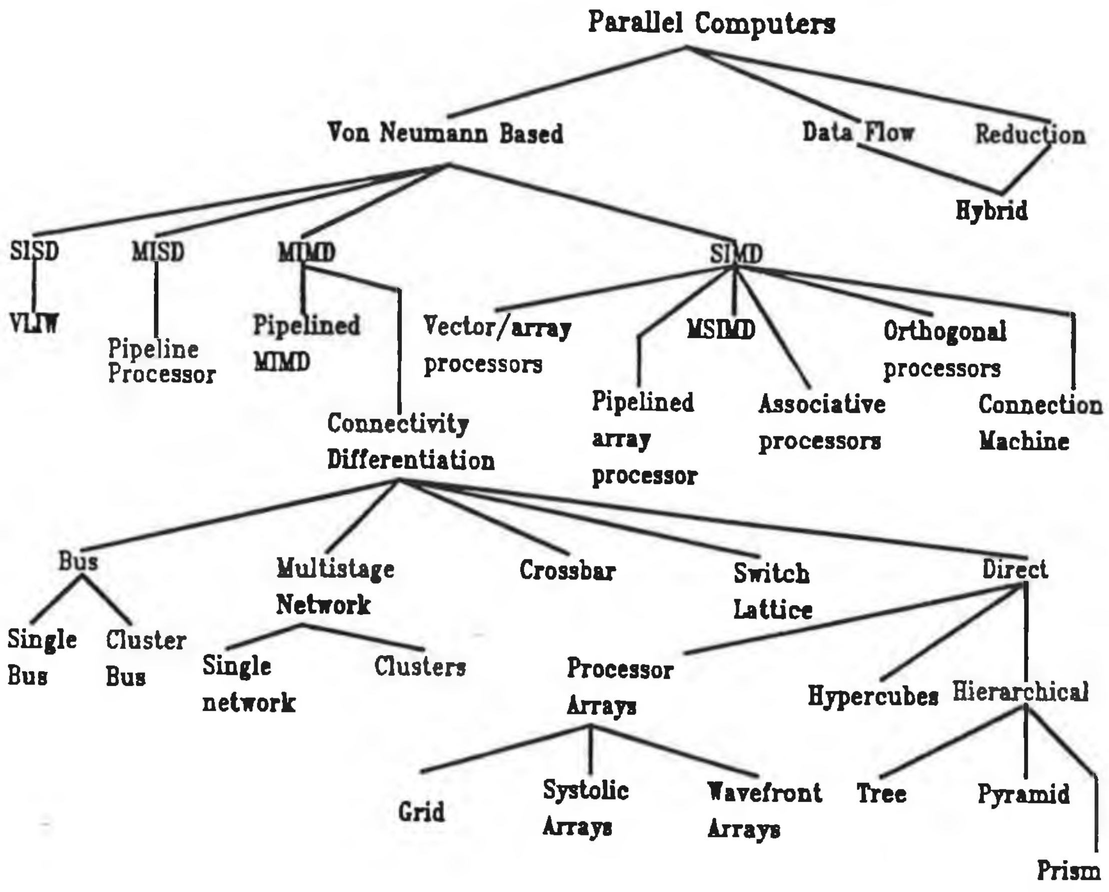
The rapid development of parallel processing technology in recent times has meant that a bewildering range of processor configurations has arisen with very little attempt at standardisation. There has been emphasis on specific parallel processing solutions for specific problems. Currently though, there is a much more concerted effort to design general purpose parallel processing to deal with more general tasks.

This has prompted investigation and research into parallel hardware and parallel software organisation [10,11,12], in an attempt to make the technology more transparent to the user and independent of the problem to be solved. Whilst neat and appropriate classification is not possible it is possible to give generalised groupings of parallel processors based on the nature of their data, their data handling techniques, and instruction set among other aspects of operation.

Parallel computation techniques may be classified into three types[13]: *von Neumann* based, the *dataflow*, and the *reduction* approach. There is a fourth technique, epitomised by the *REDIFLOW* machine of the university of Utah[14] but it is merely a hybrid of the latter two types.

The von Neumann method of exploiting parallelism involves the combination of two or more von Neumann type processors, thus allowing concurrency in their operations. This method follows the traditional fetch-execute-store cycle and uses global and possibly local memory. The second technique, the dataflow method is based on the concept of executing instructions as soon as the operands become available, thus the method is often termed data-driven. Alternatively the method of reduction executes instructions only when the results are needed for other calculations, hence the optional name of *the demand-driven* technique. Calculations are only performed when they are needed which contrasts with the dataflow technique of performing calculations whenever the operands are ready. Finally the last system of parallel computation, the hybrid method is worth a quick mention. In this approach instructions are executed if they are needed provided their operands are available. If not available the responsible instructions are calculated in a hierarchical fashion. This puts sensible requirement-specific constraints on the sequence of execution of instructions and avoids the uncoordinated stream of actions, characteristic of the dataflow style.

Figure 6 illustrates the aforementioned categories of parallel computation and their progeny types. As can be seen the von Neumann standard is the largest class, and is by far the most widespread in use in parallel processing systems, mainly due to historical factors. According to how this class implements the instruction stream and processes the data stream there are four sub-divisions: single instruction single data (SISD), single instruction multiple data (SIMD), multiple instruction multiple data (MIMD) and multiple instruction single data (MISD). One would normally classify the normal von Neumann architecture as SISD and non-parallel but by deft interconnection of multiple von Neumann processors a parallel processing system can be created. The technique is known as horizontal microprogramming and control of the processors may be done by use of a Very Long Instruction Word (or VLIW). The MISD architecture involves a data single stream be processed progressively through a string of processing units or a Pipeline as it is called. The delay in processing one piece of data from entering the pipeline to exiting the pipeline is equal to the total delay of each processing unit, if inter-processor communication is negligible. The delay between consecutive outputs from the pipeline though, is equal only to the processing delay of the slowest processing unit in the pipeline. The pipeline, along with other common and not-so-common architectures are outlined in the following section.



Classification of Parallel Processing Architectures.

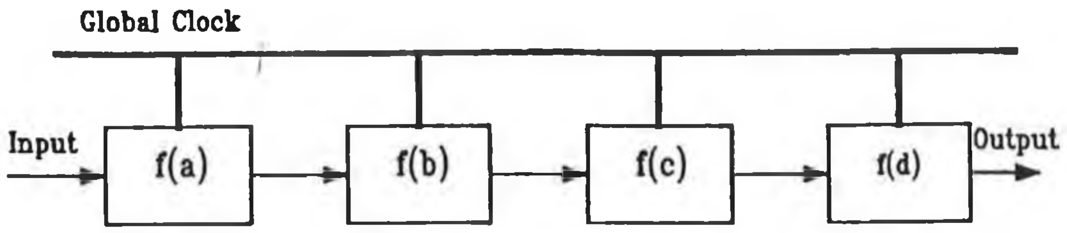
FIGURE 6

2.2.2 Several Parallel Processing Architectures Examined

The following sub-sections illustrate selected architectures, both common and uncommon, both developed systems and systems at the research stage. It is intended to give an overview and a taste of the diversity that makes parallel processing such an exciting research field. Despite the range of parallel processing configurations they are all unified by the one common objective of enhanced performance.

2.2.2.1 Pipelines

Of the early parallel architectures the pipeline [15] was the most common. The concept behind it is in fact derived from observations of a sequential system. In a sequential computer on average only 10% of the circuitry is active at any given time. It is practical therefore to conceive a system in which a process is split into sub-processes which receive information from their predecessor and pass information to their successor.



4 Stage Pipeline Structure

FIGURE 7

Figure 7 illustrates this concept. Based on this principle all components in the pipeline are active at once.

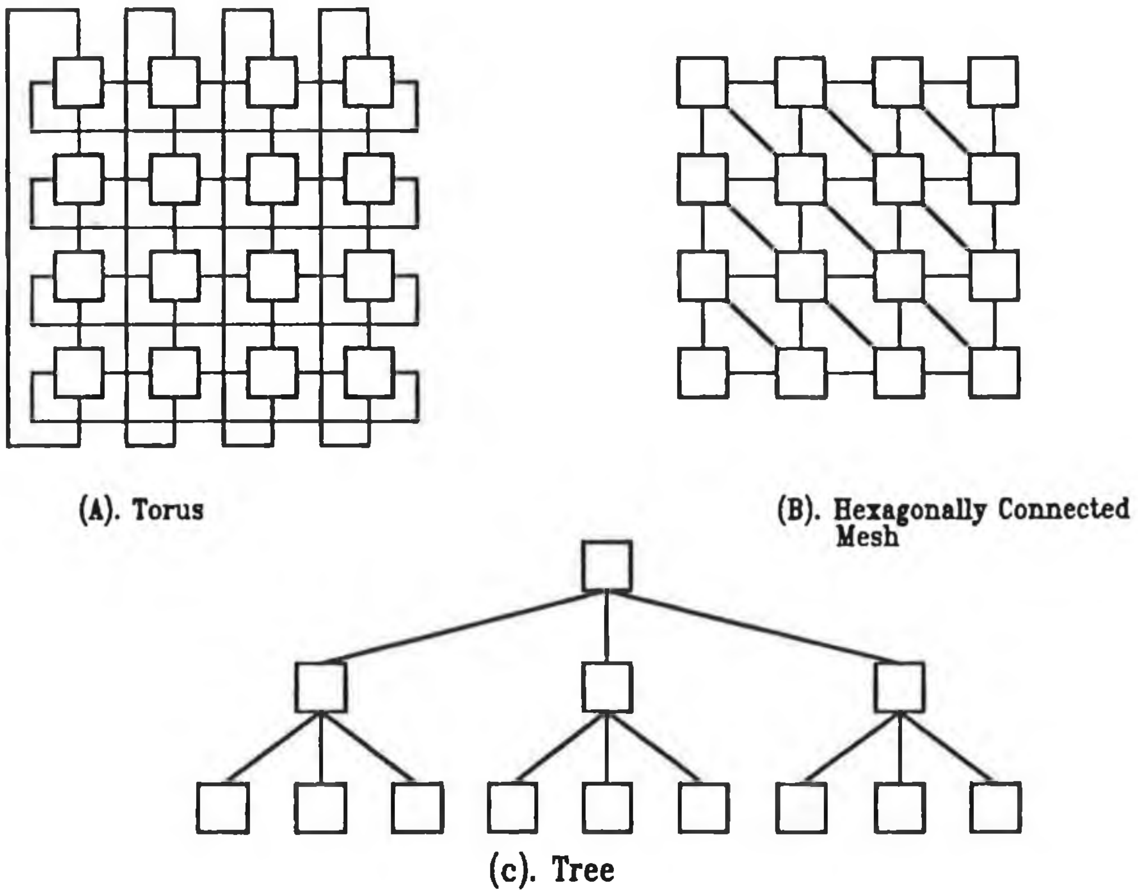
If each element in the pipeline is small and simple, as is often the case, then high clock speeds can be supported. The overall speed of the processor is dependent on the delay of the slowest processing stage and consequently the cost of speed improvements to the pipeline is paid for by an increase in the delay for an item to be processed.

In Figure 7 the data is initially fed into the first pipeline stage, f(a). Subsequent to being processed by this stage the processed data is transferred to the second stage, f(b). At precisely the moment following this transferral the next piece of data is fed into the first stage, f(a) and so on until every pipeline stage is actively processing data. At this point on every clock cycle an item of processed data leaves the last stage as output, and a new piece of information enters the pipeline as input.

2.2.2.2 Systolic Arrays

A network of regularly connected processing elements constitutes what is called a systolic array [16,...,25]. Figure 8 shows three different types of systolic array. The pipeline, fig 4.a, is in fact a particularly simple systolic array. The reason for the regular connectivity of the processing elements is explained by the fact that the the problems most suited to systolic arrays are themselves quite geometric, such as multiple matrix operations. In 1978, Kung and Leiserson [26] proposed a multiprocessor architecture which could execute multiple computations concurrently and it is upon this model that systolic processing partially originated.

The processors are connected together in a rectangular lattice. Because the data is pumped through the lattice in a fashion not unlike that of the heart the structure is known as a 'systolic' array, from the Greek word for contraction. As can be seen in Figure 8 many other connectivities have evolved since. Systolic arrays are now being applied to quite a variety of tasks such as adaptive beamforming [27,20], Kalman filtering [21], and matrix operations in control algorithms [21]. The particular array structure used in an application depends wholly on the nature of the operations performed. The processing elements receive data from their neighbours, process them and pass them on to consecutive processing elements in a wavelike fashion. Data does not necessarily travel in only one direction but may have several paths across the array.



Three Types of Systolic Arrays

FIGURE 8

Due to improved VLSI technology the growth in systolic array usage has been quite prolific. The next era of array implementation is the programmable systolic array and already INTEL have developed the WARP and the I-WARP [28] machines which can be programmed into a particular configuration. One of the main drawbacks of systolic arrays is that generally there is a relatively long through-put delay which restricts their usage to certain applications.

2.2.2.3 Neural Networks

Often overlooked as a means of parallel processing neural networks [29,...,34], may offer very pragmatic solutions to a range of control problems. Already they are being used to control robots in an 'intelligent' fashion [29] but despite this they are still at a very pioneering stage.

Neural networks, so called because they are modelled on the neuron structure of the human brain, differ dramatically from conventional electronic hardware. They consist of many model neurons interconnected in 3-D by paths of varying strength and employ graded signals for communication. In order to comprehend how a neural network can actually support an algorithm or make decisions it is imperative that the human nervous system must first be explained. The object of 'computational neurobiology' [31] is to understand how the brain receives data through the senses and processes it into conceptual information. For example, in the human visual system information is received through the eyes by light waves stimulating the retina which is then brought to the brain via chemical ions. The brain then proceeds to extract concepts of colour, depth, motion ...etc. from the observed scene which can be expressed by the human. It is widely accepted that even this simple human faculty poses a huge problem for conventional digital technology [35] to emulate.

In a conventional computer most of the silicon is inactive at any given time. Usually only the CPU and perhaps a few bytes of memory are active. Contrasting to this is the Connection machine[36] (a type of neural network developed by MIT) where most processors or neural nodes are active together, or concurrently. On the simplest level, the brain functions as follows : The 'firing' of neurons is activated (or inhibited) by other connected neurons. Thus whether a neuron is firing or not depends on the inhibitory or excitory inputs from all other neurons connected to it. How these operations combine to form valid thoughts and constitute huge amounts of memory remains mostly unknown.

The aforementioned application of a neural network by Guez et al[29] to adaptively control a robot provides a very interesting insight into the potential options that these networks offer control engineers in the field of parallel processing. The major benefits claimed by Guez are an Adaptation rate for finding optimal parameters that is faster than Model Reference Adaptive Control (MRAC) or Self Tuning Regulator (STR) methods, a simpler controller structure for multiparameter adaptive control (i.e. complexity of controller does not grow exponentially with number of unknown parameters) and finally adaptation is possible over both the discrete and continous domains. These are no small claims to make in the field of robotic control. Despite this though, neural networks have yet to develop as feasible alternatives to conventional processing architectures.

2.2.2.4 Multiple SIMD Computer Organisation

Two or more control units (CU) which share several processing units through dynamical allocation constitute a multiple SIMD system. The processors are allocated to perform specific SIMD tasks. This type of system is not common and is to great extent still at the research stage. The PASM machine[37] of Purdue University is apt example of this class, and is a reconfigurable parallel processing system that consists of a number of SIMD and MIMD sub-systems executing separate tasks. One of the important requirements of this class of multiple reconfigurable SIMD systems is that each subsection created by partitioning the overall system must be capable of operating independently, but must also possess the same capabilities as the original system. Independent operation defines that no overlap in data handling should occur. The issues involved is this specification are job-scheduling, resources allocation, and load balancing for effective and efficient operation. These are items which are catered for in the operating system.

2.2.2.5 Hypercubes

This is another of the more interesting and novel architectures for parallel processing. A hypercube (or binary n-cube) is basically a loosely coupled multiprocessorsystem consisting of N (= 2^n) processors interconnected as an n-dimensional binary cube. Each processor in the hypercube has its own non-global main memory and CPU. Every processor is interconnected via the communications network to n other neighbour processors. Additionally, the address of each processor differs from its neighbours by one digit, using an n-bit addressing scheme. Based on these properties it can seen that the trivial case of a 0-dimensional hypercube is just a conventional SISD computer. There are several reasons why hypercubes are very suitable to general parallel processing implementations. The forte of these systems is that they are topologically ideal for the mapping of multi-dimensional meshes and trees: neighbouring nodes are mapped onto neighbouring processors, thus yielding minimum communications overhead. Many scientific applications use mesh-type recursive solutions, such as the FFT, Potential gradient, Fields and Bitonic sort routine. Hence the suitability of the hypercube. Further, the hypercube has a maximum internodal distance of $\log_2 N$ hops, thus allowing for relatively fast communications.

2.3 THE LANGUAGE OF PARALLEL PROGRAMMING & SOFTWARE ISSUES

2.3.1 Specifying concurrency

When a sequential programme is written it specifies a sequence of tasks to be executed in an ordered and serial manner. Inherent in this list of operations may be instructions which can be executed independently of one another, or indeed concurrently. The dilemma of parallel software is to determine this 'parallelism' and maximise the speedup benefit in doing so[38]. There are two ways in which parallelism may be exploited. The first is to produce a sequential programme and to compile it with a compiler that examines for parallelism. The second is to rewrite the programme using a parallel programming language, and to specify the concurrency. In both methods the maximum exploitation of concurrency is not guaranteed, and in fact one can never tell if a particular implementation gives the maximum speedup over the sequential case except in trivial instances of small programmes where this can be done by inspection. There are no efficient compilers which can ensure that a sequential programme will be converted into a very efficient parallel implementation. The second method above of user inspection is obviously open to human error but at least there is the flexibility of organising the software according to the hardware so that an appropriate mapping is ensured.

In a parallel programme at one or more instances two or more processes are specified to run concurrently. In order for this happen without integrity failure these processes must communicate and be synchronised at various points in time. A number of language constructs have been defined to implement this. The following sections examine some of them.

2.3.1.1 Synchronisation Based on Shared Variables

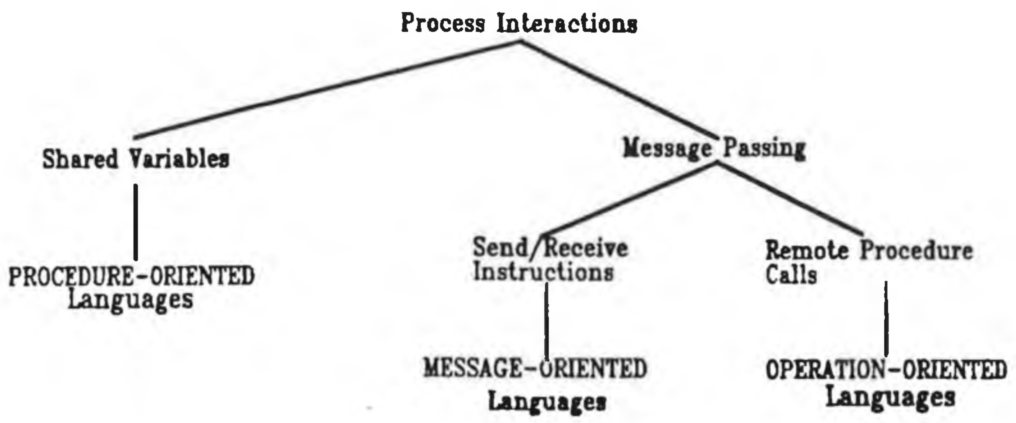
In using shared variables two types of synchronisation occur: Mutual exclusion and Condition synchronisation. In mutual exclusion there is the mutually exclusive execution of critical sections, where a critical section is a sequence of statements that must be executed as an individual operation. Condition synchronisation is where a shared variable is placed in a state whereby it cannot be accessed for operation until the process which is using it (and possibly changing its value) releases it.

2.3.1.2 Synchronisation based on message passing

In this case processes send and receive messages rather than use readable and writeable shared variables. There are two main techniques- channel naming and synchronisation mechanism -which may be implemented in many ways. This makes this method an attractive solution to programming language problems in this regard. A programming notation based on this idea, Communicating Sequential Processes (CSP), is the foundation upon which many modern parallel languages have been devised, such as Occam. The Occam language along with several other parallel programming languages are outlined and examined in appendix B.

2.3.2 Classification of Parallel Programming Languages

In Section 2.2.1 an attempt was made to classify parallel architectures according to certain criteria. Likewise in this section parallel programming languages will be categorised, but unlike the architectural classes the languages are easier to classify because of the explicit nature of language constructs. There are basically three types: Procedure-oriented, Message-oriented and Operation-oriented languages.


Process Interactions
Shared Variables
Message Passing
PROCEDURE-ORIENTED Languages
Send/Receive Instructions
Remote Procedure Calls
MESSAGE-ORIENTED Languages
OPERATION-ORIENTED Languages

Synchronisation Techniques & Languages Classes.

FIGURE 9

Figure 9 shows the genealogy of the three types. As can seen message-oriented and operation-oriented appear on different sub-branches of the same branch classification. They both use message passing, but implement it differently. Message-oriented languages utilise "send and receive" primitives, whilst operation-oriented provide remote procedure calls as the technique for interprocess communication. The final type, procedure-oriented languages utilises a method of writing individual programmes for each processor in the system, and treating inter-processor communications as a shared object. The advantage of message-orientation is that the existence of a communications network is transparent to the user, and shared memory is optional. Some operation-oriented languages have the characteristics of both message-oriented and procedure-oriented languages. Such languages include Ada[39], Occam[40], Linda[41], and Sisal[42].

2.3.3 Parallel Programming Supportive Operating Systems

The actual services required of an operating system by parallel programmes are essentially the same as those existing in conventional sequential systems, but the manner of provision is fundamentally different. It must be ensured that the way in which services are provided helps to balance the workload of the individual processors. During this coordination of facilities the operating system must avoid critical serial sections, since these would cause a significant rise in performance cost (rises linearly with the number of processors involved). Essentially the operating system must itself be a parallel programme, with decentralisation and be bottleneck free.

Oneof the more difficult tasks to be performed involves allocating and scheduling the processes to the processors so that execution time is minimised and thus speedup over the sequential case is maximised. This is closely linked to the concept of programme partitioning. The overall problem is phrased as follows: Given a parallel programme it necessary to partition the programme into a set of communicating processes that must then be allocated or scheduled on a set of processors. Intuitively it can be seen that partitioning is a optimisation problem, due to many possible choices of partitioning schemes.

Another important aspect of operating system performance is in the area of resource management. In conventional centralised systems there are tables which give up-to-date information on the status of all resources being managed. In parallel systems with distributed control this information is not available and is currently a research topic and a number of algorithms are being developed for dynamic storage allocation[43].

As mentioned an operational system is required to determine correct load balancing for a processing system. There are two ways in which load balancing may be implemented: Static[44] and Dynamic[45]. Static load balancing involves decomposing the code before execution, assigning tasks to processors and then leaving the system to proceed without further interference. Certain problems have fixed structures which are very suited to this kind of decomposition. The second method, as its name suggests the processors are dynamically assigned workloads depending on varying conditions. This occurs in event driven and conditional systems such as simulations. Chapter 3 deals with the concepts of partitioning and scheduling more extensively and objectively. Currently under research is the concept of self-scheduling for processors for DO loops[46] which allows for load balancing when the execution time of each iteration varies significantly due to conditional statements. Most of these schemes are nonpreemptive, meaning that once a processor is assigned to a task it will not be preempted until it has completed that task. Avoidance of deadlock is discussed in a paper by Tang, Yew, Fang, and Zhu[78].

2.4 APPLICATION OF PARALLEL PROCESSING TECHNIQUES TO CONTROL ENGINEERING

The concept of parallel processing is not a new one, but the application of it to control engineering tasks is a recent development, made possible by contemporary hardware and software innovation. It has long been accepted that, if properly orchestrated several processors/CPUs when combined can form a powerful processing entity. What prevented this from being implemented in commercial systems was the adequacy of the microprocessor for most tasks and hence the expense of a multi-processor system was simply not justified. With the advent of high demand systems, such as highly fault tolerant flight controllers, parallel processing became a viable option.

With specific regard to control engineering parallel processing offers many exciting prospects. It seems quite inevitable that future trends will tend towards a higher degree of parallelism in both hardware and software implementations of control laws. In order to gain maximum benefit for a particular control law its structure must be examined for inherent parallelism, and when this has been identified suitable hardware can then be chosen (of course, this is the ideal case and it is quite unlikely that the maximum speed-up could ever be achieved).

How do we examine for parallelism ? It would be desirable to have a high level compiler which could take in a control law and decompose it into concurrent processes. Such a compiler would interface between the mathematical model of the algorithm, in the programmer's head, to the actual parallel processing hardware, in the programmer's hand. Desirable as it may be, it has yet to be developed.

At present, the programmer translates the particular algorithm to be implemented into a high level parallel programming language, such as OCCAM, which then compiles the code into a form executable over a parallel system. The significance of this is that it is the programmer who decomposes the algorithm into concurrent processes because in all parallel languages every process must be defined and similarly the concurrency of these processes must also be specified. Therefore, the efficiency of the parallel implementation depends on programmer skill, and further, failure to recognise parallel operations will manifest itself as less inefficient coding.

The automated software interfacing of control laws onto parallel systems has remained somewhat of an impasse. There are no software compilers at present which allow a programmer to specify a control law in pure mathematical terminology and then decompose it into a flow diagram of concurrent processes which may then be implemented on, say, a transputer system. There are several parallel programming languages with which a programmer can generate parallel processes but in order to realise a control algorithm in parallel the programmer must have intimate knowledge of the algorithm. Therefore, efficiency is based on the ability of the programmer to recognise inherent parallelism. Eventually high level compilers will be developed which will translate completely from the purely mathematical model of the algorithm/control law to code executable on the particular parallel processing system present, in a manner transparent to the user.

As would be expected, because of the absence of any general purpose 'control law parallel compiler' most research interests have dwelled upon application of a particular parallel processing technology to a particular control task or algorithm, like for example 'Kalman filtering on a systolic array'[47] or 'target tracking using transputers'. Nonetheless, it seems only a matter of time before parallel processing becomes an integral part of real-time control engineering technology. In the following sections several instances of the successful application of parallel processing to a control task are examined. The fact that these examples are quite specific, and lack generality is more a reflection of the current state of parallel processing in control rather than an oversight.

2.4.1 Parallel Processing for Robotic Control

In a brief paper by Shaheen Ahmed of Purdue University[48] a multiprocessor scheme is proposed for robot control. He mentions that some modern microprocessors can accomodate a floating point instuction every 100ns, thus making it possible for the complete inverse dynamic and kinematic computation every sample period. Coupling of these microprocessors into a multiprocessor environment can create the potential to solve rather complex control problems. A coupling arrangement such as this would have the advantage of being reconfigurable to accomodate a new control algorithm. Their optimal operation depends on choosing the correct number of processors and scheduling the workload in a balanced manner. The proposed architecture consists of an Input/Output Processor (IOP), which generates control signals to the outside world; a Scheduling and Processing Unit (SPU) which is responsible for initiating and optimally sequencing the processing units (PU) through a global or private access bus; PU's, which perform the computations as required by the SPU.

Results produced by Ahmed are based on the usage of the Motorola 68881 processor, which has a multiply time of 5.87μs. This proves to be sufficient in a multiprocessor environment. A parallel controller is used to control a manipulator arm. It is found that seven processors can perform the forward kinematic calculations in 80μs and the resulting speedup factor with seven processors is 6.5. In calculating the inverse kinematic equation for the manipulator using five processors, (no increase in speed is gained by adding to five processors), the time taken is approximately 370μs with a corresponding speedup factor of 2.38 over the sequential case. Further to this the inverse dynamics problem may be solved for the arm (6 joints) in 710μs using a saturation limit of seven processors. The speedup over the sequential calculation is 6, approximately. This application serves to illustrate that by incorporating additional processors to implement what has traditionally been a uni-processor task significant benefits may be achieved.

There are various other schemes proposed for the control of robot manipulators by the use of parallel processing techniques[49,50,51] and although differing in decomposition techniques they generally yield speed-up factors of similar order. In chapter 6 an algorithm is developed in parallel form for the control of the PUMA 560 manipulator arm which makes use of p-d feedback to enhance system performance. Also contained in the chapter are indicators as to the effectiveness of the algorithm and speedup indices.

2.4.2 Simulation and Control of a Non-Linear Process using Transputers

One of the biggest obstacles in the usage of non-linear models in the control of highly non-linear systems is the strain on a processing system to solve the non-linear differential equations in real-time. A solution to this problem is proposed by Ponton and McKinnel[52], whereby a transputer system is suggested as the hardware capable of solving these non-linear calculations in the particular case of a distillation column. In order to maximally exploit the parallelism a correct decomposition scheme must be selected. It is claimed that one of the major fallacies of modern process control is that non-linear systems are quite often modeled by linear approximations which generally lead to degraded performance and inefficient process operation[53]. Obtaining the non-linear model is rarely difficult (in fact it is usually more difficult to obtain a linear approximation); the problem lies in the complexity of the implementation. Nonetheless, the implementation of the non-linear model provides much more accurate control signals and generally meets tighter environmental and safety standards.

In the chemical industry the distillation process is one of the most widespread. It involves seperation of various chemicals which have different boiling points, thus the control problem is to ensure an effective purification process and its accuracy determines the quality of the final product. Tight and precise control margins can be reflected as large cost savings, since the amount of energy used in a heating process is usually quite significant. Generally the objective of the control technique is to maintain the 'overheads' and 'bottoms' products at a set-up. Much time and effort has been spent in trying to overcome the problems of controlling distillation processes but very few methods have attempted to make use of direct non-linear control[54]. The strategy proposed by Ponton and McKinnel is quite novel and involves observing that the distillation column can be broken down into smaller sub-units or blocks. Direct assignment of a processor to each block may seem like an obvious implementation but the serial nature of the inter-relationship between blocks means that this would pose no benefit in speedup. Nonetheless, clever rearrangement of the equations allows for a parallel implementation and assignment of a processor per block. In a simplified explanation this involves substitution of the dependent variables with approximations and performing regular updates in data, hence all processors can run simultaneously and need only supply and receive information at the end of each calculation cycle. This information is then collated and used to drive the feedback controller. The transputer was chosen as the parallel processor most suited to the task with its fast serial links. Results show that the convergence of the parallel implementation is faster and as stable as the traditional serial controller[52].

CHAPTER 3

Algorithmic Decomposition and Parallelism

3.1 PARALLELISM AS A CONCEPT

Whilst chapter 2 delved into the issues of parallel hardware, parallel architectures and software mechanisms for exploiting and implementing parallelism, there was no discussion on the actual concept of 'parallelism':- How is it defined? How may it be quantified? How does one reconcile the difference in parallelism of an application with that of the target hardware? This section attempts to formulate and quantify not only parallelism per se, but related items such as the partitioning of programmes, the scheduling of these modules onto multiprocessors and the optimisation of these techniques.

The principle upon which parallel processing exists is that if a programme or, more generally a software system can be decomposed into a number of smaller tasks which may be executed independently then gains in speed and possibly in performance may be made by executing these tasks in parallel. This relative increase in speed, known as speedup[55], is one of the major motivating factors for research into parallel processing, especially in relation to control engineering. As speedup increases the efficiency of the multiprocessing system decreases, due to increased redundancy of processors, and in section 3.1.1 this relationship is examined at length.

It is imperative that before attempting to define parallelism one must first create a notation so that applications/algorithms may be mathematically represented as a series of concurrent and serial tasks. Several basic preliminary definitions must also be made to ensure clarity and generality.

We define, for systems of potentially parallel processes:

A process: A process is an independent sub-unit of a system. For the purposes of later definitions the restriction that a process is a list of sequential instructions is imposed. Processes may or may not be capable of running in parallel, depending on the nature of the system

Totally serial: A system is totally serial if every process must be performed in sequence and each process is in itself inherently serial. This means that at any point in time no more than one process is active.

Shortest path T_s: This is the shortest theoretical computational path from system input to output, derived from the block diagram representation of the system with concurrent processes shown in parallel computational paths.
Longest path (Critical path) T_l: Likewise, this is the longest computational path of the system block representation and is equal to the system execution time. This is a vital index for assessing the speedup of a parallel implementation.

Processor Boundary (P_b): This equal to the maximum number of available independent processors.

Speedup: This is a factor which indicates improvements in speed over the serial implementation of the system.

$$\text{speedup} = \frac{\text{Serial execution time}}{\text{Longest path time}}$$

Maximum parallelism (P_m): The maximum number of processes that are active in parallel during the course of an application.

3.1.1 Unbounded Parallelism

It is now possible to define the ***parallelism*** of a system at a point in time as the maximum number of processes which could possibly be active in parallel at that point in time, assuming unlimited availability of processing units. This definition may be extended to give an average value for parallelism and the eventuality of limited processing power, which is of common occurrence in real world applications where a high degree of parallelism in an application does not necessarily justify a corresponding amount of processing units.

Average (Unbounded) Parallelism (P_{ave}): This defines the average number of processes that are active in parallel during the execution of an algorithm/application, with the provision that the availability of processors is unbounded.

$$\begin{aligned} P_{ave} = \{ \quad & 1\text{x time spent with 1 process active} \\ + \quad & 2\text{x time spent with 2 processes active} \\ + \quad & 3\text{x time spent with 3 processes active} \\ + \quad & \dots + P_m\text{* time spent with Pm processes active}\} \\ \div \quad & \{\text{total execution time}\} \end{aligned}$$

□ Eqn 1a.

Consider, for example a totally serial system with n process. The average parallelism would be calculated as follows:

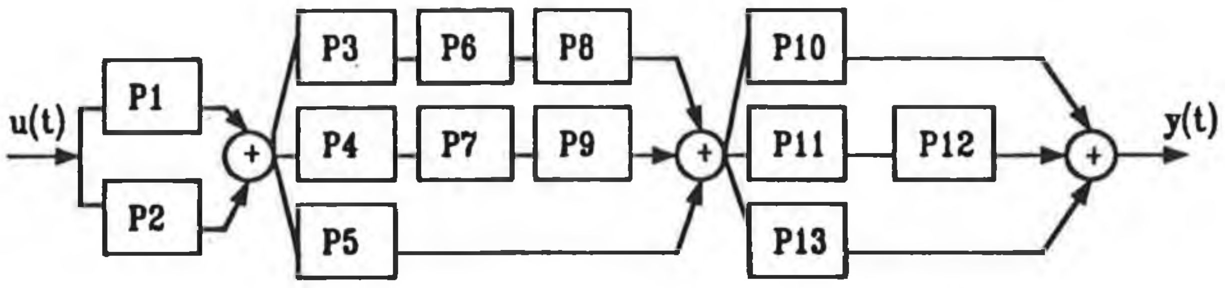
$$P_{ave} = \frac{\{1xnT\}}{nT} = 1$$

To say, therefore that a system has an average parallelism of unity is to say that that system is totally serial, because any instance of more than one process running simultaneously will always guarantee an average parallelism greater than unity.

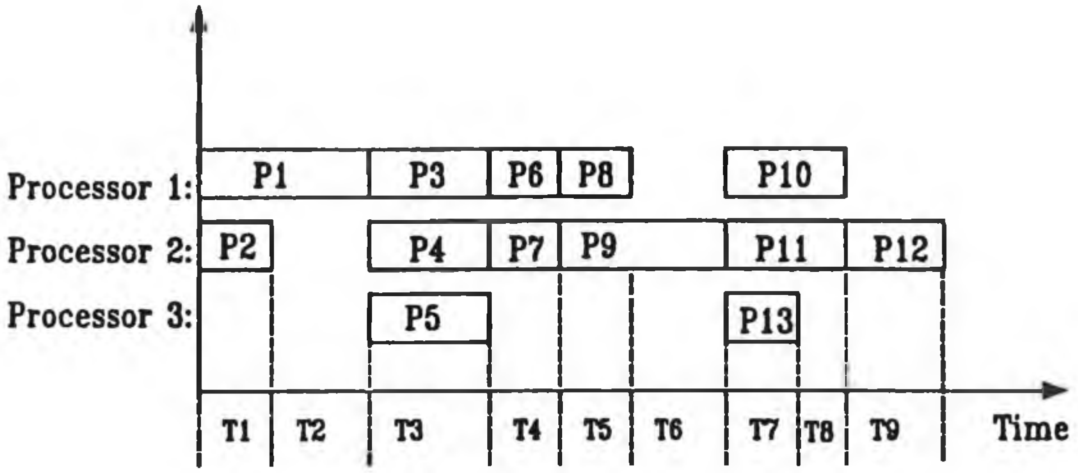
Conversely, a system with n processes of equal run times, which are independent and can run in parallel, will have an average parallelism as follows;

$$P_{ave} = \frac{nT}{T} = n$$

These two extremities in parallel/non-parallel system serve to illustrate the upper and lower bounds on the average parallelism on any system. The average parallelism in general for systems is a lot more complicated to calculate.



(A). Schematic representation of sample system



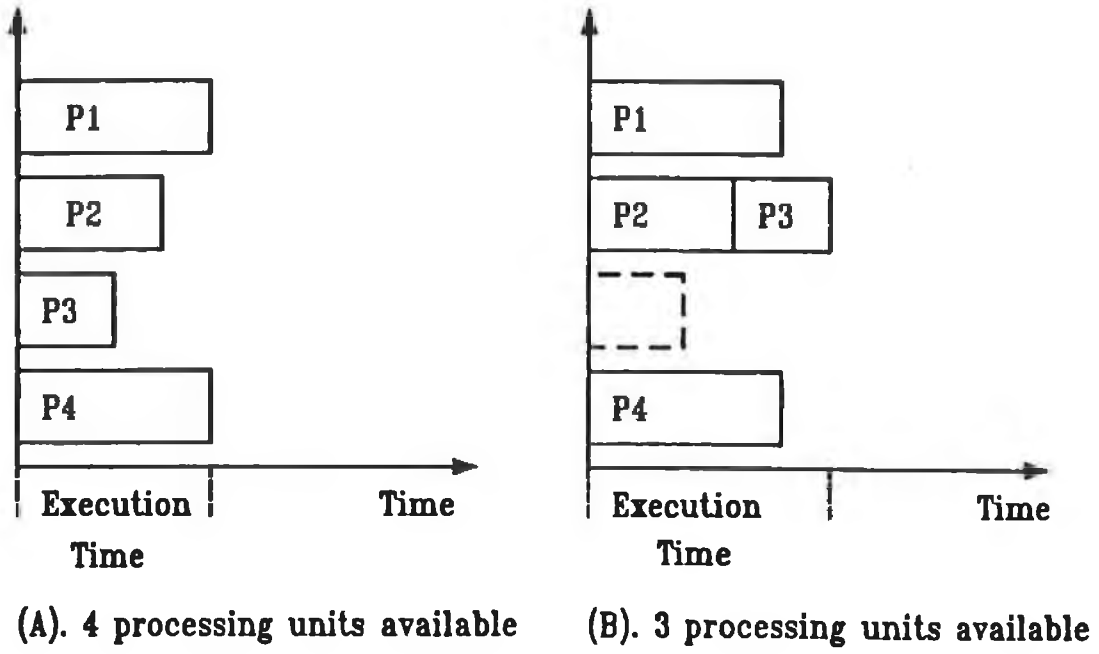
(B). Schedule of processes

$$P_{ave} = \frac{3x(T3+T7)+2x(T1+T4+T5+T8)+(T2+T6+T9)}{T1+T2+T3+T4+T5+T6+T7+T8+T9}$$

FIGURE 10

Graphing is a very suitable mechanism for representing the flow of execution of a system of parallel processes, and helps to clarify the concepts of parallelism and average parallelism, and to simplify their calculation.

Figure 10 shows both the schematic and graphic representation of a sample system of parallel processes. The thirteen processes represent independent tasks, connected by lines which represent the flow of data. The summation block (denoted +) represents a point where the output of preceding processes are collated (not unlike a rendezvous point in ADA[39]) and passed onto proceeding processes which require the data for activation. The graph demonstrates the relationship between the processes in the time domain based on their points of initiation and length of execution time, which are chosen arbitrarily for purpose of illustration. The process blocks are scaled proportionally to their run-times. The expression for P_{ave} is derived by observing the number of processes active over each time interval and collating the total time for each multiple number of parallel processes. For example three processes are active in parallel for a total of (T3+T7) time units.


P1
P2
P3
P4
Execution Time
Time
(A). 4 processing units available
P1
P2
P3
P4
Execution Time
Time
(B). 3 processing units available

Scheduling of processes.

FIGURE 11

3.1.2 Bounded Parallelism

A natural progression on the concept of average parallelism is the introduction of limits on the number of available processors. For example figure 11 shows the effect of reducing the number of available processors from four to three. Instead of executing all four processes in parallel, only a maximum of three may be executed simultaneously. Speedup in the first case is equal to the total time for all processes divided by the run-time for the slowest process. For maximum speedup in the second case the three slowest processes (p1,p2,p4) are initially run simultaneously on the three available processors and on completion of the first process (p2) the execution of the last process and fastest (p3) begins. In order to mathematically formulate an expression for this *bounded parallelism* one must first define what is meant.

Average Bounded Parallelism (P^b_{ave}): The average parallelism of a system of processes, with the imposing constraint of limited availability of processors.

In order to develop this definition in a mathematical sense several assumptions are stated and some notational definitions are given.

* 1.The maximum number of processors available to process in parallel shall be denoted by r.

* 2.The maximum parallelism of the system of processes is equal to n.

* 3.The scheduling of processes is such that if there exists n processes in parallel, where $2r>n>r$ then the r slowest processes are executed first. The remaining (n-r) processes are then executed. If $n>2r$ then after the first batch of the r slowest processes have completed the next r slowest processes are executed. Finally when this batch is complete the remaining (n-2r) processes are scheduled for execution. Likewise for any other n greater than multiples of r there are multiple batches of r processes to be executed.

* 4.Denote the number of batches of x ($x>r$) parallel processes as N_x. This means that on N_x occasions during programme run-time x processes can potentially run in parallel (although this would be prevented due to lack of available processors).

* 5.The following derivations assume that the maximum parallelism, P_m is not less than the processor boundary, P_b. Obviously, if it were then the average bounded parallelism would be equal to the average unbounded parallelism.

* 6.Define $T^{s}_{p,q}$ as the run-time for the sth slowest process in the qth batch of p parallel processes.

Therefore, as an extension to equation 1a it can be stated;

$$
\begin{aligned}
P^{b}_{ave} = \{ & P_{ave}*T_{1(\text{unbounded})} - (r+1)\{ T^{r+1}_{r+1,1} + T^{r+1}_{r+1,2} + \dots + T^{r+1}_{r+1,Nr+1} \} \\
& - (r+2)\{ T^{r+2}_{r+2,1} + T^{r+2}_{r+2,2} + \dots + T^{r+2}_{r+2,Nr+2} \} \\
& - \dots - n\{ T^{n}_{n,1} + \dots + T^{n}_{n,Nn} \} \\
& + (\text{Total run-time of all r+1 parallel process on r processors}) \\
& + (\text{Total run-time of all r+2 parallel process on r processors}) \\
& + \dots + (\text{Total run-time of all n parallel process on r processors})\} \div T_{1(\text{bounded})}
\end{aligned}
$$

□ Eqn 1b.

The next step is to formulate an expression to account for the run-time of more than r processes in parallel, under the constraint of only r available processors. Based on the scheduling scheme of 3. above the time to run the 1st batch of r+1 processes would be;

$$\text{Run-time} = r.T^{r}_{r+1,1} + 1.T^{r+1}_{r+1,1}$$

Therefore the total run-time of all N_{r+1} batches would be

$$\text{total run-time} = r.(T^{r}_{r+1,1} + T^{r}_{r+1,2} + \dots + T^{r}_{r+1,Nr+1}) + 1.(T^{r+1}_{r+1,1} + T^{r+1}_{r+1,2} + \dots + T^{r+1}_{r+1,Nr+1})$$

□ Eqn 2.

Likewise, for all N_{r+2} batches of r+2 processes in parallel;

$$
\begin{aligned}
\text{total run-time} = \ & r.(T^{r}_{r+2,1} + T^{r}_{r+2,2} + \dots + T^{r}_{r+2,Nr+2}) \\
& + 2.(T^{r+1}_{r+2,1} + T^{r+1}_{r+2,2} + \dots + T^{r+1}_{r+2,Nr+2})
\end{aligned}
$$

□ Eqn 3.

This formulation is extended to all sized batches, and in the general case where n is greater than some integer multiple times r the formulation is more complicated and is as follows for the N_n parallel processes batches;

$$\text{run-time of 1st batch} = r.(\ T^{r}_{n,1} + T^{2r}_{n,1} + \dots + T^{r.int(n/r)}_{n,1}\ ) + (n-r.int(n/r))T^{n}_{n,1}$$

The latter term represents the remainder processes after all groupings of r processors have been executed and a group of less than r processes remain.

Where int(a/b) is defined as being equal to the largest integer, z, such that: $z*b \leq a$.

The run-time for all N_n batches of n processes in parallel is therefore written as;

$$\begin{aligned}\text{total run-time} = {} & r(T^{r}_{n,1} + T^{2r}_{n,1} + \ldots + T^{r.int(n/r)}_{n,1}) + (n-r.int(n/r))T^{n}_{n,1} \\ & + r(T^{r}_{n,2} + T^{2r}_{n,2} + \ldots + T^{r.int(n/r)}_{n,2}) + (n-r.int(n/r))T^{n}_{n,1} \\ & + \ldots + r(T^{r}_{n,Nn} + T^{2r}_{n,Nn} + \ldots + T^{r.int(n/r)}_{n,Nn}) + (n-r.int(n/r))T^{n}_{n,Nn}\end{aligned}$$

□ Eqn 4.

Combining equations 2,3, and 4 and substituting into equation 1a, the total run-time, T for the complete system of processes, with maximum parallelism n, on a multiprocessor which has a processor boundary of r processing units is derived as;

$$\begin{aligned}T = \{ & P_{ave}{*}T_{l(\text{unbounded})} - (r+1)\{ T^{r+1}_{r+1,1} + T^{r+1}_{r+1,2} + \ldots + T^{r+1}_{r+1,Nr+1} \} \\ & - (r+2)\{ T^{r+2}_{r+2,1} + T^{r+2}_{r+2,2} + \ldots + T^{r+2}_{r+2,Nr+2} \} \\ & - \ldots - n\{ T^{n}_{n,1} + \ldots + T^{n}_{n,Nn} \} \\ & + r.(T^{r}_{r+1,1} + T^{r}_{r+1,2} + \ldots + T^{r}_{r+1,Nr+1}) + 1.(T^{r+1}_{r+1,1} + T^{r+1}_{r+1,2} + \ldots + T^{r+1}_{r+1,Nr+1}) \\ & + r.(T^{r}_{r+2,1} + T^{r}_{r+2,2} + \ldots + T^{r}_{r+2,Nr+2}) + 2.(T^{r+1}_{r+2,1} + T^{r+1}_{r+2,2} + \ldots + T^{r+1}_{r+2,Nr+2}) \\ & + \ldots \\ & + r.(T^{r}_{n,1} + T^{2r}_{n,1} + \ldots + T^{r.int(n/r)}_{n,1}) + (n-r.int(n/r))T^{n}_{n,1} \\ & + r.(T^{r}_{n,2} + T^{2r}_{n,2} + \ldots + T^{r.int(n/r)}_{n,2}) + (n-r.int(n/r))T^{n}_{n,1} \\ & + \ldots \\ & + r.(T^{r}_{n,Nn} + T^{2r}_{n,Nn} + \ldots + T^{r.int(n/r)}_{n,Nn}) + (n-r.int(n/r))T^{n}_{n,Nn} \}\end{aligned}$$

□ Eqn 5.

And consequently the expression for the average bounded parallelism, given in equation 1a becomes;

$$\begin{aligned}P^{b}_{ave} = \{ & P_{ave}{*}T_{l(\text{unbounded})} - (r+1)\{ T^{r+1}_{r+1,1} + T^{r+1}_{r+1,2} + \ldots + T^{r+1}_{r+1,Nr+1} \} \\ & - (r+2)\{ T^{r+2}_{r+2,1} + T^{r+2}_{r+2,2} + \ldots + T^{r+2}_{r+2,Nr+2} \} - \ldots - n\{ T^{n}_{n,1} + \ldots + T^{n}_{n,Nn} \} \\ & + r.(T^{r}_{r+1,1} + T^{r}_{r+1,2} + \ldots + T^{r}_{r+1,Nr+1}) + 1.(T^{r+1}_{r+1,1} + T^{r+1}_{r+1,2} + \ldots + T^{r+1}_{r+1,Nr+1}) \\ & + r.(T^{r}_{r+2,1} + T^{r}_{r+2,2} + \ldots + T^{r}_{r+2,Nr+2}) + 2.(T^{r+1}_{r+2,1} + T^{r+1}_{r+2,2} + \ldots + T^{r+1}_{r+2,Nr+2}) \\ & + \ldots \\ & + r.(T^{r}_{n,1} + T^{2r}_{n,1} + \ldots + T^{r.int(n/r)}_{n,1}) + (n-r.int(n/r))T^{n}_{n,1} \\ & + r.(T^{r}_{n,2} + T^{2r}_{n,2} + \ldots + T^{r.int(n/r)}_{n,2}) + (n-r.int(n/r))T^{n}_{n,1} \\ & + \ldots \\ & + r.(T^{r}_{n,Nn} + T^{2r}_{n,Nn} + \ldots + T^{r.int(n/r)}_{n,Nn}) + (n-r.int(n/r))T^{n}_{n,Nn} \} \div T_{l(\text{bounded})}.\end{aligned}$$

□ Eqn 6.†

† Equation 6 and preceeding equations are not derived from any referrable source other than the authors own research efforts in formulating a mathematical framework for expressing parallelism. An in depth derivation of these equations can be obtained by contacting the same.

This equation quantifies the extent of parallelism present in a particular implementation. It is dependent on the amount of parallelism present in the system of parallel processes, but also incorporates the effects of limited availability of processing power, and this is reflected in the complexity of the equation.

Equation 6 can be written more eloquently as;

$$P^{b}_{ave} = \{P_{ave} * T_{l(unbounded)} - \sum_{i=r+1}^{n} i \, . \sum_{j=1}^{N_i} T^{i}_{i,j}$$

$$+ \sum_{i=r+1}^{2r-1} \sum_{j=1}^{N_i} (r.T^{i}_{i,j} + (i-r)T^{i+1}_{i,j})$$

$$+ \ldots + r.\sum_{i=1}^{N_n} \, . \sum_{j=1}^{int(n/r)} T^{jr}_{n,j} + (n-r.int(n/r)) \sum_{i=1}^{N_n} T^{n}_{n,i} \} \div T_{l(bounded)}$$

□ Eqn 7.

The parallelism of a software system is not vitally important when assessing the performance of that system on a multiprocessor. The vital statistic when assessing the speedup of a particular application is the run-time of the slowest path, or critical path. Increasing the number of processes that run in parallel, whether by increased availability of processing units or by further exploiting the parallelism of the system, will increase the overall parallelism of the system but will not increase the speedup unless this increased parallelism occurs along the critical path, thus reducing its run-time. Therefore, it is possible to add extra processing units to a multiprocessor to increase the parallelism of the system, at a price of greater processor inefficiency, but not increase the performance index. This is why optimising an application to a multiprocessor can be a difficult problem, since an optimal blend of parallelism, speedup and processor efficiency/cost is required. The following sections examine these concepts and their relationships more closely.

3.2 TRADE-OFF BETWEEN EFFICIENCY AND SPEEDUP

By implementing a software system on a multiprocessor a reduction in the execution time may be achieved. The original run-time achieved on a uni-processor expressed as a ratio of the improved run-time is defined as the speedup for that particular application. As this speedup is increased (to an upper limit), by added processors or otherwise, the efficiency of the processors is reduced. Naturally as more processors are dedicated to the same software system the total amount of processor idle time due to issues such as contention, communications and software structure, is increased. This trade-off between speedup and efficiency is to some extent determined by the average parallelism, unbounded or otherwise, of the software system[57]. In this section the relationship between efficiency and speedup is examined, and a bound is determined for the maximum simultaneous degradation of both speedup and efficiency.

With regard to uni-processor architectures indices such as programme execution time and instruction speed are easily determined. In parallel processing architectures the calculation of these factors is somewhat more complex and not as straightforward. The speed of a processor is still a concern but other issues include number of processors, communications overheads and the structure of the software system. In evaluating the overall performance of a parallel system two factors are examined: speedup and efficiency. Using mathematical notation speedup is defined as;

$$S(n) = \frac{T_1}{T_n}$$

□ Eqn 8.

Where T_1 is the time taken to run the application on one processor and likewise T_n is the run-time on n processors.

Efficiency is defined as the average utilization of the n allocated processors. Ignoring I/O overheads, the efficiency of a single processor system is unity. Speedup is naturally also unity. In general though, the relationship between efficiency and speedup is given by;

$$E(n) = \frac{S(n)}{n}$$

□ Eqn 9.

If in fact efficiency remains at unity whilst additional processors are added then linear speedup is being achieved (generally for speedup it is required that $s(n)=\alpha n$, $0<\alpha\leq 1$, but only the stricter case of $\alpha=1$ is assumed here). This is the ideal situation, because no loss in efficiency is incurred despite increasing speedup. In a practical system this is impossible to achieve, due to contention between processors for shared resources, inter-processor communication and several other performance degrading issues. Minsky and Papert made the claim that typical speedups are of the order $S(n)=\log(n)$, [58]. Other studies have shown that much larger 'typical' speedups can also be obtained[59]. Whilst researchers such as Heidelberg and Trivedi[60] and Fayolle, King and Mitrani[61] developed expressions for speedup and efficiency for particular software structures, this section attempts to look more abstractly at the tradeoff between them and at the fundamental issues affecting them. The results can then be extended to specific systems if one wishes. This is in the same spirit as the efforts of Chen[62].

Amdahl's well-known law [63] provides a good starting point. It states that if a fraction f of a computation is inherently sequential then the speedup is bounded by an upper limit, which is stated below;

$$S(n) \leq 1/(f+(1-f)/n)$$

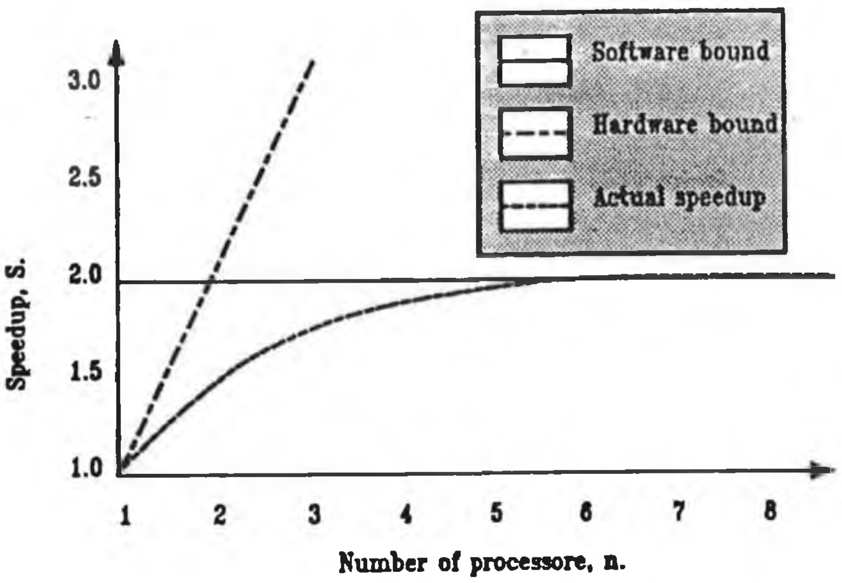
To determine upper and lower bound on speedup and hence on efficiency the concept of parallelism must be formally defined. Parallelism can be defined in four equivalent non-mathematical ways as,
(1) the average number of processors that are busy during the execution time of the software system in question, given an unbounded number of available processors,
(2) the speedup given an unbounded number of processors,
(3) the ratio of the total service required by the computation (the sum of the service demands of the subtasks) to the length of the longest path in the subtask graph (the length of a path is the sum of the service demands of its subtasks), or,
(4) the intersection point of the hardware bound and the software bound on speedup (to be defined shortly).

The equivalence of these four definitions is not entirely obvious. Recall that speedup with n processors, s(n), is defined as the ratio of the execution time when only one processor is available to the execution when n processors are available. Since the former is equal to the total service demand, and the ratio of the total service demand to the execution time gives the average number of busy processors, definition 2 is equivalent to definition 1.

If an unbounded number of processors is available, the execution time of a software system is simply the total service along some longest path. Hence, from the definition of speedup, definition 3 is equivalent to definition 2.

There are two simple upper bounds on speedup. The *hardware bound* reflects the limitation imposed by the hardware, and is given by the number n of available processors. This bound can be achieved only if all n processors can be kept busy all the time. The *software bound* reflects the limitation imposed by the software, and is derived by noting that, no matter how many processors are available to the system, the execution time must be at least as long as the longest path, as stated in section 3.1. Hence, the speedup is at most the ratio of the total service demand to the length of the longest path. The hardware and software bounds, and the actual speedup function, are depicted in figure 12.

The intersection point of the hardware and software bounds is significant; when additional processors are allocated, it is certain that there is not enough parallelism in the software in the system to keep all of the processors busy all of the time. This intersection point is the point where n (the hardware bound) is identical to the ratio of the total service demand to the length of the longest path (the software bound). Thus definition 4 is equivalent to definition 3 and hence all four definitions are equivalent. We note that the software and hardware bounds on speedup are analogous to the *Asymptotic Bound Analysis (ABA)* bounds on system throughput in a queueing network model of a computer system in which a number of identical, independent processes compete for service at a collection of system resources [64][65]. There is a single mapping between the two problem domains, with the number of independent processes corresponding in the ABA model to the number of processors in our model, and the bottleneck service demand in the ABA model corresponding to the length of the longest path in a directed graph representing the software system in our model. A similar analysis technique was used by Kumar and Gonslaves[66] for performance models of software containing critical sections. It is natural to ask how to determine the average parallelism of a particular software system. There are analytical approaches, such as graphing techniques, and experimental approaches (running the software with sufficient numbers of processors). The important issue is though, not how the measure is determined, but that once it has been determined, it provides a succinct characterisation of the inherent parallelism of the software system. As the remainder of this chapter shows, there is a considerable amount of information built into this measure.



Upper bounds & actual speedup for sample software system.

FIGURE 12

If the average parallelism is denoted as A, an expression for the lower bound on speedup can be determined (and thus for efficiency). Consider an arbitrary software structure with average parallelism A. Let T_∞ denote the system's run-time given unlimited availability of processing units. Based on the definition for average parallelism it follows that the total processor busy time is equal to AT_∞. The assumption of a work conserving scheduling scheme is assumed for the following derivations.

The execution time for the software system is given by $(AT_\infty + I(n))/n$ where I(n) is the processor idle time summed over all processors. Since the sequential execution time is nAT_∞ the speedup is by definition, $S(n)=nA/(A+I(n)/T_\infty)$.

To determine a lower limit on speedup it is necessary to determine the bound on I(n). To this end define $\omega(t)$ to be that portion of the software that has not yet completed at time t. $\omega(t)$ includes those processes that have not yet been initiated, and those that have been initiated but have not completed. The service demand of each process in $\omega(t)$ is the original service demand of that process minus any service already supplied (if any). Further, define L(t) to be the longest path within $\omega(t)$. As the execution of the software begins L(t) begins to diminish from an initial value of T_∞ to a final value of zero, on completion. If, at some time t during the execution L(t) is not decreasing then it is valid to observe that no process in the longest path is being serviced. Since no precedence constraints prevent the execution of such a task, and since the scheduling scheme is work conserving, it must be the case that that there are no idle processors at time t. Thus, processors can only be idle during those times when L(t) is decreasing. Since L(t) decreases in linear fashion for a total length of time T_∞, and since at most n-1 processors can be idle at any one point in time, the total idle time I(n) is at most $T_\infty(n-1)$, which establishes the result:

$$I(n) \leq T_\infty(n-1)$$

$$\Rightarrow S(n) \geq nA/(n+A-1) \qquad \square \text{ Eqn 10.}$$

$$\Rightarrow E(n) \geq A/(n+A-1) \qquad \square \text{ Eqn 11.}$$

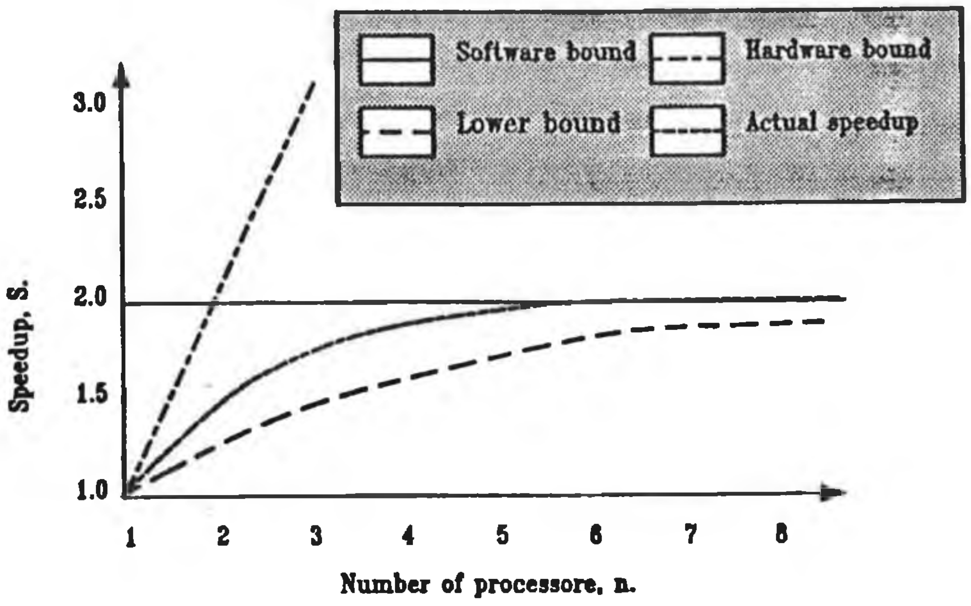
It can be observed that if, $n \gg A$ then $S(n) \to A$

and if, $n \ll A$ then $S(n) \to n$

Figure 13 is similar to figure 12 with the added exception that the lower bound on speedup is included. The significance of this is that no matter how poorly a work conserving hardware system is designed or how archaic the software may be the speedup can never fall below the given lower bound. The above derivation was concerned with obtaining a bound which represents the worst possible case over the complete space of work conserving scheduling disciplines, and may be a unreasonably poor indicator to the performance of some more rational scheduling disciplines. In a processor sharing scheduling regime tighter bounds can be obtained. The equations below demonstrate this;

$$S(n) \geq \min\left(A, \frac{nA}{n+A-1-\frac{(n-1)(A-1)}{P_{max}}}\right)$$ □ Eqn 12.

$$E(n) \geq \min\left(A, \frac{A}{n+A-1-\frac{(n-1)(A-1)}{P_{max}}}\right)$$ □ Eqn 13.



Upper bounds & actual speedup for sample software system.

FIGURE 13

How 'bad' can speedup and efficiency simultaneously become? As extra processors are added to service a software system increases in speedup are achieved at the expense of processor efficiency. Is it possible to achieve a high speedup by merit of a low efficiency? This is answered by the following deduction. From equations 12 and 13, it is observed;

$S(n)/A \geq n/(n+A-1)$

$E(n) \geq A/(n+A-1)$

$\Rightarrow S(n)/A + E(n) \geq (n+A)/(n+A-1) \geq 1$

This can be phrased as follows: *for any work conserving scheduling scheme, any* *software structure, and any number of processors, the sum of the attained efficiency and the attained fraction of the maximum speedup must always exceed unity.* Thus, for example, an average processor utilization (efficiency) of 20 per cent implies an attained speedup of more than 80 per cent of the maximum possible. These equations serve to quantify the relationship between the speedup and efficiency of a software system.

3.3 THE PARTITIONING PROBLEM AND ALLOCATION OF PROCESSES

Programming languages for multiprocessors may have implicit or explicit partitions and schedules[57], i.e. the partitioning of the software into subtasks may or may not be user-specified, and the mapping of these subtasks also may or may not be defined by the user. Partitioning and scheduling are multiprocessor-dependent issues. Partitioning is neccessary to ensure that the granularity of the parallel programme is coarse enough for the target multiprocessor, although an attempt is also made to preserve parallelism. Scheduling is necessary to achieve a good processor utilisation and to optimise inter-process communication in the target multiprocessor.

The scheduling problem is to assign subtasks in the partitioned programme to processors, so as to minimise the parallel execution time. The parallel execution time depends on processor utilisation and on the overhead of inter-processor communication. The problem of multiprocessor scheduling without communication overheads has been studied in great depth, Graham [67] for example. The general problem with arbitrary execution times and precedence constraints is NP-complete in the strong sense, along with many apparently simple cases, e.g. 2-processor scheduling with execution times equal to 1 or 2, but arbitrary precedence constraints. Despite this seemingly pessimistic result, a simple, linear-time *list-scheduling* algorithm has a constant performance bound of 2[68], i.e. the schedule generated by the list scheduling algorithm will have a parallel execution time which is at most twice the optimal parallel execution time. Thus, the multiprocessor scheduling problem doesn't pose an obstruction to achieving linear speedup.

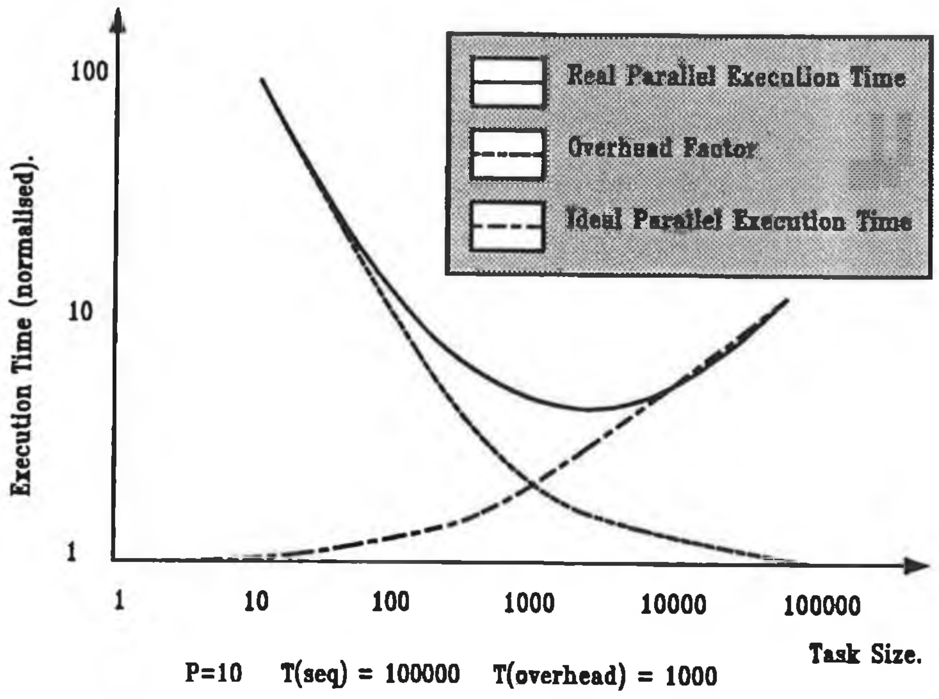
The partitioning of a software system specifies the sequential units or subtasks of computation in the system. There are three significant properties relating to subtasks;
(1) The subtask's sequential execution time, or subtask size.
(2) The subtask's total overhead, which includes scheduling overhead and communication for the subtask's inputs and outputs.
(3)The subtask's precedence constraints, which specify the parallelism in the partitioned programme.

Naturally enough, the execution time of a software system depends on its partition and schedule. Figure 14 illustrates the general way in which the parallel execution time depends on the partition. The abscissa gives the average subtask size of the partitioned programme. The ordinate gives the parallel execution time, normalised with respect to the parallel execution time necessary for ideal speedup. This normalised parallel execution time is also equal to the ratio;

(number of processors)/(Actual speedup).

The curves in figure 14 were plotted on the assumption of 10 processors being available, a sample sequential execution time of 10^5 cycles and a sample overhead of 10^3 cycles per subtask. The curves are indicative of a typical application. The curve representing the ideal parallel execution time is calculated on the assumption of zero overheads. As the subtask size rises from 10 to 10^5 cycles the normalised execution time rises from 1 to 10 due to loss of parallelism. The curve representing overhead factor is calculated by evaluating:

(Subtask size + Subtask overhead)/(Subtask size)



The Parallelism-Overhead Trade-off.

FIGURE 14

Finally, the curve denoting real parallel execution time is obtained by multiplying both the ideal parallel execution time by the overhead factor. In the graph it appears as the sum of the other two plots because of the log scaling of the axes. Some important points arise from observing the graph. Firstly, the presence of overheads, be they communications or otherwise, is to be expected in any pratical system and their presence makes ideal speedup impossible. Secondly, and more importantly, the real parallel execution time is minimised at an optimal intermediate granularity. The partitioning problem can be defined appropriately as the problem of finding the corresponding optimal intermediate partition.

The graph illustrates the trade off between parallelism and overhead. However, the continuous variation in task size is a very simplistic view of partitioned software systems. Real software systems are discrete structures. It may not be possible (and generally is not possible) to partition real parallel programmes into subtasks of equal run-times. Further, the overhead incurred by a subtask depends on the partition itself. Thus finding the optimal partition of a real software system is a much harder problem than finding the minimum of the appropriate curve in the graph of figure 14. In fact the problem is similar to the scheduling problem in that it is NP-complete in the strong sense. It is not within the scope of the thesis to investigate further the algorithms which attempt to solve either this problem or its intrinsically related problem of scheduling for more practical software systems.

CHAPTER 4

Parallel Implementation of an Explicit Self Tuning Regulator

4.1 INTRODUCTION

Like many other types of controllers the Self Tuning Regulator (STR) requires many operations to be performed within each sample period. As more complex control theory concepts are developed controllers become more taxing on the hardware used to implement them. Improved performance margins in microprocessors go some way in alleviating the problem but a more feasible solution seems to be the combination of processors in parallel/pipeline configurations to give a more powerful multiprocessing entity. An algorithm which contains operations which can be calculated independent of one another is suitable for transposing onto a parallel architecture. An algorithm which is not inherently parallel is nevertheless suitable for pipelining provided it can be fragmented into appropriate segments. The STR falls into the latter category.

At a very coarse level an explicit STR can be broken down into three separate processes:- process identification, controller parameter evaluation, and control signal calculation. If each of these processes were to form a pipeline stage (as described in section 4.2) then the result would be a 3 stage pipeline. Unfortunately the three processes are of unequal run times, thus if a processor were devoted to each stage then two processors would spend considerable time in an idle state awaiting the longer process to finish. Therefore, consideration must be given to 'time balancing' each stage and in choosing the optimum number of stages.

Furthermore, increasing the number of stages will increase the maximum speed of operation of the algorithm. This can be achieved by breaking down each stage into smaller 'substages' or by simply dividing the workload of one stage among several smaller stages. In considering these aspects it is concluded that a balance must be sought between maximising the number of pipeline stages, equalising run-times and minimising throughput delay of pipeline.

4.2 THE PIPELINING TECHNIQUE

It is appropriate at this point to give a short introduction to pipelining, outstanding from the description given in chapter 2, before proceeding to describe how the same is applied to the explicit Self Tuning Regulator. The linear pipeline structure, which was one of the earliest parallel architectures, can be represented schematically as in Figure 7 (see chapter 2). As can be seen in the diagram, information enters the pipeline on the extreme left and passes through several processors (or 'processing stages' as they are called) and eventually the processed information leaves the pipeline on the extreme right. The pipeline was developed in an attempt to overcome hardware redundancy present in conventional sequential circuitry. In fact, on average only 10% of circuitry is active at any given time in an non-optimised digital computer. The idea behind the pipeline is to split up the function to be executed into a number of sub-functions. Each of these sub-functions constitutes a pipeline 'stage'. A stage is an independent processing unit and several of these in sequence constitute a pipeline. Each stage receives data from its predecessor, processes it and passes the result to its successor. Thus in this fashion all stages can be active simultaneously. Furthermore the simpler the operations at each stage the faster the rate at which data can be clocked through the pipeline. The time interval between two consecutive outputs from the pipeline is equal to the time of the slowest stage regardless of how fast the other stages may be. Thus, an attempt should be made when constructing a pipeline to evenly distribute the workload between the stages. Failing this, some stages will spend time in an idle state, awaiting slower stages to complete. This redundancy would defeat the purpose of the pipeline.

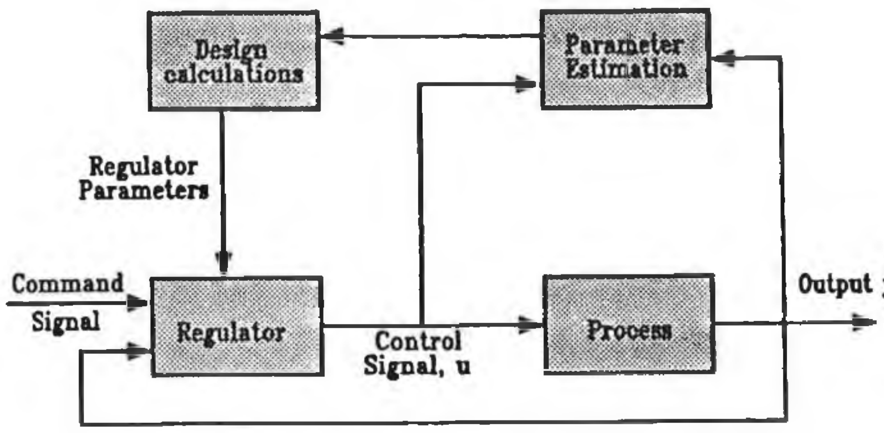
4.3 PIPELINING OF A SELF TUNING REGULATOR

The report considers the effects and both advantages and disadvantages of applying a pipelining technique to an explicit '*Self Tuning Regulator*' (STR). The pipeline is simulated on a 10 MHz intel 80286 microprocessor and any timing measurements presented are based on this microprocessor and should accordingly be interpreted as relative quantities. The objective of the simulation is demonstrate the possible speedups which may be attained through pipelining. If the same scheme was implemented on a faster pipeline, using more powerful microprocessors, the speedup results would be approximately be the same, as the number of FLOPs would be unchanged. The STR used throughout is a software controller coded in the C language, based on an algorithm presented in by Astrom and Wittenmark[68].

4.3.1 The Self Tuning Regulator

The STR consists basically of two loops. The most inner of the two loops consists of the process to be controlled and an ordinary linear-feedback regulator. The outer loop on the other hand contains a parameter estimator and a design calculator which determine the parameters of the regulator. The whole system is recursive and parameter estimates of the system are continuously updated. In this way the whole system can adapt to changes in process parameters and tune itself to accommodate these changes to avoid degradation of performance. There are several techniques available to implement the identification routine including Kalman filtering, least squares, extended or recursive least squares or stochastic approximation. With regard to the design technique the range is even broader and virtually any design method can be implemented. The STR, used in the following pipeline application, uses recursive least squares and pole placement for the identification routine and design methodology respectively.

Figure 15 shows the block structure of a self tuning regulator.



Block Diagram of a Self Tuning Regulator.

FIGURE 15

The structure of the software simulation of the STR can be simplified as follows. Note that a function is represented by the syntax *FUNCTION()*, which is based on the C function declaration and evocation protocol. The functions given are generalised to represented the main processes of the software model of the explict STR;

```
fn-STR()
 {
        RLS();
        DIOPH();
        AD-CTRL();
 }
```

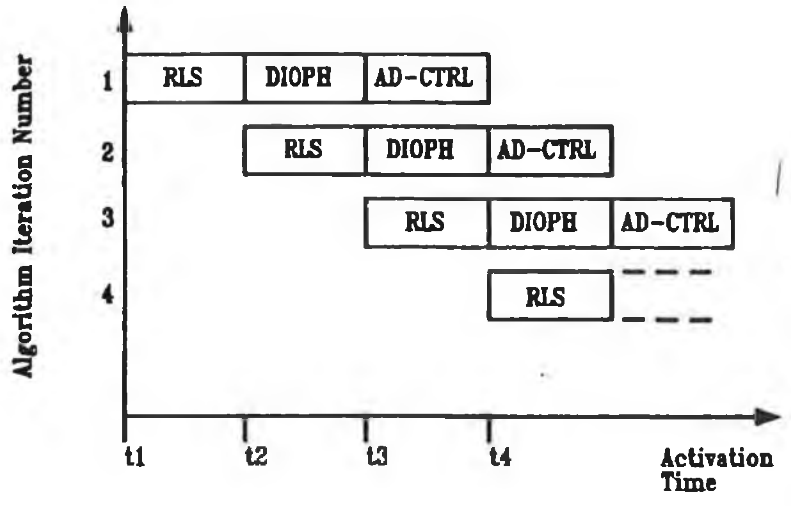
Basically there are three main functions: the recursive least squares routine (RLS), the solution of the diophantine equation (DIOPH) for solving the controller parameters and the calculation of the control signal (AD-CTRL). The STR recursive 'cycle' begins by identifying the process by performing a RLS on input and output data. The process parameters are then used to determine the controller parameters in order to achieve the desired performance. Whilst determining the controller parameters several equations arise among which is a diophantine equation[68]. This diophantine equation is solved in the call to the DIOPH function. Finally, once the process has been identified and the regulator parameters determined a control signal is calculated and applied to the process, and then the cycle begins once more.

To summarise, the parameters of the process to be controlled are firstly identified, then knowing these parameters the regulator parameters are determined. Finally using the regulator a control signal is generated and applied to the process. Intuitively it can be noticed how sequential the Self Tuning Regulator is, which makes it very unsuitable for strictly parallel implementations.

4.3.2 Mapping the Self Tuning Regulator onto a Pipeline Structure

The main principle of pipelining is to allow processes to execute concurrently even if the processes cannot exist strictly in parallel. This is illustrated graphically below in Figure 16 for the case of a 3-stage pipeline ;

As can be seen cycle one begins at time t1, cycle two at time t2 and cycle 3 at time t3. When cycle 4 commences (RLS iodentification stage), at time t4, it requires that the most recent input & output data be supplied so as to give the latest system identification. Examining the graph shows that the most recent cycle to have completed when cycle 4 commences is cycle 1. The output from cycle 1 serves as the latest information for the RLS identification routine of cycle 4. Compare this to the unpipelined case where there would be no overlap of the cycles and the last output would have been from the previous cycle, cycle 3. Thus the information used in the RLS routine comes from a cycle which began two three cycles previously. This can be conceptualised as a delay of 2 sampled periods in the information entering the RLS routine, even though the information being fed to the RLS routine is never actually delayed, but it is delayed relative to the unpipelined case when you consider that in the unpipelined instance the identification routine is fed the information directly from the previous cycle. Conclusively for an n-stage pipeline there is a relative delay of n-1 sample periods . The graph above is very simplistic and as will be shown later a pipeline with many more stages can be achieved by subdividing each of the three processes shown into smaller faster stages.



N.B. :	*RLS*	*: Recursive Least Squares routine. (Stage 1)*
	DIOPH	*: Solution to Diophantine eqn routine. (Stage 2)*
	AD-CTRL	*: Calculation of control signal routine. (Stage 3)*

These three functions executed in sequence constitute one 'cycle' of the Self Tuning Regulator. The graph shows where in time each cycle begins. When the RLS stage of the first cycle is completed (time = t2) information is passed to stage 2 and at the same instant the RLS stage of cycle 2 is initiated, and similarly when the RLS of cycle 2 completes (time = t3) the RLS stage of cycle 3 begins. In this fashion it can be seen how stage 1 is continuously active if all three stages are of equal execution time. Likewise the other two stages are also continuously active

FIGURE 16

Simulation of an 'n' stage pipeline can be achieved by introducing an n-1 sample period delay into the Recursive Least Squares (RLS) routine of an unpipelined STR. A collection of graphs at the end of the report show the resulting effect of multi-stage pipelining on controller performance and parameter convergence. The system remains stable for a 5-stage pipeline (6-stage and greater pipelines were not examined) and parameter convergence remains largely unaffected.

4.3.3 Timing of Algorithmic Subtasks

Initially difficulty was experienced in attempting to time the main functions within the programme. Repetitive measurements proved to be inconsistent even for apparently 'constant-calculation' functions. The main reasons for this seem to be periodic delays due to interrupts and i/o delays. These obstacles were overcome by connecting a high-speed digital oscilloscope to one of the PC's o/p ports (parallel port). Before entering the function to be timed a data line would be set high and on completion of this function this same data line would then be set low. Within TurboC this may be done as follows;

outportb(*address*, *data*);

Where address of parallel port = 0x378 and data was either 0x00 for a low and 0xff for a high. Also by using an address of 0x21 the internal interrupts could be disabled as required.

As the main programme looped, the oscilloscope recorded a periodic pulse or asymeterical square wave and measurement of this pulse was quite straightforward on the Hewlett Packard 54501A 100 MHz oscilloscope. The measurements were based on averages over a thousand cycles, to reduce the eventuality of periodic error. The four main functions, as described in section 3.1, yielded times given table 1.

As can be seen from the table 1 there is quite a large diversity in run times thus rendering the allocation of a stage per function as impractical. A more feasible solution is to break down the larger functions into more reasonable units.

		System Order		
		1	2	3
FUNCTION	Ad-Ctrl	0.328	0.640	1.050
	RLS	1.450	4.120	7.800
	Diophantine	0.185	0.362	0.580
	Gauss-El.	1.270	3.000	5.360
	Total	3.233	8.122	14.79
	Total Prog.	3.920	9.360	17.10

[mS]

Table showing the execution times for the three main functions in the STR

N.B. The DIOPH function is broken into two functions, Diophantine and Gauss-el. The Gauss-el (gaussian elimination) function is treated separately because of it's relatively long run time.

TABLE 1

The Recursive Least Squares (RLS) routine was broken down in the software simulation as follows;

```
fn_ls()
 {
      LOOP1
```

```
            LOOP2
             fn_ls1();
             fn_ls2();
            END LOOP2
        END LOOP1
  }
```

Where fn_ls2() is given as;

```
fn_ls2()
 {
      fn_ls3();
      fn_ls4();
      fn_ls5();
      fn_ls6();
 }
```

The six processes, fn_ls1 to fn_ls6, which represent segments of coding n the algorithm, are each designated at least one stage of the pipeline. Those that appear in loops are given the loop number of stages. If a function appears in a loop of order n then n identical stages which perform that function will appear in consecutive order in the pipeline. The 6 constituent processes have time performances shown in the table below;

System Order:	1	2	3
fn-ls1	0.500	0.688	0.876
fn-ls2	1.119	3.667	7.612
fn-ls3	0.120	0.291	0.473
fn-ls4	0.399	0.393	0.403
fn-ls5	0.222	0.609	0.831
fn-ls6	0.356	0.604	0.997

[mS]

Table of execution times for constituent functions of the Recursive Least Squares Routine

TABLE 2

The STR controller is configured to control and recursively identify a real system represented by either a first, second or third order model. The longest execution time for a function , in the RLS routine, when controlling a third order system is fn_ls6 at 0.997 mS (excluding fn_ls2 which is a dummy function containing 4 other functions). This is a dramatic improvement on the pipelining of the three main functions with previous stage times of up to 7.8 [mS].

The Ad__Ctrl function which takes 1.05 [mS] is also a candidate for decomposition so as to bring it into line with the longest stage of the RLS routine. The generalised C encoding of the Ad_Ctrl is as follows;

```
Ad_Ctrl()
 {
      switch( choice )
        case 1: fn-ad1()
                     fn–update1();
        case 2: switch( aonec )
                     {
                       case 0: fn–ad2();
                                    fn–update2();
                       case 1: fn-ad3();
                                    fn-update3();
                     }
        case 3: switch( aonec3 )
                     {
                       case 1: fn-ad4();
                                    fn-update4();
                       case 2: fn-ad5();
                                    fn-update5();
                       case 3: fn-ad6();
                                    fn-update6();
                       case 4: fn-ad7();
                                    fn-update7();
                     }
 }
```

System Order :	1	2	3
fn–ad (average)	0.183	0.360	0.809
fn–update (average)	0.154	0.277	0.247

Table showing the average execution times for the functions fn–ad & fn–update.

TABLE 3

Only two functions are ever called in a call to AD-CTRL. The average timing measurements for these sub-functions are given in Table 3.

The longest run time of 0.809 [mS] is less than that for the longest RLS decomposed function, thus the longest stage length/minimum sample time still remains at 0.997 [mS]. The diophantine routine is less than this value thus averting any need to break it down further. The Gaussian Elimination function, Gauss-El, however is far too long at 5.36 [mS] for third order process control thus necessitating some degree of decomposition. This function is broken down as follows;

```
ga-el()
 {
      LOOP1
       fn-ga1();
       fn-ga2();
           LOOP2
            fn-ga3();
           end LOOP2
      end LOOP1
 fn-ga4():
 }
```

The tabulated results are given below and again no function exceeds the maximum stage length in the RLS routine.

System Order	1	2	3
fn-ga1	0.156	0.260	0.432
fn-ga2	0.312	0.408	0.500
fn-ga3	0.488	0.676	0.760
fn-ga4	0.260	0.520	0.764

Table showing the execution times the constituent functions of the Gaussian elimination routine.

TABLE 4

4.3.4 Speedup Results and Performance Analysis

The maximum stage lengths for each process order controller are summarised in table 5 and the respective speed-ups over the unpipelined version are also given.

For control of a third order system the speedup is more significant than for lower orders. This is due to the fact that as the order is decreased the constant overheads become more significant as the order-dependent calculations become simpler (i.e smaller loops, smaller arrays, simpler matrices ... etc.). Furthermore as the order is increased so too is the number of stages. Most stages are contained within order dependent loops and some are contained in nested order dependent loops whereby an increase to (n+1)th order from nth order would result in n+1 extra stages per loop. This in turn would increase the throughput delay of the pipeline.

fn-main	System Order	Longest Pipeline Stage	Speedup Factor
17.8 mS	3	fn-ls6 @ 0.997 mS	17.85
9.68 mS	2	fn-ls1 @ 0.688 mS	14.07
3.90 mS	1	fn-ls1 @ 0.500 mS	7.800

Table showing the maximum speedups possible for 1st, 2nd and 3rd order systems, by the pipelining technique.

N.B. Speedup = (fn-main)/(Longest Stage)

This is based on the assumption that the time between two consecutive control signals from the pipelined STR is equal to the slowest stage of the pipeline. Thus speedup is the time between control signals in the unpipelined STR (Full STR Cycle) divided by the longest pipeline stage.

TABLE 5.

The graph section also contains 'computation time versus system order' graphs for the Least squares routine and the Gaussian elimination routine. As can be seen the time of execution increases exponentially with order. To determine the relationship between order and time a line can be fitted to the graph of ln(Time) Vs. Order. This technique yields the following results;

<u>For RLS routine :</u>

Execution Time = Exp(0.841xOrder - 0.401)

<u>For Gauss Elimination :</u>

Execution Time = Exp(0.719xOrder + 0.434)

4.4 SUMMARY

The last section presented encouraging figures for the increase in the speed of the pipelined controller execution time. However, these figures represent absolute and theoretically achievable speedups. In an actual hardware implementation there are several aspects to be considered. There would be overheads associated with the passing of information from processor to processor and the delay in sending such information. Additionally, because of the high degree of decomposition of the main functions the pipeline would have many stages and the actual number of stages would vary depending on the order of the system to be controlled. This would make the devotion of one processor to each stage quite unfeasible. Thus, there would have to be time-sharing on most processors slowing down the pipeline even further. Nevertheless, considering the system remains stable any speedup, provided it is achievable at an acceptable cost is to be welcomed.

By examining the sub-functions of the main processes it was possible to devise a five-stage pipeline with a speedup of 4.24, due to the execution time of the slowest stage being 4.03 milliseconds. This compares with a maximum figure of 5.0 for the theoretical speedup of a five-stage pipeline with each stage receiving equally balanced work-loads. The most important initial concern when implementing a pipeline is to ensure that the controller remains stable despite the imposition of parallelism upon the calculation sequence. In appendix C, graphs 4.1 to 4.6 show how each of the parameters, estimated by the recursive least squares to model the 3rd order process being controlled, converges. All paramters converge quite rapidly and all have a maximum error which occurs in the first few cycles of the STR pipeline. It is necessary that the parameters behave in this manner, to avoid sporadic errors after the pipeline completes its first few initial starup cycles. Graph 4.7 shows the response of the STR to a step input. The control signal is also shown. The control signal doesn't behave in an erratic or erroneous manner and the system output converges quickly and accurately.

From a mathematical viewpoint there are a few points to consider with regard to pipelines in general. The time of the slowest function within the programme will naturally constitute the longest stage, but it will also be equal to the time delay of the pipeline (i.e. the time between two consecutive pipeline outputs). Therefore the theoretical overall speedup will be expressed as a ratio of total unpipelined execution time to run-time of the slowest stage of the pipeline. Furthermore the recursive least squares routine will always begin at the start of a sample period and not when the previous pipeline stage completes. The delay introduced by the pipeline will always be a unit number of sample periods therefore. Finally, if the algorithm decomposes into say, ten subtasks, for example it is not sufficient (although it is a requirement) to say that all ten subtasks can execute in sequence in less than ten sample periods. It must also be shown that each subtask individually takes less than one sample period to execute. Consequentially the minimum sample period that can be chosen in the control process is equal to the time of the slowest stage.

In conclusion it can be said that a reduction in sample period for the system makes it possible for the real time execution of algorithms which would otherwise be confined to the realms of control textbooks.

CHAPTER 5

Development of a Parallel Simulator for the PUMA 560 Robot Arm

5.1 THE PUMA 560 ROBOT MODEL

This chapter initially develops a 3rd order dynamic model for the first three joints of a PUMA 560 robot. The PUMA 560 manipulator arm, which is an elbow type industrial robot, is illustrated in Figure 17. The manipulator has six degrees of freedom. The positions of the first three links determine the end-effector position, while the last three links specify tool orientation. For convenience the first three joints are referred to as the waist, shoulder and elbow joints respectively and the final three joints collectively form the wrist and gripper. In order to specify tool position and orientation, knowledge of the link sizes and joint angles is required. In the geometric sense, a link may be considered to be a rigid structure, supporting one or two joint axes. When specifying the link dimensions it is therefore necessary to give the relationship between the joint axes. There are two groups of manipulator joint: revolute joints and prismatic joints. The PUMA 560 joints are of the former type. Thus, any joint can be fully described by two dimensions: Link length and Link twist (or angle)[70]. Further, any link with the exception of the end links, will have two common normals associated with it. One for the lower joint and one for the higher link. The distance along the axis between the two normals is called the distance between the links, d_n. The angle between the links is measured in a plane normal to the axis. To summarise, the four parameters associated with a link are:

(i). Link length, shortest distance between axes,

(ii). Link twist, angle between axis measured in plane perpendicular to a common normal,

(iii). Distance between two links, distance measured along axis,

and,

(iv). Angle between links, angle measured between common normal on plane perpendicular to axis.

Specification of the four values above for each of the six manipulator links allows one to determine the link and end-effector positions. This, however, would prove to be a tedious task. To simplify the procedure it is convenient to assign a coordinate frame to each link, and then to form a set of homogeneous transformations which describe the relative position and orientation of each link to that of the previous one [70,71].

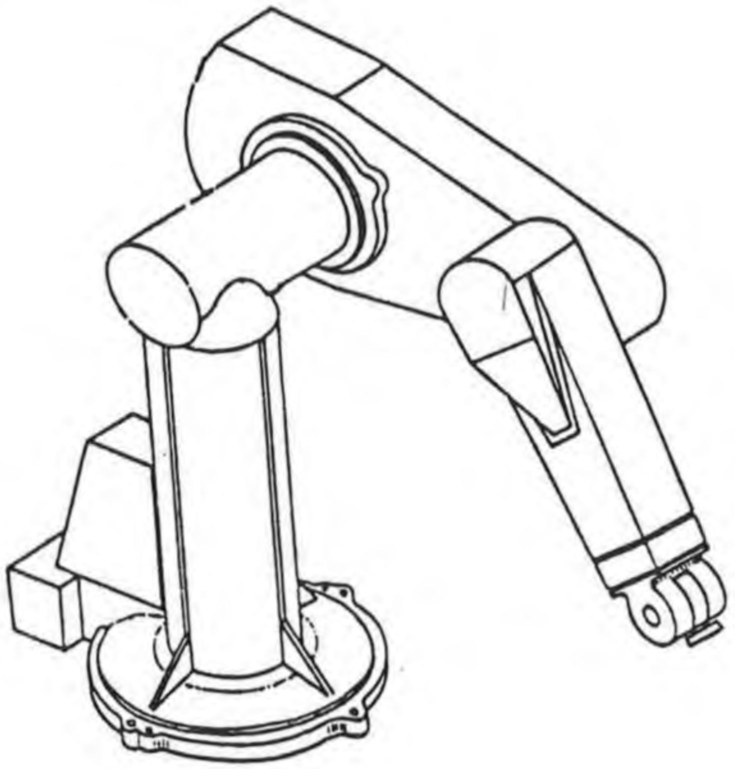


FIGURE 17

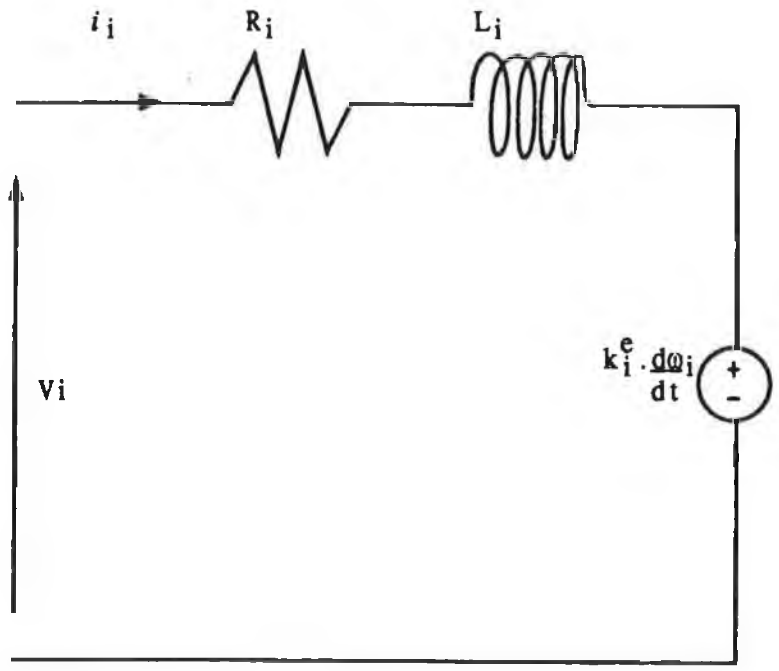
The initial task is that of chosing a coordinate frame for each joint and specifying origins and axes. The scheme adopted is that presented by Denivit & Hartenburg[72]. The origin of each coordinate frame for a link n is chosen as the intersection of the common normal between the axes of joints n and n+1, and the axis of the (n+1)th joint. The axis selection technique suggested by Paul[70] is also adopted. It designates the z axis of the cartesian frame of joint n to be the axis of rotation of joint n. The x axis is specified as the normal directed from joint n towards joint n+1. Finally, the y axis completes the conventional orthogonal set of right handed axes.

As stated, knowledge of the angular position of the first three links gives end-effector position, while tool orientation is determined by the wrist joints. The positioning and path tracking problem is dominated by the dynamics of these three links. Normally, when tracking a given path, the gripper and payload are treated as a lumped mass on link three. This does not introduce significant errors since the last three links are dimensionally small and the payload travels with the third link. It is sufficient to model the manipulator in space by modeling the dynamics of the initial three links of the arm[73].

Once each link is given a specified coordinate frame the task reduces to developing an interrelationship scheme between joints. This makes it possible to ascertain the position of each manipulator link at any time. One method of achieving this is to create homogeneous transformations which are conveniently performed using matrix algebra. In this context, the transformation matrices are known as T matrices ($\in \mathbb{R}^{4x4}$). These are chosen to transform the origin of a reference coordinate frame to the origin of the present frame under consideration, as suggested by Denivit & Hartenburg[72]. Using this method, the effect on the end joint of movement of the first joint can be calculated by multiplying the intermediate joint to joint transformation matrices.

By inspection the model can be decomposed into subtasks, some of which can be executed independently or in parallel. The manner of decomposition determines whether these *concurrent* subtasks are classified as *fine grain parallellism* or *coarse/medium grain parallelism*[69]. The scheme chosen involves exploiting both medium grain and coarse grain parallelism, in an attempt to develop a parallel simulator which will achieve significant speedup over the standard sequential model. Before specifying parallelism in the PUMA 560 manipulator dynamics the 3rd order model must first be derived.

5.1.1 Third Order Dynamical Equations



ω_i = motor position
L_i = armature inductance
k_i^t = voltage constant
V_i = armature voltage
R_i = armature resistance
i_i = armature current
k_i^e = torque constant

FIGURE 18

The third order model for the PUMA 560 manipulator is an extension of the second order model to incorporate the actuator dynamics and friction forces. Quite often the friction forces in manipulators are significant, and in larger manipulators they can account for up to 25% of the required torque for motion[71]. Friction in the PUMA 560 is modeled as Coulomb friction. Actuation of the joints is achieved by means of permanent magnet d.c. servomotors. There are two types of motor employed in the robot, 100 Watt motors for the first three links and 50 Watt motors for each of the three wrist joints. Only the larger motors need to be analysed. Figure 18 shows a simple equivalent circuit of such a d.c. servomotor.

The equations modeling this circuit are attained relatively easily. Equation 14 expresses the input voltage, Vi, as a function of the armature current, armature resistance, armature inductance and motor angular position.

$$V_i = R_i . i_i + L_i . \frac{di_i}{dt} + k_i^e . \frac{d\omega_i}{dt}$$

□ Eqn 14.

Further, the torque produced by a dc motor is proportional to the armature current of the dc motor:

$$F_i = k_i^t . i_i$$

□ Eqn 15.

where F_i is the torque experienced at joint i.

The joint position be can related to the motor position by the following equation:

$$\omega_i = N_i . q_i$$

□ Eqn 16.

where N_i is the gearing ratio of joint i.

By substitution of equations 15 and 16 into equation 14 the following relationship is derived:

$$V_i = k_i^e . N_i . \frac{dq_i}{dt} + (R_i F_i + L_i . \frac{dF_i}{dt})/k_i^t$$

□ Eqn 17.

Equation 17 expresses a relationship between the input voltage to the robot joint motors and the resulting torque and torque derivative. This equation is not sufficient on its own to model the robot manipulator. Every sample period it is required to evaluate the torque and the torque derivative, so an equation representing the manipulator should incorporate these additional calculations. There are several schemes to do this, the most common being the Euler-Lagrangian (E-L) technique. For a open-loop kinematic chain type manipulator with n degrees of freedom the Euler-Lagrangian is formulated as follows:

$$F_i = \sum_{j=1}^{n} D_{ij}\ddot{q}_i + I_{ai}\ddot{q}_i + \sum_{j=1}^{n}\sum_{k=1}^{n} C_{ijk}\dot{q}_j\dot{q}_k + G_i + H_i\dot{q}_i$$

□ Eqn 18.

where,

q_i = position of joint i,

F_i = torque acting on joint i,

I_{ai} = actuator inertia of joint i,

D_{ii} = effective coupling of joint i,

D_{ij} = coupling inertia on i joint due to joint j,

C_{ijj} = centripetal force on i due to joint j,

C_{ijk} = coriolis force on joint i due to joints j and k,

G_i = gravity loading of joint i,

H_i = coefficient of friction for joint i.

Bejczy, [75], has defined the inertial, centripetal and gravity terms as follows, using the trigonometric abbreviations given:

Let,

$S_i = \mathrm{Sin}(q_i)$,

$C_i = \mathrm{Cos}(q_i)$,

$S_{ij} = \mathrm{Sin}(q_i + q_j)$,

$C_{ij} = \mathrm{Cos}(q_i + q_j)$.

$$\begin{aligned} D_{11} &= m_1 k^2{}_{1yy} \\ &+ m_2(k^2{}_{2xx}S^2{}_2 + k^2{}_{2yy}C^2{}_2 + a_2{}^2C^2{}_2 + 2a_2x_2C^2{}_2) + \\ &\quad m_3[(k^2{}_{3xx}S^2{}_{23} + k^2{}_{3zz}C^2{}_{23} + d^2{}_3 + a_2{}^2C^2{}_2 + a^2{}_3C^2{}_{23} \\ &+ 2a_2a_3C_2C_{23} + 2x_3(a_2C_2C_{23} + a_3C^2{}_{23}) + \\ &\quad 2y_3d_3 + 2z_3(a_3C_{23}S_{23} + a_2C_2S_{23})] \end{aligned}$$

□ Eqn 19.

$$D_{12} = m_2a_2z_2S_2 +$$

$$m_3[(d_3x_3 + a_3y_3 + a_3d_3)S_{23} + (a_2y_3 + a_2d_3)S_2 - d_3z_3C_{23}]$$

□ Eqn 20.

$$D_{13} = m_3[x_3d_3 + a_3y_3 + a_3d_3)S_{23} - z_3d_3C_{23}]$$

□ Eqn 21.

$$D_{22} = m_2(k^2{}_{2zz} + a^2{}_2 + 2a_2x_2) + m_3[(2a_2a_3 + 2a_2x_3)C_3 + 2a_2z_2S_3 + k^2{}_{3yy} + a^2{}_2 + a^3{}_3 + 2a_3x_3]$$

□ Eqn 22.

$$D_{23} = m_3[(a_2x_3 + a_2a_3)C_3 + a_2z_3S_3 + 2a_3x_3 + a^2{}_3 + k^2{}_{3yy}]$$

□ Eqn 23.

$$D_{33} = m_3(k^2{}_{3yy} + a^2{}_3 + 2a_3x_3)$$

□ Eqn 24.

$$C_{112} = m_2(k^2{}_{2xx} - k^2{}_{2yy} - a^2{}_2 - 2a_2x_2)C_2S_2 + m_3[k^2{}_{3xx}(C_2S_2 + C_3S_3 - 2S_2S_3S_{23}) + k^2{}_{3zz}(2S_2S_3S_{23} - C_2S_2 - C_3S_3) + x_3(-2a_2C_2S_{23} + 4a_3S_2S_3S_{23} + a_2S_3 - 2a_3C_2S_2 - 2a_3C_3S_3) + z_3(a_2C_2C_{23} - a_2S_2S_{23} + 2a_3C^2{}_{23} - a_3) + a_2a_3S_3 - 2a_2a_3C_2S_{23} - a^2{}_2C_2S_2 + 2a^2{}_3S_2S_3S_{23} - a^2{}_3(C_2S_2 + C_3S_3)]$$

□ Eqn 25.

$$C_{113} = m_3[k^2{}_{3xx}(C_2S_2 + C_3S_3 - 2S_2S_3S_{23}) + k^2{}_{3zz}(2S_2S_3S_{23} - C_2S_2 - S_3C_3) + x_3(4a_3S_2S_3S_{23} - 2a_3C_2S_3 - 2a_3C_3S_3 - a_2C_2S_{23}) + z_3(2a_3C^2{}_{23} + a_2C_2C_{23} - a_3) + 2a^2{}_3S_2S_3S_{23} - a_2a_3C_2S_{23} - a^2{}_3C_2S_2 - a^2{}_3C_3S_3]$$

□ Eqn 26.

$$C_{122} = m_2a_2z_2C_2 + m_3[d_3z_3S_{23} + (d_3x_3 + a_3y_3 + a_3d_3)C_{23}]$$

□ Eqn 27.

$C_{123} = m_3[d_3z_3S_{23} + (d_3x_3 + a_3y_3 + a_3d_3)C_{23}]$

□ Eqn 28.

$C_{133} = m_3[d_3z_3S_{23} + (d_3x_3 + a_3y_3 + a_3d_3)C_{23}]$

□ Eqn 29.

$C_{213} = 0$ (because of general PUMA geometry)

□ Eqn 30.

$C_{223} = m_3[(-a_2x_3 - a_2a_3)S_3 + a_2z_3C_3]$

□ Eqn 31.

$C_{233} = m_3[(-a_2x_3 - a_2a_3)S_3 + a_2z_3C_3]$

□ Eqn 32.

$G_1 = 0$ (because of general PUMA 560 geometry)

□ Eqn 33.

$$\begin{aligned} G_2 = \; & m_2g(x_2 + a_2)C_2 - \\ & m_3g(x_3C_{23} + z_3S_{23} + a_3C_{23} + a_2C_2) \end{aligned}$$

□ Eqn 34.

$G_3 = -m_3g(x_3C_{23} + z_3S_{23} + a_3C_{23})$

□ Eqn 35.

By Newton's second law of physics, the following equalities hold:

$$\begin{aligned} D_{ij} &= D_{ji} \\ C_{ijk} &= C_{ikj} \\ C_{ijk} &= -C_{kji} \text{ for } i,k \geq j \\ C_{iji} &= 0 \text{ for } i \geq j \end{aligned}$$

□ Eqn 36.

The equalities of equation 36 give rise to the following relationships:

$$\begin{aligned} & D_{21} = D_{12},\ D_{13} = D_{31},\ D_{32} = D_{23}, \\ & C_{111} = C_{222} = C_{333} = 0,\ C_{121} = C_{112}, \\ & C_{131} = C_{113},\ C_{132} = C_{123},\ C_{221} = C_{212}, \\ & C_{231} = C_{213},\ C_{232} = C_{223},\ C_{321} = C_{312}, \\ & C_{331} = C_{313},\ C_{332} = C_{323},\ C_{211} = -C_{112}, \end{aligned}$$

$$C_{311} = -C_{113},\ C_{312} = -C_{213},\ C_{322} = -C_{223},$$
$$C_{313} = C_{323} = C_{212} = 0$$

□ Eqn 37.

The quantities x_i, y_i and z_i are the Cartesian coordinates of the centre of mass of joint i referenced to the base of the robot. The quantity m_i is the mass of joint i and k^2_{ixx}, k^2_{iyy} and k^2_{izz} are the radii of gyration for joint i. The quantities d_i and a_i are the link twists and the link lengths. The values of these geometric and inertial parameters which relate to the three primary joints of the PUMA 560 are listed in Table 6 and Table 7. These are the estimates obtained by Bejczy [75]. He arrived at these values by first taking detailed measurements of all link internal components, then calculating their individual moments of inertia, and later getting the cumulative effect using the Parallel Axis Theorem, Goldstein [76].

PUMA 560 Inertial Parameters

Link	Centre of Mass			Mass	Radius of Gyration (cm)		
i	X_i	Y_i	Z_i	$g.s^2/cm$	K^2_{ixx}	K^2_{iyy}	K^2_{izz}
1	00.00	30.88	03.89	13.21	1816.30	0151.93	1811.1
2	-32.89	0.00	20.38	22.80	0595.70	1355.60	1513.60
3	-2.04	-1.37	00.30	05.11	0151.48	0155.23	0020.70

TABLE 6

PUMA 560 Geometric Parameters

a_2(cm)	a_3(cm)	d_2(cm)	d_3(cm)
43.18	01.91	15.05	43.31

TABLE 7

It also possible to evaluate the first derivative with respect to time of the expression for torque, in the normal way:

The quantity $\dot{F}_i$ is the first derivative of the joint torque and is given by the equation overleaf:

$$\dot{F}_i = \sum_{j=1}^{3} (D_{ij}\dddot{q}_j + \dot{D}_{ij}\ddot{q}_j) + I_{ai}\dddot{q}_i$$

$$+ \sum_{j=1}^{3} \sum_{k=1}^{3} (C_{ijk}\ddot{q}_j \dot{q}_k + C_{ijk}\dot{q}_j \ddot{q}_k + \dot{C}_{ijk}\dot{q}_j \dot{q}_k)$$

$$+ \dot{G}_i + H_i\ddot{q}_i$$

□ Eqn 38.

By substituting the expression for torque and torque derivative into equation 18 the complete model can then be written as:

$$V_i = k_i^e.N_i.\dot{q}_i + R_i.[H_i\dot{q}_i + G_i$$

$$+ \sum_{j=1}^{3} D_{ij}\ddot{q}_i + I_{ai}\ddot{q}_i + \sum_{j=1}^{3} \sum_{k=1}^{3} C_{ijk}\dot{q}_j\dot{q}_k]/k_i^t$$

$$+ L_i.[\dot{G}_i + \sum_{j=1}^{3} (D_{ij}\dddot{q}_j + \dot{D}_{ij}\ddot{q}_j) + I_{ai}\dddot{q}_i$$

$$+ \sum_{j=1}^{3} \sum_{k=1}^{3}(C_{ijk}\ddot{q}_j \dot{q}_k + C_{ijk}\dot{q}_j\ddot{q}_k + \dot{C}_{ijk}\dot{q}_j\dot{q}_k) + H_i\ddot{q}_i]/k_i^t$$

□ Eqn 39.

This is the third order model equation for each primary joint of the PUMA 560.

5.1.2 Matrix Representation of Robot Manipulator Model

By observation, it can be seen that equation 39 may be rewritten in matrix form. A method described by Anderson[77] suggests the following notation, which has been adopted:

LMAT = Diagonal(L_1/k_1^t, L_2/k_2^t, L_3/k_3^t)

RMAT = Diagonal(R_1/k_1^t, R_2/k_2^t, R_3/k_3^t)

HMAT = Diagonal(H_1, H_2, H_3)

IMAT = Diagonal(I_{a1}, I_{a2}, I_{a3})

KMAT = Diagonal($N_1k_1^e$, $N_2k_2^e$, $N_3k_3^e$)

$\underset{\sim}{G}$ = Gravity Vector(G_1, G_2, G_3)

D = matrix which contains all the effective and coupling inertial terms,
D^1 = matrix which contains the centripetal and coriolis forces experienced by joint 1,
D^2 = matrix which contains the centripetal and coriolis forces experienced by joint 2,
D^3 = matrix which contains the centripetal and coriolis forces experienced by joint 3.

The simulator is defined as having three inputs (actuator voltages) and three outputs types, namely joint accelerations, velocities and positions. The simulator is designed to aid in the evaluation of possible control algorithms and to decide their suitability for manipulator control. It is desired to have a simulator that will run as closely as possible to real time.

Hence equation 39 can be rewritten as :

$$\begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \mathrm{LMAT}.\left[\, D + \mathrm{IMAT} \,\right].\begin{bmatrix} \dot{q}_7 \\ \dot{q}_8 \\ \dot{q}_9 \end{bmatrix} +$$

$$\Big(\, \mathrm{LMAT}.\dot{D} + \mathrm{RMAT}.\{\, D + \mathrm{IMAT} \,\} + \mathrm{HMAT} \; +$$

$$\mathrm{LMAT}.\begin{bmatrix} (q_4,\ q_5,\ q_6).D^1 \\ (q_4,\ q_5,\ q_6).D^2 \\ (q_4,\ q_5,\ q_6).D^3 \end{bmatrix} \Big).\begin{bmatrix} q_7 \\ q_8 \\ q_9 \end{bmatrix} +$$

$$\Big(\, \mathrm{LMAT}.\begin{bmatrix} (q_7,\ q_8,\ q_9).D^1 \\ (q_7,\ q_8,\ q_9).D^2 \\ (q_7,\ q_8,\ q_9).D^3 \end{bmatrix} + \mathrm{RMAT}.\begin{bmatrix} (q_4,\ q_5,\ q_6).D^1 \\ (q_4,\ q_5,\ q_6).D^2 \\ (q_4,\ q_5,\ q_6).D^3 \end{bmatrix}$$

$$+ \mathrm{LMAT}.\begin{bmatrix} (q_4,\ q_5,\ q_6).\dot{D}^1 \\ (q_4,\ q_5,\ q_6).\dot{D}^2 \\ (q_4,\ q_5,\ q_6).\dot{D}^3 \end{bmatrix} + \mathrm{RMAT}.\mathrm{HMAT} + \mathrm{KMAT}\Big).\begin{bmatrix} q_4 \\ q_5 \\ q_6 \end{bmatrix}$$

$$+ \mathrm{LMAT}.\dot{\underset{\sim}{G}} + \mathrm{RMAT}.\underset{\sim}{G}$$

□ Eqn 40a.

The following quantities are defined to simplify the model equation :

$$\underline{D} = \mathrm{LMAT}.\left[\, D + \mathrm{IMAT} \,\right]$$

□ Eqn 40b.

$\underline{P}(q) =$

$$\Big(\text{LMAT}.\dot{D} + \text{RMAT}.\{ D + \text{IMAT} \} + \text{HMAT} \; +$$

$$\text{LMAT}.\begin{bmatrix} (q_4, q_5, q_6).D^1 \\ (q_4, q_5, q_6).D^2 \\ (q_4, q_5, q_6).D^3 \end{bmatrix} \Big).\begin{bmatrix} q_7 \\ q_8 \\ q_9 \end{bmatrix} +$$

$$\Big(\text{LMAT}.\begin{bmatrix} (q_7, q_8, q_9).D^1 \\ (q_7, q_8, q_9).D^2 \\ (q_7, q_8, q_9).D^3 \end{bmatrix} + \text{RMAT}.\begin{bmatrix} (q_4, q_5, q_6).D^1 \\ (q_4, q_5, q_6).D^2 \\ (q_4, q_5, q_6).D^3 \end{bmatrix}$$

$$+ \text{LMAT}.\begin{bmatrix} (q_4, q_5, q_6).\dot{D}^1 \\ (q_4, q_5, q_6).\dot{D}^2 \\ (q_4, q_5, q_6).\dot{D}^3 \end{bmatrix} + \text{RMAT}.\text{HMAT} + \text{KMAT}\Big).\begin{bmatrix} q_4 \\ q_5 \\ q_6 \end{bmatrix}$$

$$+ \text{LMAT}.\dot{\underset{\sim}{G}} + \text{RMAT}.\underset{\sim}{G}$$

□ Eqn 40c.

Hence the model equation can be written as :

$$\begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \underline{D}.\begin{bmatrix} \dot{q}_7 \\ \dot{q}_8 \\ \dot{q}_9 \end{bmatrix} + \underline{P}(q)$$

□ Eqn 41.

Rearranging one gets :

$$\Rightarrow \begin{bmatrix} \dot{q}_7 \\ \dot{q}_8 \\ \dot{q}_9 \end{bmatrix} = - \underline{D}^{-1}.\underline{P}(q) + \underline{D}^{-1}.\begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix}$$

□ Eqn 42.

The following relationships are a consequence of the definitions of joint position, velocity and acceleration:

$$\begin{aligned} \dot{q}_1 &= q_4 & \qquad \dot{q}_4 &= q_7 \\ \dot{q}_2 &= q_5 & \qquad \dot{q}_5 &= q_8 \\ \dot{q}_3 &= q_6 & \qquad \dot{q}_6 &= q_9 \end{aligned}$$

Hence the full ninth order comprehensive model for the first three joints of the

PUMA 560 can be written in state space format:

$$\begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \hdashline \dot{q}_4 \\ \dot{q}_5 \\ \dot{q}_6 \\ \hdashline \dot{q}_7 \\ \dot{q}_8 \\ \dot{q}_9 \end{bmatrix} = \begin{bmatrix} q_4 \\ q_5 \\ q_6 \\ \hdashline q_7 \\ q_8 \\ q_9 \\ \hdashline \\ -\underline{D}^{-1}.\underline{P}(q) \\ \\ \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \hdashline 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \hdashline & & \\ & \underline{D}^{-1} & \\ & & \end{bmatrix} . \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix}$$

□ Eqn 43.

The state vector for the model is q ($\in R^9$), where,

$$q = [\ q_1\ q_2\ q_3\ q_4\ q_5\ q_6\ q_7\ q_8\ q_9\ ]^T.$$

Note that $\underline{P}(q)$ is a vector whose elements are dependent on the vector q and the manipulator parameters,

$$\underline{P}(q) \in R^3.$$

This vector is complex and requires considerable processor time to compute at each interval. It is very nonlinear, and the sine and cosine functions are required to calculate the elements of the inertial, centripetal and coriolis matrices. Gravity terms are also a nonlinear element.

$\underline{D}$ is a matrix whose elements are dependent upon the vector q and the manipulator parameters,

$$\underline{D} \in R^{3X3}.$$

This matrix is derived from two static matrices and D, the inertial matrix which is dependent on the state vector.

5.1.3 Computational Model for Computer simulation

Possessing a model for the PUMA 560 manipulator in matrix form is a vital step towards computer simulation of the arm. There are several outstanding aspects yet to be considered, however. The first concern in implementing the state description of equation 43 is to find an adequate numerical integration technique which is suitable for digital computer applications. A numerical integration method is used to calculate the system states at time k, based on the values of states at time k-1, and the present system inputs. The simplest technique is the Euler method. This approximates the curve $x = f(t)$ by a polygon whose slope, at each time t_r is given by the tangent to the curve $x = f(t)$ at t_r. It is a first order method with a truncation error per step of order h^2. Errors occur because the slope of f(t) changes over the interval h. A better approximation of the slope, over the interval, will result in a closer estimate of the function.

A much more accurate system is to implement fourth order Runge-Kutta numerical integration. This method takes a weighted sum of slopes about each point, denoted by t, and is a self-starting technique. The step size is easily changed between iterations, and stability is one of its more desirable properties. Runge-Kutta integration is also quite suited to systems with piece-wise constant inputs, as would be the case under normal simulation conditions.

For the nth order equation written as :

$$\dot{x} = f(x,t)$$

$$\dot{x}_i = f_i(x_1, x_2, \dots x_n, t) \qquad 1 \leqslant i \leqslant n,$$

the formula for advancing the solution one step is :

$$x_{i,r+1} = x_{i,r} + (k_{i1} + 2(k_{i2} + k_{i3}) + k_{i4})/6$$

where,

$$x_{i,r+1} = x_i(t_{r+1}) = x_i(t_0 + (r+1)h)$$

$$k_{i1} = hf_i(x_{1,r}, x_{2,r}, \dots x_{n,r}, t_r)$$
$$k_{i2} = hf_i(x_{1,r} + 0.5k_{11}, \dots x_{n,r} + 0.5k_{n1}, t_{r+0.5})$$
$$k_{i3} = hf_i(x_{1,r} + 0.5k_{12}, \dots x_{n,r} + 0.5k_{n2}, t_{r+0.5})$$
$$k_{i4} = hf_i(x_{1,r} + 0.5k_{13}, \dots x_{n,r} + 0.5k_{n3}, t_{r+1})$$

For the nth order system with an external input :

$$\dot{x} = f(x,u,t)$$

$$\dot{x}_i = f_i(x_1,\ x_2,\ \ldots\ x_n,\ u_i,\ t) \qquad 1 \leqslant i \leqslant n,$$

the Runge–Kutta algorithm must be altered.

If the system has an external input then the function must also be differentiated with respect to the input. When the input is held constant over the interval then the partial derivatives with respect to the input will be zero giving no cause for adapting the standard formula. Thus the input will be treated like a system state and the following solution applies :

$$\dot{x}_i = f_i(x_1,\ x_2,\ \ldots\ x_n,\ u_i,\ t) \qquad 1 \leqslant i \leqslant n,$$

the formula for advancing the solution one step is

$$x_{i,r+1} = x_{i,r} + (k_{i1} + 2(k_{i2} + k_{i3}) + k_{i4})/6$$

where,

$$x_{i,r+1} = x_i(t_{r+1}) = x_i(t_0 + (r+1)h)$$

$$k_{i1} = hf_i(x_{1,r},\ x_{2,r},\ \ldots\ x_{n,r},\ u_{i,r},\ t_r)$$
$$k_{i2} = hf_i(x_{1,r} + 0.5k_{11},\ \ldots\ x_{n,r} + 0.5k_{n1},\ u_{i,r+0.5},\ t_{r+0.5})$$
$$k_{i3} = hf_i(x_{1,r} + 0.5k_{12},\ \ldots\ x_{n,r} + 0.5k_{n2},\ u_{i,r+0.5},\ t_{r+0.5})$$
$$k_{i4} = hf_i(x_{1,r} + 0.5k_{13},\ \ldots\ x_{n,r} + 0.5k_{n3},\ u_{i,r+1},\ t_{r+1})$$

This set of equations is implemented in the simulation of the PUMA 560 robot arm.

A further concern in the simulator implementation is the matrix inversion routine. Execution of equation 43 involves inverting the $\underline{D}$ matrix (as defined in equation 40b). It needs to be shown that the inversion will not fail due to non-singularity. Recall that $\underline{D}$ is defined as;

$$\underline{D} = \mathrm{LMAT}.[\ D + \mathrm{IMAT}\ ]$$

Where the matrices are defined as:

$$\text{LMAT} = \text{Diagonal}(\ L_1/k_1^t,\ L_2/k_2^t,\ L_3/k_3^t\ )$$

$$\text{IMAT} = \text{Diagonal}(\ I_{a1},\ I_{a2},\ I_{a3}\ )$$

Both LMAT AND IMAT are diagonal matrices, ($\in \mathbb{R}^{3x3}$). The elements of LMAT are quotients of the motor equivalent circuit parameters, i.e. armature inductance divided by the torque constant for the motor. The first three motors of the PUMA 560 robot are, for practical purposes, identical and thus have identical equivalent circuit parameters. Thus, in the case of the PUMA 560 arm LMAT is merely a positively scaled version of the identity matrix. IMAT contains the reflected motor inertias, which by definition are positive, and incidentally, due to manipulator design are comparatively large. The remaining matrix for consideration is the D matrix. This is defined previously as:

$$D(q) = [d_{ij}]$$

and is symmetric as determined by Bejczy[75] and is also positive definite:

i.e.

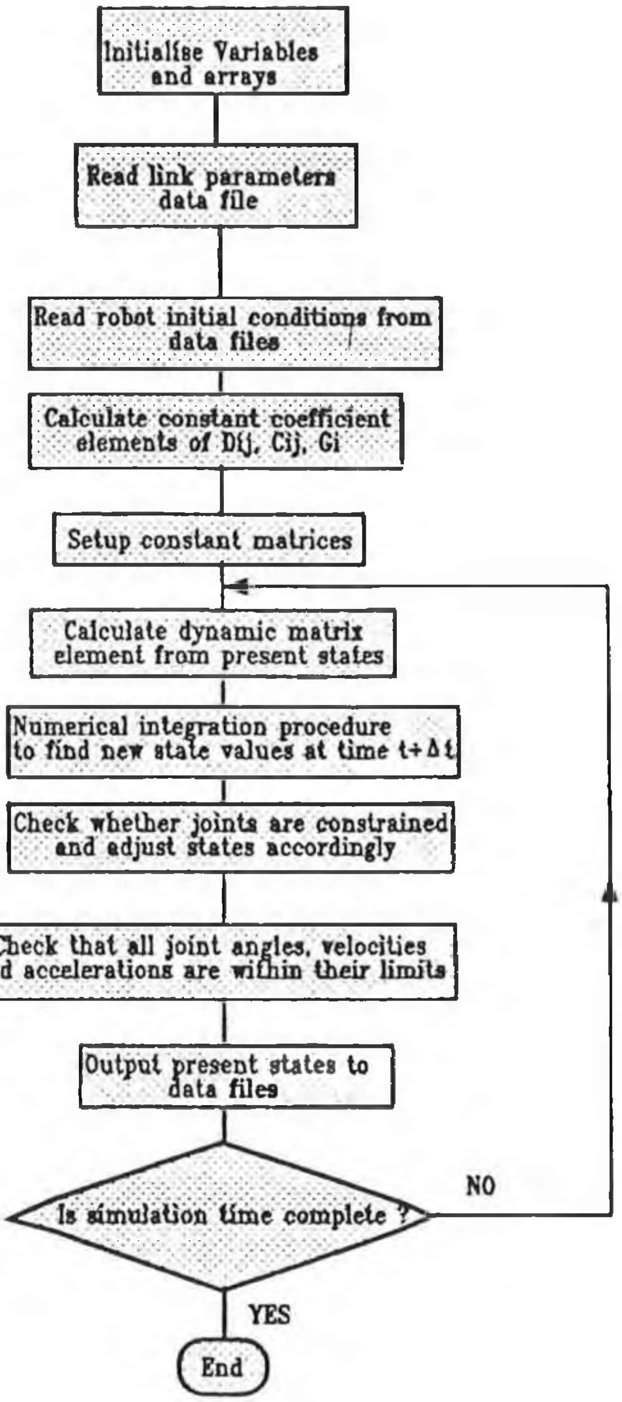
$$d_{ij} = d_{ji} \qquad \text{for} \quad \text{all } i,j$$

and,

$$d_{ii} > 0 \qquad \text{for all } i$$
$$d_{ij}^2 < d_{ii}d_{jj} \qquad \text{for all } i,j$$

Therefore it is deduced that $\underline{D}$ is positive definite from the above definitions of the constituent matrices LMAT, IMAT and D, over the space of operation defined by the positive values of elements in the equivalent model of the joint motors. Hence, $\underline{D}$ is defined as being non-singular. One final consideration in the inversion routine is from a practical point of view, rather than a theoretical one. Round-off errors must be kept small during the inverse evaluation. In Gauss Jordan elimination round off errors occur when there are large magnitude differences in the elements of matrix columns. A technique to reduce this effect is known as *partial pivoting*, which involves taking the largest element in the matrix column as the pivotal coefficient. In this manner round off was found to be satisfactory when testing matrix inverses with elements differing in the order of 10^3 using double precision arithmetic. Single precision reduced accuracy by about 15%.

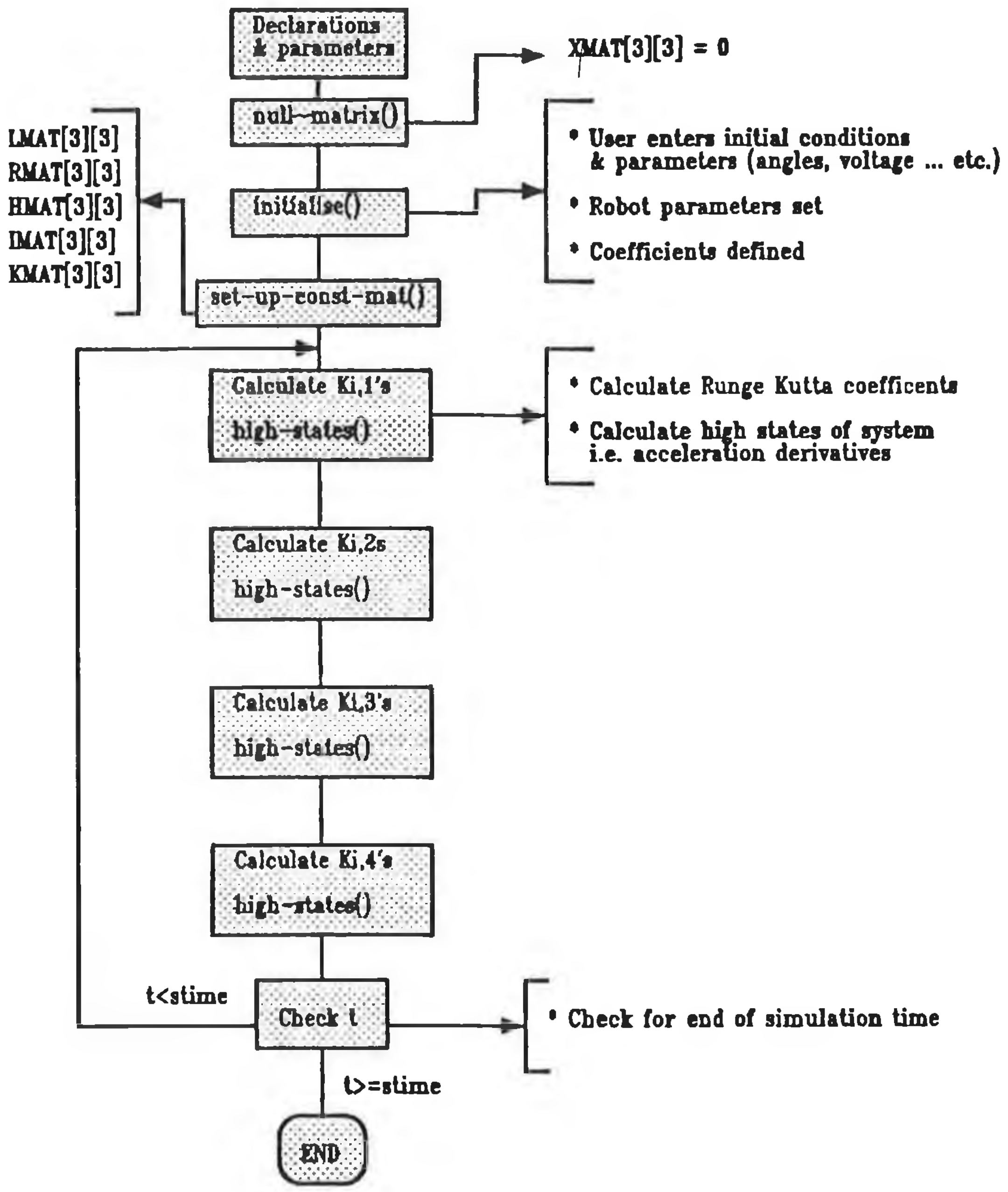
The Runge Kutta integration technique described above combined with the robot model in a state-space form that is decomposed into algebraic matrix operations provide adequate mechanisms for computer simulation of the PUMA 560 manipulator, as will be shown in this section.


Initialise Variables and arrays
Read link parameters data file
Read robot initial conditions from data files
Calculate constant coefficient elements of Dij, Cij, Gi
Setup constant matrices
Calculate dynamic matrix element from present states
Numerical integration procedure to find new state values at time t+Δt
Check whether joints are constrained and adjust states accordingly
Check that all joint angles, velocities and accelerations are within their limits
Output present states to data files
Is simulation time complete ?
NO
YES
End

Flowchart of PUMA 560 robot arm simulator

FIGURE 19

Figure 19 shows the flowchart of operations required to simulate the forward dynamics of the manipulator, as derived in sections 5.1.1 and 5.1.2. The initial variable and array assignment is the first phase followed by the reading of fixed robot parameters, those estimated by bejczy[75], from data files. The evaluation of the inertial, centripetal, coriolis and gravitational terms is accomplished by implementing equations 19 to 37, using the read parameters. Before the algorithm enters the simulation loop the constant matrices are evaluated. These matrices include the parameters which model the dc motors of the manipulator joints, and will be explained more fully in this section when the software implementation is discussed. The next step, the beginning of the simulator loop, is to implement the robot dynamical state equations, and solve for the differential of the states in the ninth order model given by equation 43.



Flowchart illustration of the software computational model for PUMA 560 simulation

FIGURE 20

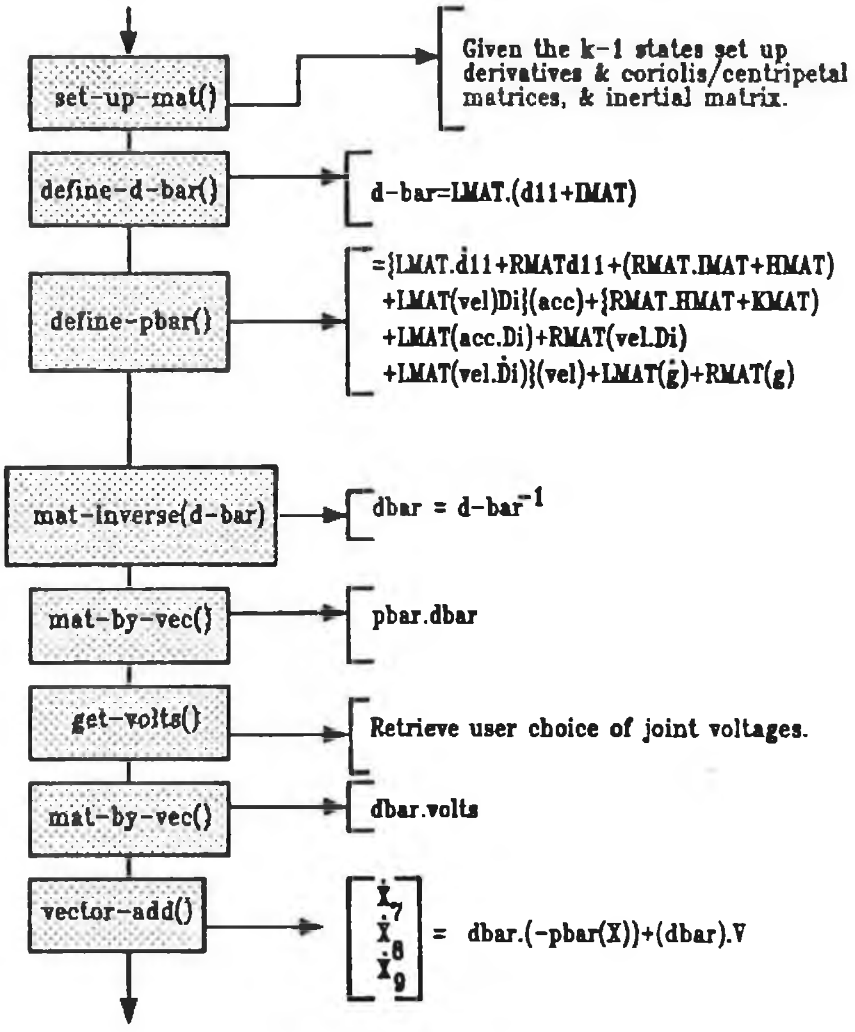
The highest 3 states represent the derivatives of the accelerations of the three joints and a fourth order Runge Kutta numerical integration technique is used to solve for the acceleration values. When the states have been solved the current positions, velocities and accelerations of the joints are updated. The following step is to check whether the joint positions are within physical limits, and if not to adjust them accordingly. Finally after these current values for position, velocity and acceleration have been recorded a time check is made to see if the simulation time is complete. If so the loop is terminated and the simulation ends, otherwise the cycle rebegins.

The software model for the simulator, which is derived from the computational model of Figure 19, is illustrated in Figure 20. Functions are defined as independent routines, containing one or more tasks (not necessarily an ADA-type task), which may be evoked during the execution of the main function. A function is defined by the notation *function()*. The actual simulator was initially written in the C language for validation using the TurboC editor (derived from ANSI C)[80], but the software model will be explained from an objective viewpoint, not necessitating a knowledge of C.

As before the initial task is to create the arrays and variables necessary for computation. The state vector is stored in XMAT[3][3] and this is initially assigned a null value, by calling the function null-matrix(). The initialise() routine prompts the user for several items of information: the initial positions of the robot manipulator joints, the sample period for simulation, the simulation duration time and finally the voltages to be applied to the first three joints. Further to this it also evaluates the robot parameters including the coriolis, centripetal, inertial and gravitational forces. The equations used are those given by equations 19 to 37. Based on these parameters the function set-up-const-mat() constructs the constant system matrices, LMAT (inductance matrix), RMAT (resistance matrix), HMAT (frictional matrix), IMAT (inertial matrix) and KMAT (torque constant matrix). At this point it is possible to construct the state space model for the system using the parameters and matrices calculated previously. The loop in the simulation is based upon solving this state space description. Fourth order Runge Kutta is used to determine the integral value of the differentiated states. The function high-states(), which is one of the most computationally intensive functions in the programme is used to evaluate the ninth order state model, given in equation 43. This function can be decomposed into a series of sub-functions. Figure 21 is a graphical illustration of the sequential flow of operations and sub-functions within the function high-states().

There is considerable parallelism inherent in the computations of high-states(), which will be examined in Section 5.2.1 and quantified numerically in Section 5.5. The sequential flow of processing will only be considered in this Section, however. Looking at Figure 21 the first function to be executed is set-up-mat(). This creates the coriolis, centripetal and inertial matrices and evaluates their derivatives. The following two functions are define-d-bar() and define-pbar() which are implementations of equations 40b and 40c respectively, which evaluate $\underline{D}$ and $\underline{P}(q)$.

The calculations involved in define-pbar() are substantially more significant than those of define-b-bar(). In order to evaluate the ninth order state space model of equation 43 a matrix inversion must be performed on the quantity d-bar. This is achieved by the routine mat-inverse() and the result is multiplied by pbar using mat-by-vec().


set-up-mat()
Given the k-1 states set up derivatives & coriolis/centripetal matrices, & inertial matrix.
define-d-bar()
d-bar=LMAT.(d11+IMAT)
define-pbar()
={LMAT.ḋ11+RMATd11+(RMAT.IMAT+HMAT)+LMAT(vel)Di}(acc)+{RMAT.HMAT+KMAT)+LMAT(acc.Di)+RMAT(vel.Di)+LMAT(vel.Ḋi)}(vel)+LMAT(ġ)+RMAT(g)
mat-inverse(d-bar)
dbar = d-bar⁻¹
mat-by-vec()
pbar.dbar
get-volts()
Retrieve user choice of joint voltages.
mat-by-vec()
dbar.volts
vector-add()
Ẋ7 Ẋ8 Ẋ9 = dbar.(-pbar(X))+(dbar).V

Sequential flowchart of function high-states().

FIGURE 21

The system input of the state model is defined as the three joint voltages, and therefore these must be retrieved from the stored user input. This is achieved by the simple routine get-volts(). Following two elementary matrix/vector operations, multiply and add, the derivatives of the three highest states are evaluated. On completion of high-states() these values are passed, using memory pointers[79], back into the main function, main(), whereupon they are used as inputs to the Runge Kutta routine. The high-states() function is called from the main function four times during the simulation loop, due to the order of the numerical integration routine. Section 5.2 examines this computational model for parallelism and qualifies the different granularity of parallelism present. The proceeding Section, Section 5.3, looks at the software aspects of parallelising the simulator, and develops coding in parallel C for the concurrent processes. A hardware implementation scheme is proposed in Section 5.4 and this is furthered by Section 5.5, the penultimate Section of this Chapter, which tackles the numerical analysis of the parallel simulator performance, yielding indices such as parallelism, bounded parallelism, processor efficiency and speedup. It also validates the model by examining its performance under particular input tests and analysing the output in graphical form.

5.2 PARALLELISM INHERENT IN COMPUTATIONAL MODEL

To examine for parallelism one must not merely examine the computational model but must also look carefully at the actual software implementation of the simulator, so that there is no dichotomy between the theoretical parallelism and that which is practical within the software model. This Section examines the software resulting from the model in matrix form, and attempts to quantify the parallelism of the calculations objectively.

The flowchart of the sequential model, Figure 20, shows clearly the looping nature of the simulator with initial assignment overheads. This loop is traversed several hundred times in a typical simulation run. The functions and calculations executed within the loop are therefore the main target of parallel decomposition, and not the initial matrix calculations and variable assignments, which are only performed once per run. The two tasks of interest then, are the Runge Kutta routine and the high-states() function. Firstly, the Runge Kutta numerical integration routine is examined. For a system with an external input, which is constant over the period of integration, the following equations apply:

$$\dot{x}_i = f_i(x_1, x_2, \dots x_n, u_i, t) \qquad 1 \leqslant i \leqslant n,$$

the formula for advancing the solution one step is

$$x_{i,r+1} = x_{i,r} + (k_{i\,1} + 2(k_{i\,2} + k_{i\,3}) + k_{i\,4})/6$$

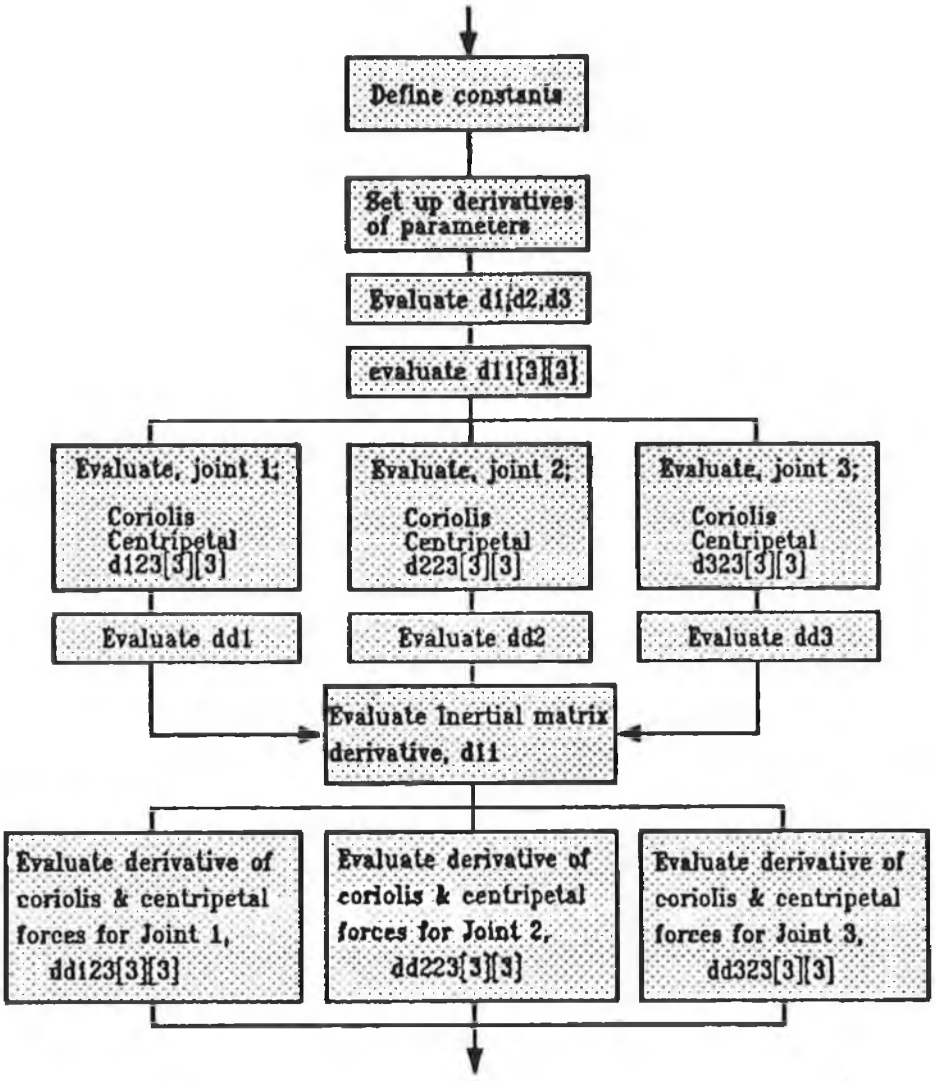
where,

$$x_{i,r+1} = x_i(t_{r+1}) = x_i(t_0 + (r+1)h)$$

$$k_{i1} = hf_i(x_{1,r}, x_{2,r}, \dots x_{n,r}, u_{i,r}, t_r)$$
$$k_{i2} = hf_i(x_{1,r} + 0.5k_{11}, \dots x_{n,r} + 0.5k_{n1}, u_{i,r+0.5}, t_{r+0.5})$$
$$k_{i3} = hf_i(x_{1,r} + 0.5k_{12}, \dots x_{n,r} + 0.5k_{n2}, u_{i,r+0.5}, t_{r+0.5})$$
$$k_{i4} = hf_i(x_{1,r} + 0.5k_{13}, \dots x_{n,r} + 0.5k_{n3}, u_{i,r+1}, t_{r+1})$$

These equations are highly sequential, in fact sequentiality is an integral part of any numerical integration technique. In order to evaluate a state from its derivative its previous value and the relevant four Runge Kutta parameters must be known. The Runge Kutta technique doesn't lend itself to parallelisation methods, but were it neccessary (and it will seen not to be) it would be suitable for pipelining. It would also be possible to implement the elementary coefficient evaluations on a systolic array type architecture, such as a PACE chip[81,82,83]. The various parameters could be fed in from previous calculations, the present coefficients calculated and these then used to estimate the states for the state space model evaluation and finally fed into the systolic array for the next sample time evaluations, and so on until completion of simulation time. This would, however, be quite exorbitant when one considers the added complexity in incorporating a systolic array into a coarse grain multiprocessor system (used for executing the larger grain parallelism of the other functions), and that the Runge Kutta evaluations only constitute 8% of the calculation overheads within the simulator software loop, thus offering no more than an maximum increase of 0.08 to a speedup of one, or 0.3 to a speedup of 2. It is concluded that the Runge Kutta routine remains sequential as no significant benefit is gained in altering it otherwise.

One's attention turns therefore to the only other function in the loop; high-states(). The overall speedup depends on how successful the decomposition of this function is. The sequential representation of this function is illustrated in block diagram form in Figure 21, Section 5.1.2. As can be seen, it is comprised of eight subfunctions, which generally differ considerably in their execution times. The first task is the setting up of the time variant matrices, accomplished by a call to the function set-up-mat(). This function initially defines the coefficients used in the formulation of the matrices, sets up parameter derivatives and evaluates the inertial coupling parameters between the joints and stores them in the matrix, d11. The next step in the sequential implementation is to evaluate the coriolis and centripetal forces on each of the joints. It is possible to perform these calculations separately for each of the three joints and their decoupled nature is justified through graphical analysis in Section 5.5. Figure 22 shows these stages of operation in graphical form. Following the coriolis and centripetal calculations the inertial coupling matrix derivative is evaluated. The final task of function set-up-mat() is to calculate and store the derivatives of the coriolis and centripetal forces on the three joints. Once again three separate computational paths may be established, one for each joint.


Define constants
Set up derivatives of parameters
Evaluate d1,d2,d3
evaluate d11[3][3]
Evaluate, joint 1; Coriolis Centripetal d123[3][3]
Evaluate, joint 2; Coriolis Centripetal d223[3][3]
Evaluate, joint 3; Coriolis Centripetal d323[3][3]
Evaluate dd1
Evaluate dd2
Evaluate dd3
Evaluate Inertial matrix derivative, d11
Evaluate derivative of coriolis & centripetal forces for Joint 1, dd123[3][3]
Evaluate derivative of coriolis & centripetal forces for Joint 2, dd223[3][3]
Evaluate derivative of coriolis & centripetal forces for Joint 3, dd323[3][3]

Block diagram illustration of parallel form of function set-up-mat().
FIGURE 22

On completion of this function, whereupon the various forces which the joints are subject to are stored in matrix form in memory, the control of programme flow is returned to the function high-states(). The reduction in run-time for the function set-up-mat() based on the separate evaluation of link forces is examined and quantified in Section 5.5. The next two functions to be executed in the sequential flowchart of high-states() are define-d-bar() and define-pbar(). Both these functions require common variables but do not alter their respective values, thus can be run independently. Unfortunately the run times of the two function are dramatically different, and in fact executing both concurrently would give a moderate speedup of 1.08 over the sequential execution, as illustrated in Section 5.5. It is possible however to decompose define-pbar() into two concurrent processes. Recall the definition of $\underline{P}(q)$, (p-bar), shown overleaf:

$$\underline{P}(q) =$$

$$\Big(\mathrm{LMAT}.\dot{D} + \mathrm{RMAT}.\{ D + \mathrm{IMAT} \} + \mathrm{HMAT} \; +$$

$$\mathrm{LMAT}.\begin{bmatrix} (q_4, q_5, q_6).D^1 \\ (q_4, q_5, q_6).D^2 \\ (q_4, q_5, q_6).D^3 \end{bmatrix} \Big) . \begin{bmatrix} q_7 \\ q_8 \\ q_9 \end{bmatrix} +$$

$$\Big(\mathrm{LMAT}.\begin{bmatrix} (q_7, q_8, q_9).D^1 \\ (q_7, q_8, q_9).D^2 \\ (q_7, q_8, q_9).D^3 \end{bmatrix} + \mathrm{RMAT}.\begin{bmatrix} (q_4, q_5, q_6).D^1 \\ (q_4, q_5, q_6).D^2 \\ (q_4, q_5, q_6).D^3 \end{bmatrix} +$$

cont'd→

$$\mathrm{LMAT}.\begin{bmatrix} (q_4, q_5, q_6).\dot{D}^1 \\ (q_4, q_5, q_6).\dot{D}^2 \\ (q_4, q_5, q_6).\dot{D}^3 \end{bmatrix} + \mathrm{RMAT.HMAT} + \mathrm{KMAT} \Big) . \begin{bmatrix} q_4 \\ q_5 \\ q_6 \end{bmatrix}$$

$$+ \mathrm{LMAT}.\dot{\underset{\sim}{G}} + \mathrm{RMAT}.\underset{\sim}{G}$$

□ Eqn 44.

From this definition the calculations can be partitioned into two equations with the first equation defined as follows;

$$\underline{P_1}(q) =$$

$$\Big(\mathrm{LMAT}.\dot{D} + \mathrm{RMAT}.\{ D + \mathrm{IMAT} \} + \mathrm{HMAT} \; +$$

$$\mathrm{LMAT}.\begin{bmatrix} (q_4, q_5, q_6).D^1 \\ (q_4, q_5, q_6).D^2 \\ (q_4, q_5, q_6).D^3 \end{bmatrix} \Big) . \begin{bmatrix} q_7 \\ q_8 \\ q_9 \end{bmatrix}$$

$$+ \mathrm{LMAT}.\dot{\underset{\sim}{G}} + \mathrm{RMAT}.\underset{\sim}{G}$$

□ Eqn 45.

The second equation is defined to be $\underline{P_2}(q)$ and is thus given as the remainder of equation 44;

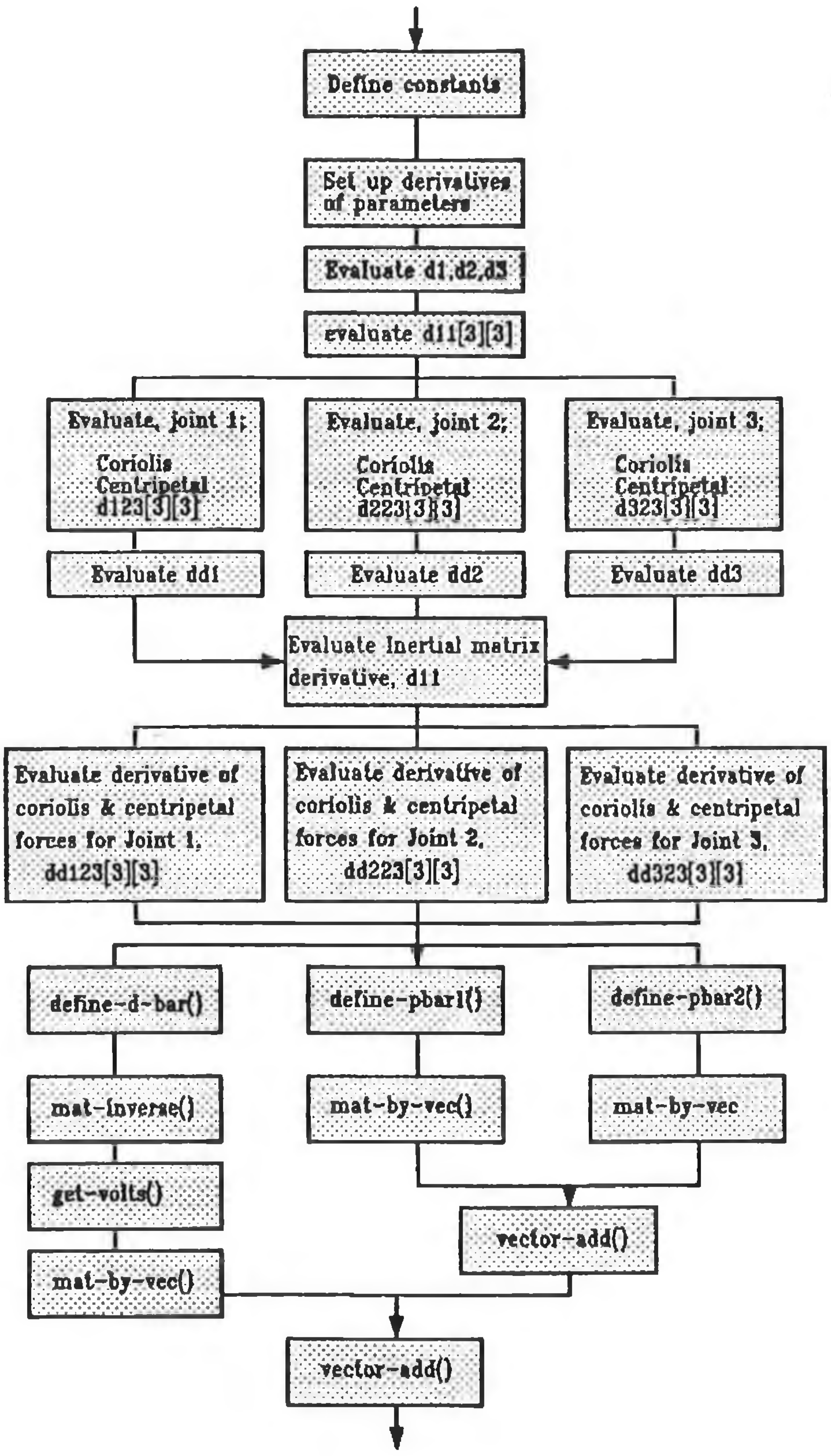
$$\underline{P_2}(q) =$$

$$\Big(\mathrm{LMAT}.\begin{bmatrix} (q_7, q_8, q_9).D^1 \\ (q_7, q_8, q_9).D^2 \\ (q_7, q_8, q_9).D^3 \end{bmatrix} + \mathrm{RMAT}.\begin{bmatrix} (q_4, q_5, q_6).D^1 \\ (q_4, q_5, q_6).D^2 \\ (q_4, q_5, q_6).D^3 \end{bmatrix}$$

$$+ \mathrm{LMAT}.\begin{bmatrix} (q_4, q_5, q_6).\dot{D}^1 \\ (q_4, q_5, q_6).\dot{D}^2 \\ (q_4, q_5, q_6).\dot{D}^3 \end{bmatrix} + \mathrm{RMAT.HMAT} + \mathrm{KMAT} \Big) . \begin{bmatrix} q_4 \\ q_5 \\ q_6 \end{bmatrix}$$

□ Eqn 46.

The two resulting processes are notated as define-p1bar() and define-p2bar(). By executing these two processes in parallel along with define-d-bar() results in speedup of almost 2.2 over the serial execution time, yielding a processor efficiency of more than 0.7.


Define constants
Set up derivatives of parameters
Evaluate d1,d2,d3
evaluate d11[3][3]
Evaluate, joint 1; Coriolis Centripetal d123[3][3]
Evaluate, joint 2; Coriolis Centripetal d223[3][3]
Evaluate, joint 3; Coriolis Centripetal d323[3][3]
Evaluate dd1
Evaluate dd2
Evaluate dd3
Evaluate Inertial matrix derivative, d11
Evaluate derivative of coriolis & centripetal forces for Joint 1, dd123[3][3]
Evaluate derivative of coriolis & centripetal forces for Joint 2, dd223[3][3]
Evaluate derivative of coriolis & centripetal forces for Joint 3, dd323[3][3]
define-d-bar()
define-pbar1()
define-pbar2()
mat-inverse()
mat-by-vec()
mat-by-vec
get-volts()
vector-add()
mat-by-vec()
vector-add()

Parallel Decomposition of function high-states().

FIGURE 23

The remaining matrix/vector operations are relatively fast, with the exception of the inversion routine, which must be performed on $\underline{D}$. Examining equation 43, it is observed that only the inverse of $\underline{D}$, and not $\underline{D}$ itself, is required for state evaluation, and therefore this may be evaluated in parallel with the evaluation of $\underline{P}_1(q)$ and $\underline{P}_2(q)$. This further increases speedup and efficiency since the processor previously dedicated to solving $\underline{D}$, would have spent some time awaiting the other two processors, assigned to $\underline{P}_1$ and $\underline{P}_2$, to finish and now has an additional task to occupy some of its waiting time.

The get-volts(), mat-by-vec() and vector-add() functions have reasonably short execution times due to their simplicity, and have a limited effect on speedup and efficiency, but nonetheless there is some scope to run them in parallel. This is illustrated clearly in Figure 23 which displays the coarse grain parallelism inherent in the high-states() function.

The timing indices for high-states() and its sub-functions are contained in section 5.5 and an evaluation of speedup and processor efficiency based on these values is derived. It is worth noting at this stage that the maximum parallelism in the parallelised simulator is in fact equal to the number of joints being simulated, three. The objective is therefore to attempt to achieve speedup as close to this figure as possible.

Another method of approach to implementing the simulator in parallel involves the pipelining of the main simulation loop. The loop, as illustrated in Figure 20, calls the function high-states() four times and interspersed between these calls the Runge Kutta numerical integration coefficients are evaluated. The pipeline would optimally consist of four stages, each executing a high-states() function and evaluating a set of Runge Kutta coefficients. Because the input to the pipeline would be derived from the previously generated pipeline output the initial four inputs would be estimates (because for the first four sample periods the pipeline would have an inaccurate output), and this would result in a transient error not found in the sequential implementation. This is examined in more detail in Section 5.5. Further, because the maximum parallelism in the overall simulator is three, if a pipeline were implemented then for optimal speedup twelve processors would be necessary to achieve maximum speedup. As the number of processors is reduced speedup is reduced, (and processor efficiency is increased). This degradation in performance is chartered as a function of available processors in Section 5.5.

5.3 PARALLEL C MODEL OF PARALLEL SIMULATOR

Both the sequential and parallel models of the PUMA 560 simulator were extensively investigated in Sections 5.1 and 5.2. In this section Parallel C [84,85,Appendix B] is introduced and a scheme to implement the simulator in this language is developed.

Parallel C is, as one might expect, an extension of the C language to incorporate facilities to specify concurrency in processes. It is designed specifically for implementation on a target Transputer[86,..,89] system. A more commonly used language on Transputer systems is the Occam [Appendix B] programming language, developed by INMOS. It offers the user some powerful constructs. Occam also allows for the channel data type and gives the user some control over the placement of processes on different Transputers, but has several disadvantages. For one, it is a new language, with a small existing base of source code and programming know-how. Another problem is the static nature of the language: every variable must be allocated at compile time and no recursion is allowed. Although some claim that any problem can be solved without dynamic allocation of variables or the use of recursion, it is often much easier and less complex to use recursion. Occam also has a smaller set of statements and operators than many more popular languages. On the other hand, the programming language C is well-known to software developers and is particularly suited to systems programming and the handling of matrix and array operations. Vast source code libraries for the C language can be obtained, and unlike Occam, a C programme is dynamic. It provides for recursion and dynamic allocation of variables, and has powerful expression syntax and range of statements. The only thing C lacks is a dedicated set of statements and operators for the Transputer.

To obtain both the advantages of the C language and the Transputer architecture, C had to be extended with some statements and datatypes dedicated to the Transputer. These extensions had to fit in with the C programming language in a logical way. A brief description of these extensions is perhaps appropriate at this point.

The first of the extensions to the C language is the channel datatype, a new data type providing synchronised communication between processes, and processors. A programmer uses channel variables to transmit data between separate processes running concurrently on the same Transputer as well as declaring the external links of each Transputer. From the programmers point of view, process-to-process communication is the same, regardless of whether the processes are on the same chip or not. The Transputer's specialised hardware handles both types of message sending with equal facility.

Using the keyword *channel*, one can declare a variable of the channel type;

```
channel ch1,ch2,ch3,ch4;
```

Each channel variable declared in this way, will occupy one word in memory. Sending a message with the aid of a channel variable is most simply performed with an assignment operator. Channels can also be used in expressions, just like any other data types. When channel variables are read they take their values from other processes rather from memory.

Another language extension is the link interface. Every Transputer has four high speed serial links, and each link is mapped to two internal channels. It is interesting to note how the link interfaces are integrated into the Transputer instruction set. A link behaves like any other channel. There are only a few differences. A link communicates only between processes on different Transputers, and are DMA based, so they consume no processor time. The links are placed at fixed addresses. Using pointers, it is not difficult to access the links from within a C programme. On conventional machines, a common technique to access a memory mapped I/O register is by creating a pointer to the type of the that register and initialising this pointer with the address of that I/O register. The same technique can be used to access the links on the Transputer:

```
channel *Link0Out = LINKOUT;
```

This declaration defines a pointer to the channel Link0Out. This value stands for the address of this link, address 0x80000000 on a T414 Transputer. To send information across this link, the following expression suffices:

```
*Link0Out = value;
```

This expression sends the value *value* over the link.

The system timer is a reserved word in the C compiler that can be accessed if it were an integer variable. The system clock, incremented every 64 microseconds (low priority process) or every microsecond (high priority process), is available as the timer variable. For example:

```
main()
      {
      .... int t;
      t = timer; ....
      }
```

This would simply load the current system timer value into t. Another powerful ability of the timer is to put a process *to sleep* for a determined amount of time. This allows a time oriented application to put itself to sleep to free up cpu time, and then reawaken itself to continue processing. A process puts itself to sleep, for example for xxx clock cycles, by the following command:

```
timer += xxx;
```

An essential addition to the C language for parallelisation is the *par* construct. The par construct allows the programmer to instantiate several parallel processes within the body of the programme. A par construct consists of the keyword *par*, followed by a complex statement:

```
par{
        statement1;
/* subprocess 1 */
        statement2;
/* subprocess 2 */
            .
            .
            .
        statementn;
/* subprocess n*/
}
```

Statements nested within the par construct statement are executed in parallel. A compiled par statement will continue to execute until all of these subprocesses have terminated. On the Transputer, the par statement translates into machine instructions quite easily. Each subprocess in a par statement is instantiated with the aid of only two Transputer instructions, namely the *startp* and the *endp* instruction. Naturally enough, the startp initiates a subprocess and at the end of a par statement each subprocess is terminated with a endp instruction. After the last subprocess has ended, the calling process resumes execution.

The *alt* construct is another important language extension for parallel programming. Like the par construct, it is closely tied to hardware features of the Transputer. The alt construct is used to wait for certain events and can also used instead of a standard C switch statement or nested if statements. The syntax of the alt mirrors that of the C's switch statement:

```
alt     {
        guard  guardexpression:
               code;
                 .
                 .
                 .
        guard  guardexpression:
                code;
        default:
                code;
        }
```

In the alt construct two new keywords appear: *alt* and *guard*. The keyword alt indicates the start of an alt statement. The guards compare to case labels in a switch statement. However there are some principle differences. Guards are evaluated at runtime whereas case statements are evaluated at compile time. Therefore, guard expressions are not restricted to constant expressions. They can use complex expressions that must be evaluated at runtime.

The guards in an alt statement are evaluated in the order in which they are placed within the alt braces. The result of a guard expression can be active or inactive. The code of the first guard that is active will be executed. If none of the guards is active, the process executing the alt will be descheduled until one of the channel or timer guards becomes active. There are five types of guard:

boolean guard
channel guard without boolean
channel guard with boolean
timer guard without boolean
timer guard with boolean

The boolean guard is the simplest of these different types of guards. A boolean guard consists of the keyword guard, followed by an expression with a boolean result. If the result of the boolean expression is true, the guard is active, otherwise the guard is inactive. The boolean guard can be used instead of nested if statements, for example:

```
if          (a<3) b=1;
else if     (a>8) b=2;
else if     (a>5) b=3;
```

is equivalent to, and can be replaced by:

```
alt {
        guard a<3: b=1; break;
        guard a>8: b=2; break;
        guard a>5: b=3; break;
      }
```

Channel guards are the second type of guards. The channel guard consists of an expression with a channel address and an optional boolean expression. The channel guard is active if the channel it points to is active (ready for input or output) and the optional boolean expression, if present, is true. The channel guard is very useful for multiplexing multiple internal channels to one external channel:

The third type of guard is the timer guard. Briefly, like the channel guard it may be used with or without a boolean expression. A timer guard, for example, can be used to timeout a channel. This can be done by using the timer guard in an alt with some channel guards. As a final alt example, suppose it is necessary to timeout a channel for 1000 internal clock cycles. This can be achieved in the following Parallel C coding extract:

```
alt {
      guard &intchannel: /* channel to wait for */
            outchannel = intchannel;
      guard timer=timer+1000: /* this guard if no message in 1000 cycles*/
      printf("Error: no response from channel\n");
      exit(0);
      break;
     }
```

By exploiting these extensions to the C instruction set processes can be specified to execute concurrently, or in parallel if the hardware to do so is available. If a multiprocessor hardware is not available (consider a uni-Transputer) then the processes are executed in a time-sharing regime. Using these extensions the coding of the PUMA 560 simulator in parallel may be accomplished with little difficulty. The initial C coding of the simulator, described in Section 5.2.3, can be modified by specifying the parallelism. Figure 20, which illustrates the sequential comptutational model, shows the sequence of process execution. Figures 22 and 23 detail, in graphical format, the parallelism in the functions set-up-mat() and high-states() respectively. These are the only functions along with define-pbar(), which can be decomposed into medium/large grain parallel processes. The sequential C programming of the function set-up-mat() is given in simplified form as:

```
set-up-mat( x1,x2,x3,x4,x5,x6 ) /*system states*/
 {
      define-constants(...);          /* Define sine and cosine constants*/
      par-derivative(...);            /* Set up parameter derivatives */
      eval-d(...);                    /* Evaluate d1, d2, d3 */
      eval-d11(...);                  /* Evaluate d11[3][3], inertial matrix */

      eval-d123(...);         /* Evaluate coriolis and centripetal */
                              /* forces for joint 1 */

      eval-d223(...);         /* Evaluate coriolis and centripetal */
                              /* forces for joint 2 */

      eval-d323(...);         /* Evaluate coriolis and centripetal */
                              /* forces for joint 3 */

      eval-dd1(...);          /* Evaluate derivative of d1 */

      eval-dd2(...);          /* Evaluate derivative of d2 */
```

```
        eval-dd3(...);              /* Evaluate derivative of d3 */

        eval-dd11(...);             /* Evaluate derivative of inertial */
                                    /* coupling matrix */
        eval-dd123(...);            /* Evaluate derivative of coriolis and */
                                    /* centripetal forces on joint 1 dd123[3][3]*/
        eval-dd223(...);            /* Evaluate derivative of coriolis and */
                                    /* centripetal forces on joint 2 d223[3][3]*/
        eval-dd323(...);            /* Evaluate derivative of coriolis and */
                                    /* centripetal forces on joint 3 d323[3][3]*/
   }
```

The above cited subfunctions are fictitious functions used for simplicity and represent blocks of coding in the actual programme. Based on Figure number 22 the processes can be reorganised in parallel form, and the software rewritten to accommodate this specification:

```
set-up-mat( x1,x2,x3,x4,x5,x6 ) /*First 6 system states*/
{
seq{
    define-constants(...);          /* Define sine and cosine constants*/
    par-derivative(...);            /* Set up parameter derivatives */

    eval-d(...);                /* Evaluate d1, d2, d3 */
    eval-d11(...);              /* Evaluate d11[3][3], inertial matrix */

    par{
      seq{
        eval-d123(...);             /* Evaluate coriolis and centripetal */
                                    /* forces for joint 1 */
        eval-dd1(...)               /* Evaluate derivative of d1 */
        }
      seq{
        eval-d223(...);             /* Evaluate coriolis and centripetal */
                                    /* forces for joint 2 */
        eval-dd2(...)               /* Evaluate derivative of d1 */
        }
      seq{
        eval-d323(...);             /* Evaluate coriolis and centripetal */
                                    /* forces for joint 3 */
        eval-dd3(...)               /* Evaluate derivative of d1 */
        }
      }

        eval-dd11[3][3]             /* Evaluate derivative of inertial */
                                    /* coupling matrix */
    par{
        eval-dd123(...);            /* Evaluate derivative of coriolis and */
                                    /* centripetal forces on joint 1 dd123[3][3]*/
        eval-dd223(...);            /* Evaluate derivative of coriolis and */
                                    /* centripetal forces on joint 2 d223[3][3]*/
        eval-dd323(...);            /* Evaluate derivative of coriolis and */
                                    /* centripetal forces on joint 3 d323[3][3]*/

      }
   }
}
```

In this parallel realisation the use of channels is not necessary, since no inter-process communication takes between active processes. The functions eval-di23() and eval-ddi() require and alter variables common to both, but because they are executed sequentially variables may be read from memory without the danger of simultaneous read and write.

In a likewise fashion the function high-states() can also be modified to accommodate parallelisation. Figure 21 shows the sequential nature of this function, whilst Figure 23 demonstrates the parallelism to be found in the operations contained in the function. Recall the task of high-states() is to evaluate the derivatives of the three joint accelerations, or the three highest states of the system state vector, of the state space model given by equation 43. The simplified coding of this function, before parallelisation is given below:

```
high-states(x1,x2,x3,x4,x5,x6,x7,x8,x9)
 {
      set-up-mat(...);          /* Setup initial matrices */
      define-d-bar(...);        /* Solve equation 40b */
      define-pbar(...);         /* Solve equation 40c */
      mat-inverse(d-bar);       /* Invert equation 40b */
      mat-by-vec(...);          /* Multiply pbar by dbar */
      get-volts(...);           /* Retrieve joint voltages from file */
      mat-by-vec(...);          /* Multiply dbar by voltage vector   */
      vector-add(...);          /* Add result of two multiplications */
 }
```

A scheme of parallelisation based on the schematic representation of Figure 23 can consequently be imposed on the function high-states(). Note that the function define-pbar() can be decomposed into two almost equally workload-balanced subfunctions, pbar1() and pbar2(), as outlined in Section 5.2. An outline of the parallel C implementation of the function is given:

```
high-states(x1,x2,x3,x4,x5,x6,x7,x8,x9)
 {
seq{
    define-constants(...);        /* Define sine and cosine constants*/
    par-derivative(...);          /* Set up parameter derivatives */

    eval-d(...);                  /* Evaluate d1, d2, d3 */
    eval-d11(...);                /* Evaluate d11[3][3], inertial matrix */

    par{
      seq{
         eval-d123(...);          /* Evaluate coriolis and centripetal */
                                  /* forces for joint 1 */
         eval-dd1(...)            /* Evaluate derivative of d1 */
         }
      seq{
         eval-d223(...);          /* Evaluate coriolis and centripetal */
                                  /* forces for joint 2 */
         eval-dd2(...)            /* Evaluate derivative of d1 */
```

```
            }
        seq{
            eval-d323(...);          /* Evaluate coriolis and centripetal */
                                     /* forces for joint 3 */
            eval-dd3(...)            /* Evaluate derivative of d1 */
            }
        }

            eval-dd11[3][3]          /* Evaluate derivative of inertial */
                                     /* coupling matrix */
    par{
            eval-dd123(...);         /* Evaluate derivative of coriolis and */
                                     /* centripetal forces on joint 1 dd123[3][3]*/
            eval-dd223(...);         /* Evaluate derivative of coriolis and */
                                     /* centripetal forces on joint 2 d223[3][3]*/
            eval-dd323(...);         /* Evaluate derivative of coriolis and */
                                     /* centripetal forces on joint 3 d323[3][3]*/
        }

    par{
        seq{
            define-d-bar(...);       /* Solve equation 40b */
            mat-inverse(d-bar);      /* Invert equation 40b */
            get-volts(...);          /* Retrieve joint voltages from file */
            mat-by-vec(...);         /* Multiply dbar by voltage vector */
            }
        seq{
            define-pbar1(...);       /* Solve equation 45 */
            mat-by-vec(...);         /* Multiply pbar1 by dbar */

        seq{
            define-pbar2(...);       /* Solve equation 46 */
            mat-by-vec(...);         /* Multiply pbar2 by dbar */
            }
        }
            vector-add(...);         /* Add dbar.pbar1 and dbar.pbar2 */
            vector-add(...);         /* Add previous sum to dbar.voltage to */
                                     /* yield 3x1 vector of highest states*/
    }
}
```

It is observed that both pbar1 and pbar2 are multiplied by dbar (the inverse of d-bar). To perform this multiplication the result from mat-inverse(d-bar) must be sent to the respective mat-by-vec() functions. This can be achieved by using channels of correct data type, two dimensional arrays, between the functions mat-inverse(d-bar) and the two mat-by-vec() functions. There is one major drawback however in doing so. Both define-pbar1() and define-pbar2() are almost three times as computationally intensive as mat-inverse(d-bar), and therefore although the two channels in mat-inverse(d-bar) may be ready to transmit data the receive points in both multiplication functions will not be active for a considerably longer time, as both function will be awaiting pbar1() and pbar2() to complete. This would cause a processor, assuming a processor has been allocated, to suspend execution and thus reduce the available processing power in the multiprocessor system. The method by which this is overcome is to change the receive points to the pbar1() and pbar2()

functions at roughly a point in the sequence of calculations when mat-inverse(d-bar) would complete execution and would be ready to transmit. The mat-inverse(d-bar) function would then be able to terminate without waiting or being descheduled. The data received by from the channel could is then transferred to the mat-by-vec() functions through normal variable assignment and storage. The channels are defined as:

```
channel d-to-p1, d-to-p2;
```

and are specified as vector data types, using the assignment keyword, VECTOR , when defining their structure types:

```
struct VECTOR {
      float v[3];
      } db;

inverse-to-p1 = inverse-to-p2 = db;
```

As mentioned in Section 5.2 not only can one exploit the parallelism inherent in the sequence of operations to be performed in the simulator, one can also apply the principle of pipelining to the loop of the simulation computational model. Pipelining provides a efficient method of utilising a multiprocessor system when there is no apparent parallelism in an application. A pipeline is basically several processors operating on the same stream of data in tandem. To apply pipelining in this case one must first consider the nature of the programme, and examine for suitability.

The first obstacle is the fact that the simulator data stream flows in a loop. This means that the first processing stage of the pipeline awaits its input data from the last processing stage. Naturally enough the last stage ultimately depends on the first stage to supply it with data which is processed via the remaining intermediate stages. This deadlock situation is overcome by feeding the initial stage with dummy data until such time as the last stage begins to output process data. At that point the dummy input is removed and the processed data is fed back int the first cycle, thus completing the cycle. The effects of doing this are investigated in Section 5.5. The error introduced by the initial data performs in a transient manner and eventually significantly after a number of sample periods. Refer to Figure 20 which illustrates the sequential flow of tasks in the complete simulator. The coding for the simulator is given in the simplified format:

```
main()
 {
      define-parameters(...);
      null-matrix(...);
```

```
            initialise(...);
            set-up-const-mat(...);

            runge-kutta(1,...);
            high-states(...);
            runge-kutta(1,...);
            high-states(...);
            runge-kutta(1,...);
            high-states(...);
            runge-kutta(1,...);
            high-states(...);
            sim-time-check(t);
     }
```

To create the pipeline it is necessary to define each pipeline stage. The simulator loop decomposes naturally into for stages of equal execution times. Each stage contains a Runge-Kutta numerical integration coefficient evaluation routine and a high-states() function. The software realisation of the pipeline can therefore be written:

```
main()
{
seq{
            define-parameters(...);
            null-matrix(...);
            initialise(...);
            set-up-const-mat(...);
 do{
     par{
        seq{
            if iteration < 4
                  { get-dummy-data(iteration,...) = ch-stage1; }
            else
                  { input-data1 = ch-stage1; }
            runge-kutta(1,...);
            high-states(...);
            ch-stage2 = output-data1;
            }
        seq{
            input-data2 = ch-stage2;
            runge-kutta(1,...);
            high-states(...);
            ch-stage3 = output-data2;
            }
        seq{
            input-data3 = ch-stage3;
            runge-kutta(1,...);
            high-states(...);
            ch-stage4 = output-data3;
            }
        seq{
            input-data4 = ch-stage4;
            runge-kutta(1,...);
            high-states(...);
            ch-stage1 = output-data4;
            }
          }
   }while (sim-time-check(t)==0);
    }
}
```

The function get-dummy-data() supplies the input data to the first processing stage for the first four iterations, or sample periods. On the third iteration the last processing stage begins processing data, and produces its first output at the end of the fourth sample period. It is interesting to note at this point that four processors would normally be required for optimum speedup results on a four stage pipeline but because the maximum paralellism of the high-states() function is three, then for best results overall twelve independent processors are needed. The following Section discusses the hardware aspects of implementing the parallel simulator.

5.4 OPTIMISED HARDWARE STRUCTURE FOR PARALLEL IMPLEMENTATION OF SIMULATOR

5.4.1 The Transputer Architecture

Ideally a parallel language should have as its target architecture a multiprocessor system which contains identical processing units. The combination of various processors, although desirable in very specific instances, reduces programming flexibility and limits scope for application modifications. In most instances the specific tailoring offered by a pot pouri of processing elements is far outweighed by the resulting programming constraints. Generalised parallel programming techniques are centred around the practice of programming for transparent hardware configurations, for the purposes of portability and modification. It should not be necessary for the programmer to be aware of individual processor speeds or processor topologies for general parallel processing applications, and only after the software is designed should an attempt be made to develop optimal processor amalgamations, if necessary.

The parallel C programming language was written specifically for the Transputer[86,...,89] processor. A Transputer is a single VLSI device with memory, processor and communications links for direct connection to other Transputers. Concurrent systems can be constructed from a collection of Transputers which operate concurrently and communicate through links. The Transputer can therefore be used as an integral unit in concurrent systems construction. An important property of VLSI technology is that communication between devices is very much slower than communication on the same device. Memory is of paramount importance in the execution of instructions in computers, and a Transputer therefore possesses its own on board memory on the same integrated circuit.

Since it's development by INMOS in early 1986 the Transputer has become one of the more widely used processors in parallel system realisation. In VLSI technology it is observed that communication between separate devices is much slower than communications within a single device. Thus a processor can spend a lot of time accessing it's memory store. Based on this premise the Transputer was designed with both processor and memory on the same integrated circuit.

Communication between Transputers is executed in serial fashion using point to point connections. A system consists of a number of Transputers connected in some ordered fashion. Each Transputer can be directly connected to a maximum of 4 neighbouring Transputers. A notable feature of the Transputer is that it can support concurrent processes internally, albeit by time sharing the processor between the processes.

The small number of registers, six, is testament to the availability of fast on-board memory. This fact, coupled with the presence of a simple instruction set makes for fast data paths when executing a task. The six registers are as follows;

(i). The workspace pointer which points to an area of store where local variables are kept.
(ii). The instruction pointer which points to the next instruction to be executed.
(iii).The operand register which is used in the formation of instruction operands.
(iv). The A,
(v). B,
(vi). and C registers which form an evaluation stack, and are the sources and destinations for most arithmetic and logical operations.

One of the design decisions of the Transputer is that it should be programmable in a high level language and therefore the small and simple instruction set makes for easy and efficient compilation. The instruction set is independent of processor wordlength so that the microcode is equally applicable to two Transputers of differing wordlength. Each instruction is 1 byte. The first four most significant bits are the function code and the second four are the data. Included in the set are 13 of the most necessary functions for a computer to operate realistically, including ;

load constant	add constant	
load local	store local	load local pointer
load non-local	store non-local	load non-local ptr
jump	conditional jump	call

Two of the function codes are used to allow for the extension of the operand to any length. They are 'prefix' and 'negative prefix'. The 4 least significant bits of the operand register are used to hold the data bits which is then treated as the operand.

In the case where the prefix instruction is used the 4 data bits are loaded as normal and then shifted up 4 places. The negative prefix is the same except the data is complemented before shifting. Thus operands of any length up to the size of the operand register can be represented. Even with this facility research shows that approximately 80% of executed functions are encoded in a single byte. Therefore in only 20% of cases is a prefix used. This augurs well for encoding efficiency. This means also that several instructions will be got during a memory fetch cycle.

Transputers not only operate in parallel but also support the parallel C model of concurrency internally. There is a microcoded scheduler present which enables any number of processes to be executed concurrently by sharing the processor time. The actual processor itself does not have to dynamically allocate the storage space since this is handled by the parallel C compiler. A feature of the scheduler is that it prevents inactive processes from consuming processor time.

Parallel C communications are point to point, synchronised and unbuffered. Between Transputers the channel is implemented by a point to point serial link and internally the channel between two processes is realised by a single word in memory.

The clock of the Transputer operates with a period of 1 microsecond and its current value can be accessed via a 'Read Timer' instruction. A process may arrange its input against the timer so that it begins to execute at a fixed point in time. This is done using a 'Timer Input' instruction. This is basically a descheduling of the process priority. On arrival of the specified time the process is then re-scheduled.

Several schemes have been proposed for the multiprocessor control of robot manipulators [90,91,92], but very few schemes implement a real time model of the forward dynamics of such manipulators, for the purpose of controller evaluation. Nonetheless, it is interesting to note the various methods used to balance the workload among the processing units of these parallel controllers, as this problem is not unlike the scheduling problem for the simulator software. Jones and Fleming [93] propose a technique whereby the calculations involved in inverse dynamical equation evaluation are distributed among a processor farm, thus ensuring a high rate of efficiency. Unfortunately, the overheads of an intensive scheduling algorithm, which is necessary to ensure continuous processor activity, prove to be as computationally demanding as the calculations themselves, as expressed by Jones;

"The structure chosen is a processor 'farm' .... the initial measurements showed that the scheduling process itself took longer than the computation of tasks ... thus destroying any speed-up due to parallelism."

Although this problem was surpassed somewhat by the introduction of 'pre-determined schedules' the inter-Transputer communications overhead proved also to be substantial. This was not anticipated in simulation tests by Jones and Fleming. Since communications are constant due to the fixed nature of the software and since also the controller needs to work within a limited and fast sample time, the percentage overhead for Transputer communications becomes prominent, and by adding more processors can be increased even further. This problem would be reduced in an application where, according to Jones,

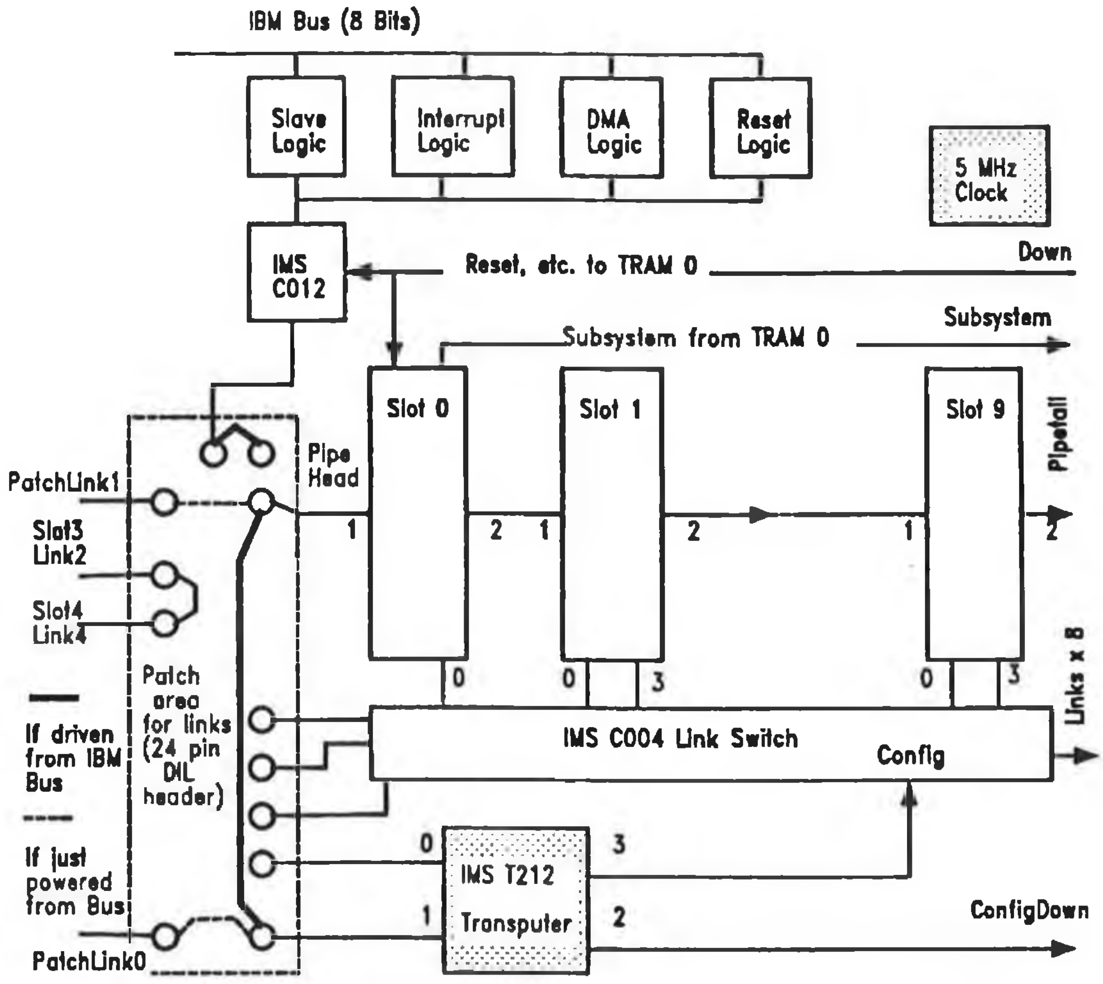
" ... given a problem with ten times as much computation - and ten times as much time to accomplish it - the overheads would not be as serious."

Unfortunately, the aspirations of Jones due not apply to the parallel implementation of the PUMA 560 robot simulator. It is proposed, based on these observations that a Transputer configuration be used, other than a processor farm, whereby complete processes be mapped onto individual Transputers. This removes the need for dynamic scheduling and simplifies the performance analysis, and more importantly improves actual speedup. The generalised software routines of Section 5.3 illustrated the fact that at only one instance during one cycle of the simulator (one sample period) were channels utilised. This has major benefits in the overall run-time of the simulator and proves to be a quite desirable property in the software design. Considering that the maximum parallelism (the maximum number of processes occurring in parallel) is three, equal to the number of simulated links, the optimum number of Transputers is also three. If it is desired to implement a four stage pipeline structure as described in Section 5.3 then one must provide three processors for each of these stages, twelve Transputers in all, to achieve optimum performance.

5.4.2 The IMS B008 Motherboard

From a practical viewpoint a Transputer system may be housed in a conventional IBM PC (XT or AT) by means of the IMS B008 motherboard [94], which is plugged into the computer in the same manner as any normal extension board. It has slots for up to ten TRAMs (TRAMs are board-level Transputers that integrate processor, memory and peripheral functions, and which communicate with the outside world by means of INMOS serial links, arranged in a standard DIL pin-out). Links 1 and 2 from each of the TRAMs are hard wired on the IMS B008, such that the TRAMs, when plugged in, form a pipeline of processing elements. The remaining links can be 'softwired' using an INMOS IMS C004 programmable link switch, incorporated on the IMS B008. This arrangement allows a large variety of networks to be created under direct software control. Figure 24 illustrates the main blocks of the IMS B008 motherboard.

The IMS C004 device is controlled by an IMS T212 16-bit Transputer. Configuration data for the IMS C004 is fed into link 1 of the IMS T212, which then passes it on to the IMS C004 on link 3. The same data is also passed out of the IMS T212 link 2 to the 37-way D-connector on the edge of the board. In this way IMS B008 boards can be cascaded with the IMS T212's forming a chain. Configuration data passes down this chain, with each IMS T212 sending the appropriate data to the IMS C004 to which it is connected.



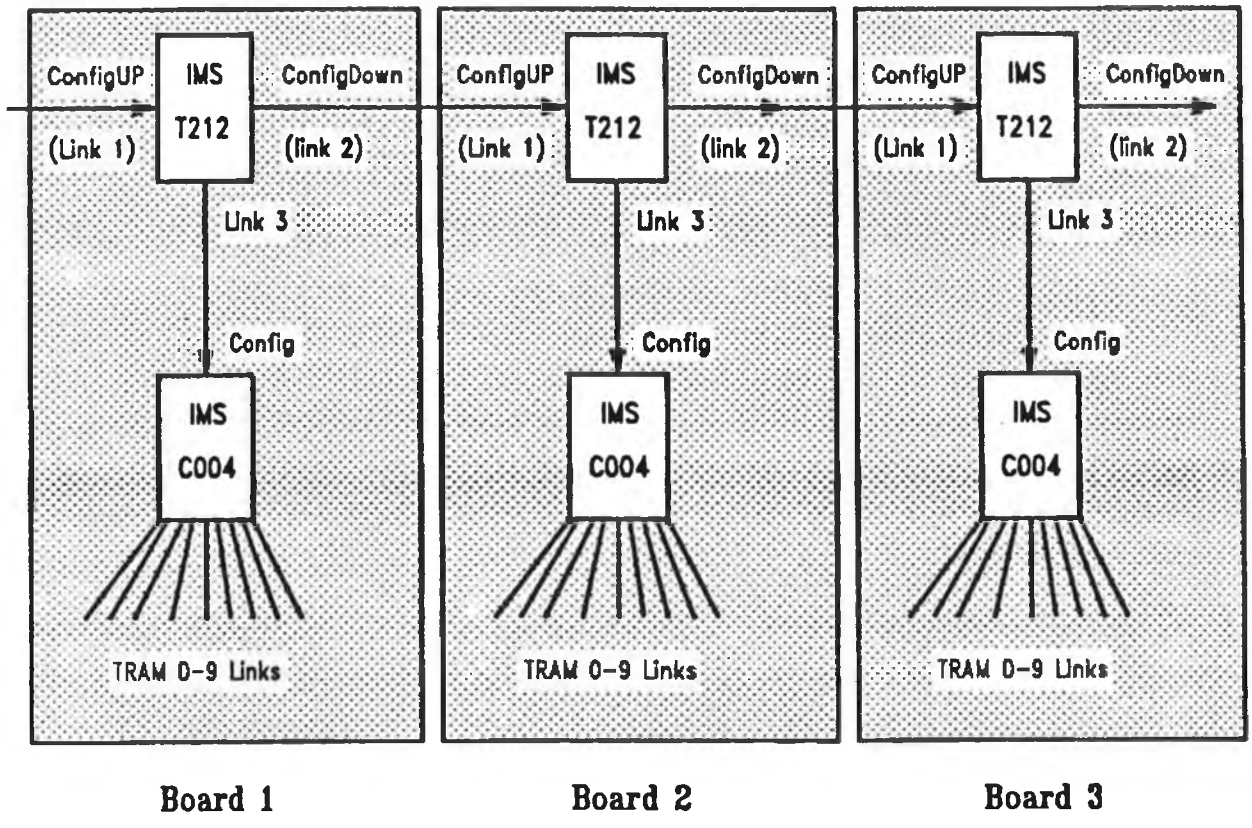
Simplified Block diagram representation of IMS B008 Motherboard

FIGURE 24

An interface to the IBM bus is provided so that the programme running on the

IBM PC can control the TRAMs on the IMS B008 and pass data to or from them. Data communication can take place by means of a software routine which uses polling, or via a DMA mechanism which gives a higher data rate. Different events on the IMS B008, selectable by the programmer, can generate an interrupt on the IBM PC to continuously poll status registers on the IMS B008, so that the PC can carry on with other tasks while programmes are running on the IMS B008.

As mentioned the IMS B008 has sufficient slots to accommodate ten TRAMs. The board is hardwired such that TRAM(N), link 2 is connected to TRAM(N+1), link 1, producing a 10 TRAM pipeline configuration. However TRAM3, link 2 and TRAM4, link 1 are taken to the patch area, (refer to [94], appendix F, for a detailed description), so that this pipeline may be broken if the user so wishes. The link entering the first TRAM of the pipeline (TRAM0, link 1) is termed the *Pipehead*, and the link leaving the last TRAM (TRAM9, link 2) is called the *Pipetail.*



Schematic Representation of IMS T212 Configuration Pipeline

FIGURE 25

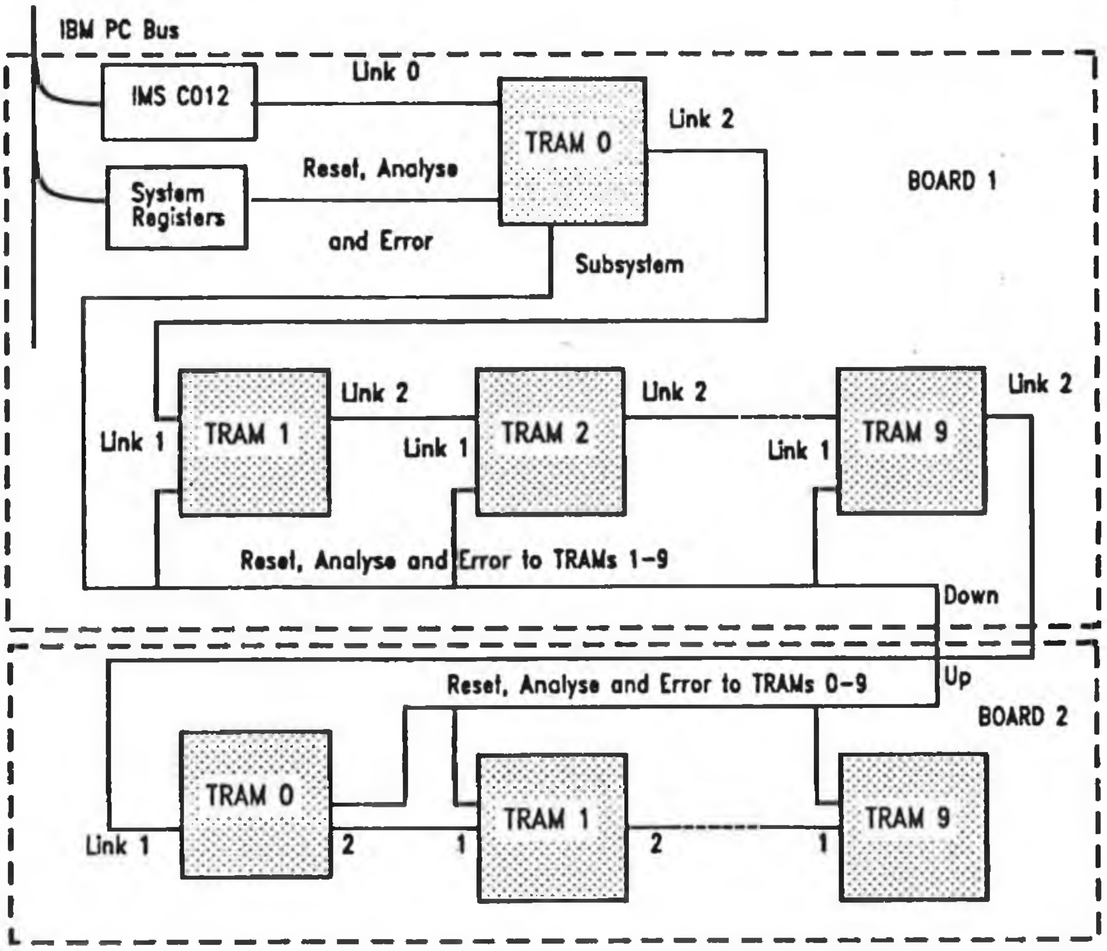
Pipehead is connected to the patch area so that input to the pipeline may be sourced from the either the IBM PC or another IMS B008 board and the pipetail is connected to the 37-way D-connector, facilitating connection to external boards. This allows for the construction of very complex topologies consisting of multiple IMS B008 motherboards, although this will be seen not to be necessary in this particular application. Since link 1 and 2 of each TRAM are used in the formation of the pipeline both link 0 and 3 are left for connection to other TRAMs on the same or neighbouring motherboards for the construction of TRAM systems. The design of the IMS B008 is such that TRAM0, link 3 and TRAM(1..9), links 0 and 3 are taken directly to the IMS C004, a 32-way link switch. TRAM0, link0 is not connected directly to the IMS C004 switch since it is connected to the patch area to facilitate connection to the IBM bus. It is also possible to connect TRAM0, link 0 to the switch via the patch area, if needed.

The IMS C004 has 32 link outputs and 32 link inputs and a configuration link. The 32 links may be optionally connected to one another by the sending of the correct configuration signals along the C004 configuration link. As will be seen this allows for link 0 or 3 of any TRAM to be connected to link 0 or 3 of another TRAM. In addition eight links are taken from the IMS C004 switch to the 37-way connector, and permits inter-connection between TRAMs on different IMS B008 motherboards. Control of the IMS C004 switch is executed on a T212 16-bit Transputer. Innovatively links 1 and 2 of the T212 are taken to the D-connector so that they too may be pipelined in the same manner as the TRAMs. Configuration data may be sent in on link 1, sent out on link 2 and so on along the chain of T212's. Each T212 extract the data applicable to its subject IMS C004 programmable switch. Figure 25 shows how this mechanism is realised.

5.4.3 A Tailored Transputer Topology

In the instance of the pipeline realisation of the parallel simulator, as discussed in Section 5.4.2, it was found that twelve processing units were needed to achieve optimum speed performance. The layout of these Transputers would involve three Transputers processing in parallel at each of the four stages of the pipeline. Each IMS B008 board is capable of supporting ten TRAMs, thus one board is insufficient. Two boards are necessary, and two of the pipeline stages, or six TRAMs, can be located on both. Because of the particular physical layout of the IMS B008 TRAM slots (cooling requirements) and the physical size of TRAM cards slots 0,1 & 2 on IMS B008 board 1 were chosen as processing stage 1 of the pipeline, slots 4,5 & 9 on IMS B008 board 1 were chosen as processing stage 2 and the same slot numbers were chosen on IMS B008 board 2 as stages 3 and 4 respectively. It is necessary to devise a connection scheme for these TRAMs, so that both localised parallelism is maintained and also the overall data flow of the pipeline is consistent. It is imperative that a system of control hierarchy is established. It is proposed that the application on the TRAM network run under the control of TRAM0, board 1. This facilitates programme development and debugging at a localised level and prevents the resetting of this TRAM every time the application is reset. The control TRAM is defined as the source of the **notReset** and **notAnalyse** control signals and can be established by the setting of appropriate jumpers on the IMS B008 motherboard. By connection of subsystem ports of the TRAMs to the Subsystem line, shown in Figure 24, the *reset, analyse and error* functions of the TRAMs are put under the control of TRAM0. This includes the IMS T212 Transputer which controls the IMS C004 switch.

Furthermore, the configUp link of the IMS T212 (link 1) must be connected to the pipehead, link 1 of TRAM0. The allows the TRAM running the Module Motherboard Software (MMS)[95] to feed the configuration data directly to the IMS T212, which relays it onto the IMS C004 switch. This connection can be made via the patch area. The IMS C012 link is also connected to TRAM0 (link 0), so that TRAM0 always boots down this link, again via the patch area.



Schematic represemtation of a two IMS B008 board TRAM system.

FIGURE 26

The TRAM slots that are vacant can be by-passed by connection of location points for links 1 and 2 by special 8-pin *pipejumpers*. Figure 26 demonstrates the hierarchical layout of the twelve TRAMs, where TRAM0 on board 1 is the controller. All other TRAMs are connected to the Reset, Analyse and Error line. Interface between the two boards is by connection of TRAM9, board 1, link2 to TRAM0, board 2, link1. The separation of the TRAMs on two different boards is transparent to the application. In order for TRAM0 to control all the TRAMs on the second board it is necessary to connect the SUBSYSTEM port, on board 1, to the UP port on board 2. Continuation of the pipeline is achieved by connecting Pipetail, board 1 to Pipehead, board 2. This means that the patch area on board 2 must be altered so that the ConfigUpLink on its IMS T212 is brought to the 37 way D-connector via PatchLink0. Therefore board 1, ConfigDownLink should be connected to board 2, PatchLink0. These and several other control connections, which are made across the two D-connectors, are given in Table 8.

BOARD 1	PIN NO.	BOARD 2	PIN NO.
notSubsystemReset	33	notUpReset	20
notSubsystemAnalyse	15	notUpAnalyse	2
notSubsystemError	34	notSubSystemError	21
PipetailLinkOut	16	PatchLink1	14
PipetailLinkIn	35	PatchLinkOut1	32
ConfigDownLinkOut	17	PatchLink0	13
ConfigDownLinkIn	36	PatchLinkOut0	31

Table showing the necessary connections between the two D-connectors of the two IMS B008 motherboards.

TABLE 8

To interface with the IBM PC bus the IMS B008 must implement some sort of protocol. The simplest method is polling. This technique is explained fully in reference [94].

For higher data rates, a DMA interface has been incorporated into the IMS B008. For a complete understanding of this mechanism it is imperative that the DMA controller chip in the PC is understood[96]. Once the DMA controller has been initialised, the DMA interface may be used in several ways. A value of "0" is written to the DMA control register on the IMS B008 board when data is to be transferred from the PC to the board. A value of "1" signals data transfer in the opposite direction. The IMS B008 makes a DMA request for a single byte at a time. In this manner a byte is transferred in between each instruction execution on the PC's processor. In general a DMA transfer has twice the data transfer rate as an optimised assembly routine for polling. Termination of a DMA transfer is denoted in two ways: the DMA controller can be polled by reading the status register, or an interrupt can be sent to signal transmission end. To avoid the IBM PC CPU having to continuously poll the IMS B008 DMA controller's status register interrupts may be used. The IMS B008 is capable of interrupting the PC on the occurrence any one of the four events:

- A DMA transfer has completed
- An error has occurred on the IMS B008
- The IMS B008 is ready to receive a byte of data via the IMS C012
- The IMS B008 is ready to send a byte of data via the IMS C012

The 4 types of interrupt are individually enabled/disabled by the interrupt control register, located at *boardbase+$13*. It is also important to note that the link speeds are all set to 20 Mbits/sec by setting the speed switch on the IMS B008 board to 000. There are 7 other switch combinations which allow the user to select combinations of speeds of the TRAMs and the IMS T212 Transputer.

In order to connect the TRAMs in a pre-determined configuration it is necessary to write a *softwire* file specifying the proposed connection scheme to be implemented by programmable links, and also a *hardwire* file listing the jumper settings, or hardwired connections between TRAMs. These files are read by the Module Motherboard Software (MMS) and by examining the hardwired file the softwire file is tested for illegal combinations. The softwire file contains a list of user specified connections between TRAMs, whilst the hardwire file contains a list of connections already made on the board by the physical placing of jumpers. The MMS needs to examine the hardwire file first, before programming the links in the softwire file, since certain combinations of TRAM link connections would be illegal. Without deliberating on the details of the formulation of softwire and hardwire files, (this can referenced in the MMS user guide [96]), the softwire file for this particular application would be as follows:

```
SOFTWIRE
    PIPE 0
         SLOT 0, LINK 3 TO SLOT 2, LINK 3
         SLOT 4, LINK 3 TO SLOT 9, LINK 3
         SLOT 9, LINK 2 TO EDGE 3
    PIPE 1
         SLOT 0, LINK 1 TO EDGE 6
         SLOT 0, LINK 3 TO SLOT 2, LINK 3
```

```
          SLOT 4, LINK 3 TO SLOT 9, LINK 3
END
```

And the hardwire file, which specifies the definition of board types and pipeline layout, reads as follows:

```
DEF boarda
    SIZES
      T2 1
      C4 1
      SLOT 10
      EDGE 10
END

T2CHAIN
    T2 0, LINK C4 0
END

HARDWIRE
    SLOT 0,LINK 2 TO SLOT 1,LINK 1
    SLOT 1,LINK 2 TO SLOT 2,LINK 1
    SLOT 2,LINK 2 TO SLOT 3,LINK 1
    SLOT 3,LINK 2 TO SLOT 4,LINK 1
    SLOT 4,LINK 2 TO SLOT 5,LINK 1
    SLOT 5,LINK 2 TO SLOT 6,LINK 1
    SLOT 6,LINK 2 TO SLOT 7,LINK 1
    SLOT 7,LINK 2 TO SLOT 8,LINK 1
    SLOT 8,LINK 2 TO SLOT 9,LINK 1

    C4 0,LINK 10 TO SLOT 0,LINK 3

    C4 0,LINK 1 TO SLOT 1,LINK 0
    C4 0,LINK 11 TO SLOT 1,LINK 3

    C4 0,LINK 2 TO SLOT 2,LINK 0
    C4 0,LINK 12 TO SLOT 2,LINK 3

    C4 0,LINK 3 TO SLOT 3,LINK 0
    C4 0,LINK 13 TO SLOT 3,LINK 3

    C4 0,LINK 4 TO SLOT 4,LINK 0
    C4 0,LINK 14 TO SLOT 4,LINK 3

    C4 0,LINK 5 TO SLOT 5,LINK 0
    C4 0,LINK 15 TO SLOT 5,LINK 3

    C4 0,LINK 6 TO SLOT 6,LINK 0
    C4 0,LINK 16 TO SLOT 6,LINK 3

    C4 0,LINK 7 TO SLOT 7,LINK 0
    C4 0,LINK 17 TO SLOT 7,LINK 0

    C4 0,LINK 8 TO SLOT 8,LINK 0
    C4 0,LINK 18 TO SLOT 8,LINK 3

    C4 0,LINK 9 TO SLOT 9,LINK 0
    C4 0,LINK 19 TO SLOT 9,LINK 3
    C4 0,LINK 20 TO EDGE 0
    C4 0,LINK 21 TO EDGE 1
    C4 0,LINK 22 TO EDGE 2
    C4 0,LINK 23 TO EDGE 3
    C4 0,LINK 24 TO EDGE 4
    C4 0,LINK 25 TO EDGE 5
    C4 0,LINK 26 TO EDGE 6
    C4 0,LINK 27 TO EDGE 7
    C4 0,LINK 28 TO EDGE 8
    C4 0,LINK 29 TO EDGE 9

DEF boardb
    SIZES
      T2 1
      C4 1
      SLOT 10
      EDGE 10
END

T2CHAIN
    T2 0, LINK C4 0
END

HARDWIRE
    SLOT 0,LINK 2 TO SLOT 1,LINK 1
    SLOT 1,LINK 2 TO SLOT 2,LINK 1
    SLOT 2,LINK 2 TO SLOT 3,LINK 1
    SLOT 3,LINK 2 TO SLOT 4,LINK 1
    SLOT 4,LINK 2 TO SLOT 5,LINK 1
    SLOT 5,LINK 2 TO SLOT 6,LINK 1
    SLOT 6,LINK 2 TO SLOT 7,LINK 1
    SLOT 7,LINK 2 TO SLOT 8,LINK 1
    SLOT 8,LINK 2 TO SLOT 9,LINK 1

    C4 0,LINK 10 TO SLOT 0,LINK 3

    C4 0,LINK 1 TO SLOT 1,LINK 0
    C4 0,LINK 11 TO SLOT 1,LINK 3

    C4 0,LINK 2 TO SLOT 2,LINK 0
    C4 0,LINK 12 TO SLOT 2,LINK 3

    C4 0,LINK 3 TO SLOT 3,LINK 0
    C4 0,LINK 13 TO SLOT 3,LINK 3

    C4 0,LINK 4 TO SLOT 4,LINK 0
    C4 0,LINK 14 TO SLOT 4,LINK 3

    C4 0,LINK 5 TO SLOT 5,LINK 0
    C4 0,LINK 15 TO SLOT 5,LINK 3

    C4 0,LINK 6 TO SLOT 6,LINK 0
    C4 0,LINK 16 TO SLOT 6,LINK 3

    C4 0,LINK 7 TO SLOT 7,LINK 0
    C4 0,LINK 17 TO SLOT 7,LINK 0

    C4 0,LINK 8 TO SLOT 8,LINK 0
    C4 0,LINK 18 TO SLOT 8,LINK 3
    C4 0,LINK 9 TO SLOT 9,LINK 0
    C4 0,LINK 19 TO SLOT 9,LINK 3

    C4 0,LINK 20 TO EDGE 0
    C4 0,LINK 21 TO EDGE 1
    C4 0,LINK 22 TO EDGE 2
    C4 0,LINK 23 TO EDGE 3
    C4 0,LINK 24 TO EDGE 4
    C4 0,LINK 25 TO EDGE 5
    C4 0,LINK 26 TO EDGE 6
    C4 0,LINK 27 TO EDGE 7
    C4 0,LINK 28 TO EDGE 8
    C4 0,LINK 29 TO EDGE 9

PIPE boarda, boardb END

END
```

The language which describes the connections is known as HL1. Two boards are defined in the above listing, as is can be seen from the description DEF boardx. The description of both boards is the same. The hardwire file initially specifies, for each board, the quantity of on-board IMS T212 Transputers (T2) as one and the quantity of IMS C004 programmable switches (C4) also as one. The number of slots and edges are both defined as ten. The T2CHAIN command describes how the IMS T212 Transputer is connected to the IMS C004 switch. The command in the above hardfile designates that Transputer 0 is connected to C004 0 via link 3. The pursuing list of extensive hardwire specifications are used to inform the MMS of the actual connections between TRAMs.

When the MMS is instigated it is supplied with the two configurations files. It then proceeds to programme the links specified in the softwire file, all the while monitoring each combination of link connections to avoid an illegal connection of a link which has already been hardwired as per the hardwire file. It achieves this link programming by the passing configuration information on to each IMS T212, which decides whether the data is applicable to the C004('s) under its control. The IMS T212's are organised in a pipeline structure (see Figure 33) so that the data is passed wholly along through all IMS T212's. The MMS, which is run from the DOS prompt on the PC, offers the user a suite of menu functions and makes debugging and diagnostic testing of the source files possible without the hardware being on-line. In conclusion this Section details the Transputer hardware scheme necessary for optimum speed performance of the parallel PUMA 560 robot manipulator simulator. The following Section attempts to formulate this performance mathematically. It also graphically investigates the relationship between processor availability, by examining performance degradation due to reduced processing power. From the pending results some conclusions are drawn.

5.5 EVALUATION OF SIMULATOR PERFORMANCE IN PARALLEL FORM

It is imperative that before actually evaluating model performance indices, such as speedup, processor efficiency etc., one must first validate the model accuracy and vindicate its representation of the physical PUMA 560 manipulator. The following Section, Section 5.5.1, attempts to do this by examining open loop tests on the each of the links. The results are compared against expected manipulator responses, and against certain characteristics inherent in the manipulator. Assuming a fully validated model Section 5.5.2 proceeds to perform an analysis on the proficiency of the parallelised simulator.

5.5.1 Evaluation of PUMA 560 Manipulator Model

The most appropriate method of testing the model is to graphically examine the simulator output over a given time domain in response to particular inputs. All graphical data is displayed in Appendix C, and any graphical data quoted in this Section is contained in the same. The tests carried out on the robot simulator are listed and explained in sequence below:

TEST 1: The first test is to monitor the behaviour of each joint whilst their respective 'hold' voltages are applied. The joint hold voltages are the voltages required to hold each joint in a fixed position of 0 radians. Graphs 5.1,5.2 and 5.3 show the joint positions, velocities and accelerations respectively. Both joint 1 and 3 behave in a very constrained manner and there is a small deviation in the position of joint 2 from 0 radians, albeit of the order of 10^{-5} radians. Both velocity and acceleration can be predicted from the joint position trajectory. The acceleration of joint 2 is the only significant joint acceleration, and is a maximum at $11x10^{-6}$, from which it decreases in a underdamped fashion to almost zero over a three second period.

The following seven tests are designed to examine the coupling effects between the three joints due to joint positioning, movement and acceleration.

TEST 2: The second test, the first of these coupling tests, graphs 5.4, 5.5 & 5.6, involves moving joint 1 whilst both joint 2 and 3 are held still. This is to examine the coupling effects on joints 2 & 3 due to movement of joint 1. The response of joint 1 is denotative of an integrative system, or a motor with constant inertial load, which is as desired. The velocity and acceleration, joint 1, are derived directly from the ramp and step nature of the position and as can be seen they are both almost zero for the other two joints. The movement in the other two joints was found to be of the order of 0.05° and because of the nature of the first joint this is to be expected.

TEST 3: Both joint 1 and joint 2 are moved whilst joint 3 is held steady for the third test. The third joint stays at the zero radian position whilst the other two joints are moved, see graph 5.7, and as a result its velocity and acceleration are both zero (graphs 5.8,5.9). The coupling effect of joint 2 on joint 1 seems to be more significant in this case compared to the instance where joint 2 is stationary. This is explained by the changing coupling centripetal and inertial torques as joint 2 changes position, whilst joint is moving.

TEST 4: In this test, as opposed to test 3, joint 2 is held while joints 1 & 3 are moved, by a step input. For most of the duration of the test joint 2 remains stationary, except near to simulation termination it deviates from the zero position and this is reflected in the joint velocity and the erratic acceleration at about 12 seconds. (Graphs 5.10, 5.11 & 5.12 give position, velocity & acceleration.) The cause of this is joint 3 reaching its joint limit and suddenly stopping. This is consistent with the actual jarring that would occur in the actual robot performance. This is a reflection of the strong coupling between joints 2 & 3, which is present in the robot.

TEST 5: In this particular test all joints are moved, by applying a step voltage to each joint to examine for the coupling effects between all three joints. First joint 1 reaches it joint limit, then ditto joint 3 and finally joint 2(negative step). Compare this to the case where joint 3 is held, and joint 2 reached its limit much quicker (graph 5.7). This can be attributed to the decreased speed due to the increased coupling effects between joints 2 & 3 when the robot arm is moving, which is consistent with the actual PUMA 560.

TEST 6: Tests 6 and 7 involve moving joint 2 whilst holding joint 1 or both joints 1 and 3 steady. Test 6 specifies that both joint 1 and 3 are held and joint 2 moved to examine the coupling effects due to joint 2. Graph 5.16 shows the positions of the joints. Other than a small rippling deviation on joint 3 the effect of joint 2 on the other two joints is minimal (graph 5.16), and there is a smaller coupling effect on joint 1 than on joint 3. Again this is consistent with the real manipulator system.

TEST 7: This the penultimate test involves moving joints 2 & 3 and holding joint 1. When both joint 2 & 3 reach their limits joint 1 deviates from the 0 radian stationary position. This is consistent with the actual event of both manipulator joints reaching their limits simultaneously.

TEST 8: For this test joints 1 & 2 are held whilst joint 3 is moved to its joint limit to test the coupling effects of joint 3. The abrupt termination of joint 3's trajectory at the joint limit causes a resonance in joints 1 and 2, as can be seen in graph 5.22. This is also reflected in the erratic velocity and acceleration performance, shown in graphs 5.23 and 5.24.

In most cases, especially those which are most significant, the behaviour of joint trajectories can be explained by reference to the actual robot. As with the robot the coupling effects varied between the different joints, and also depended on joint movement. Joint 1 has the least coupling effect on the other joints. This is explained by the physical construction of the first joint. It revolves in a horizontal plane, whilst the other two joints operate in a vertical plane. The strong coupling between joint 2 and 3 is explained by the fact that the location of joint 3 has a large effect on the degree of loading experienced by joint 2, and once again this is present in the simulator of the PUMA 560 robot manipulator.

5.5.2 Analysis of Speedup and Processor Efficiency

In Chapter 3 the performance indices of a parallel system were extensively investigated and formulated. Speedup for a parallel programme was defined as the sequential execution time for that programme divided by the execution time of the parallel version. In this Section speedup is evaluated on the assumption of unlimited availability of processing units. In Section 5.5.3 the eventuality of constraints on the numberof processors available is considered and its effect on speedup, and consequently processor efficiency. In order to evaluate the speedup of the parallel simulator the execution times of the sub-processes must first be measured. Table 9 shows the execution times of the required C function routines. Both the run times on an Intel 80286µP and INMOS T414-20 Transputer are given. This is to facilitate the performance comparision for two different multiprocessor systems. A T414-20 Transputer is quoted at 11.5µs per FLOP and the INTEL 80286 microprocessor performs at 54µs per FLOP.

The reader's attention is drawn to figures 22 and 23, in Section 5.2, which illustrate the parallelism in the software programme functions set-up-mat() and high-states() respectively. Based on the timing measurements given in table 9 speedup can be calculated using these flowcharts as guides to the processes executed in parallel. Using both the information in Table 9 and figures 22 and 23 a graph of the scheduling scheme for the processes can be extracted. The first task is to evaluate the speed increase for the parallelised version of set-up-mat(). Figure 27 demonstrates the schedule of execution for this function, although it must be noted this schematic representation is not drawn to scale, and is intended for illustration (note, however, that the longer the process box the longer the execution time).

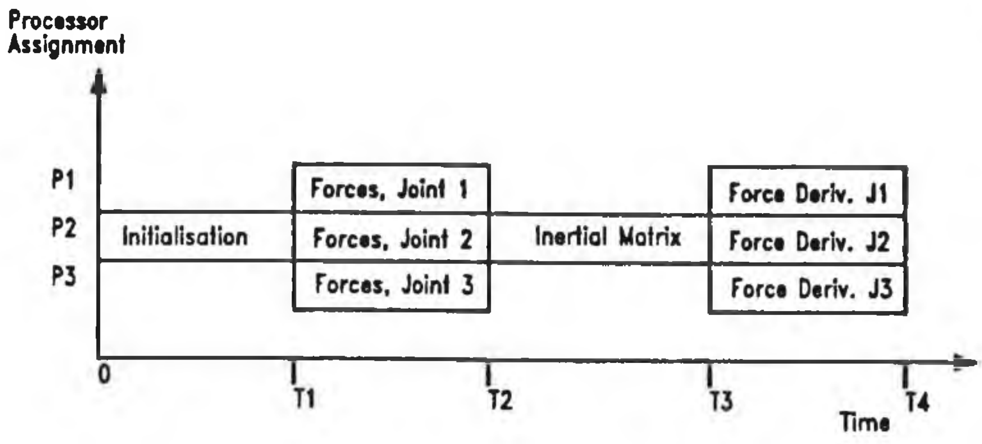
	Execution Time [mS]	
Function Name	INTEL 80286 μP	INMOS T414-20
high-states()	42.228	8.993
null-matrix()	00.275	0.0058
vector-add	00.172	0.0368
vector-sub()	00.174	0.037
mat-sub()	00.552	0.1175
mat-add()	00.552	0.1175
mat-mult()	01.300	0.2768
vec-by-mat()	00.512	0.109
mat-by-vec()	00.512	0.109
mat-inverse	05.958	1.2684
get-volts()	00.094	0.002
initialise()	129.741	27.630
set-up-const-mat()	01.578	0.3356
setup-mat()	12.275	2.6141
define-d-bar()	01.856	0.3952
define-p-bar()	21.058	4.4857

Execution times of software functions used in simulation
TABLE 9

Execution of the function, set-up-mat(), begins at time 0, and completes at time T4. The first task to be scheduled is the initialisation of variables and assignment of parameters, which takes 0.262ms. At time T1 when this completes the three joint forces evaluation routines are scheduled, and the longest of these, for joint 1, takes 106.9μs. At time T2 the inertial matrix is evaluated on its own in sequential fashion, taking 188.6μs. The final group of tasks is to calculate the derivatives of the joint forces, (centripetal and coriolis forces), and these may be calculated in parallel for each joint. The slowest of these processes is again for joint 1, and has an execution time of 229.0μs, as compared to 37.4μs and 26.2μs for joints 2 and 3 respectively.

The slowest path in this realisation is therefore 787.3μs. This value may be ported into the parallel version of high-states(), which may now be evaluated.
The scheduling scheme corresponding to the parallel implementation of high-states() is

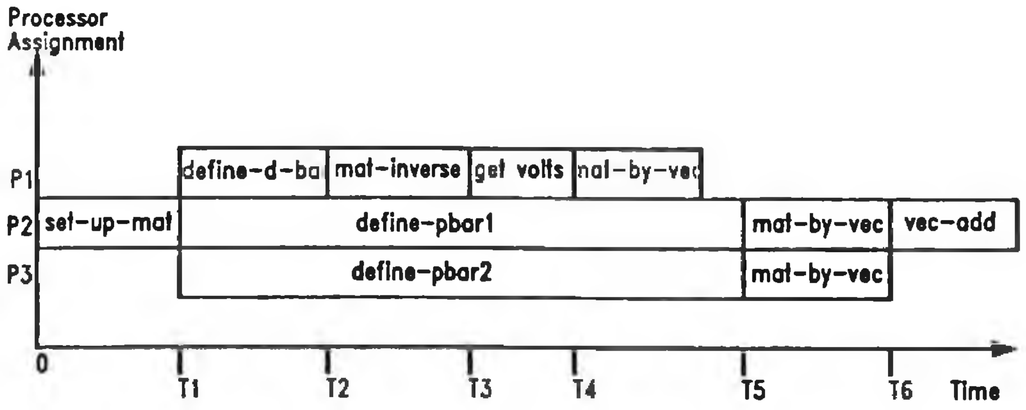
shown in Figure 28. Initially the sut-up-mat() function is executed in parallel form, as described above. For the purpose of schedule illustration it is represented by a single process on a single processor, it would of course require three processors to exploit its maximum parallelism.



Schedule of processes for function set-up-mat()

FIGURE 27

The computational path, on completion of set-up-mat(), forks three ways: define-d-bar(), define-pbar1() and define-pbar2(). The define-d-bar() routine is followed by the functions mat-inverse(), get-volts() and mat-by-vec() whose aggregate execution time along with that of define-d-bar() is less than either define-pbar1() or define-pbar2().



Scheduling graph of sub-functions in the function high-states()

FIGURE 28

Of the two latter functions define-pbar2() takes the longest time, taking 2.272ms. The two other computational paths are placed in a wait state on completion of their tasks, awaiting the completion of the longest path. This degrades the processor efficiency rate. If no wait states occurred, assuming unlimited availability of processors, and the "n" processors were continually busy then the speedup would be equivalent to the maximum parallelism (the maximum number of processes that can run in parallel for that application). This is the optimal and ideal situation, with no possibility of realisation in an actual parallel system. The factors which prevent this are predominately inter-processor communications and unbalanced processor workloads.

The next two processes to be scheduled are the mat-by-vec() multiplication routines, both with execution times of 109.0μs. These are followed by two sequential vector addition routines (36.6μs each).

The computationally longest path is found to be 3.242 milliseconds. In order to evaluate the theoretical speedup, recall that the execution time for the sequential version of high-states() is 8.993ms. Using the computational model of Figure 20, it is observed that the run-time for one simulation loop(i.e. four calls to high-states() and the evaluation of the Runge Kutta coefficients), in sequential form, is 37.194ms:

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}} = \frac{37.194}{15.254} \cong 2.438$$

Therefore, since no more than three processors are required at any processing stage, the efficiency for a 3-processor system is given by:

$$\text{Efficiency} = \frac{\text{Speedup}}{n} \cong \frac{2.438}{3} = 0.8127$$

These values, which represent absolute speedup and efficiency and do not allow for overheads normally present in a multiprocessor system, are for the non-pipelined case. Introduction of the pipeline technique offers the prospect of a much improved speedup margin and greater processor efficiency, at the expense of a larger multiprocessor system. The pipelining principle, as applied to the simulator, involves the division of the simulator loop in processing stages which can be executed concurrently. Sections 5.2 and 5.3 describe this method in detail and it was found that the simulation loop naturally decomposed into four pipeline stages, each containing a Runge Kutta coefficient evaluation routine and a call to the function high-states(). The combination of the pipeline and the exploited parallelism inherent to each pipeline stage yields a speedup four times greater than without pipelining:

$\text{Speedup} = 4 \times 2.438 = 9.752$

and a processor utilisation efficiency given by:

$\text{Efficieny} = \text{Speedup} \div 12 = 9.752 \div 12 = 0.8126$

It is assumed, once more, that there is unlimited availability of processors, and for the pipeline the optimal number is twelve, hence the figure for efficiency. This is derived from the fact that the optimal number for each stage is three (maximum parallelism is three), and there are four stages in all. Section 5.5.3 considers the effects on speedup and efficiency of both limiting the number of processors and adding additional processors to the optimised multiprocessor system.

One further concern in the usage of a pipeline architecture is the effect on the simulator performance. In this particular application the output of the pipeline is fed back into the input of the pipeline. In normal sequential operation the first processing stage is fed initial start up data, (joint angular position), defined by the user. This information is processed and passed onto the second stage, which processes it further, and likewise to stage three and four. When stage four has completed, the relevant data is returned to stage one, thus rebeginning the simulation software loop. By pipelining the software because all stages start at the same time there is no available output from stage one to serve as input to stage two. The second stage must be fed temporary data (i.e. initial conditions) until the first stage has completed processing. It takes one complete sample period before accurate data from processing stage one is available to the second processing stage. Likewise stage three must wait two sample periods before it can be supplied with correctly processed data from both stage one and two, and for stage four it takes three sample periods. During this initial period errors in PUMA 560 manipulator simulation performance are most likely to occur.

In order to quantify this error the pipelined simulator output is compared to the output from the unpipelined simulator. The errors on all three joint positions were found to be of similar magnitude. Graph 5.29 shows the average percentage error between the pipelined response and the unpipelined response to step inputs on the manipulator joints. As expected the initial error is quite large, due to the inaccuracy of the data being fed into the various processing stages. This average percentage error decays dramatically over the next few iterations. On about the tenth iteration the error has almost been removed from the output joint positions of the simulator, and is found to be less than 0.2 of a percent. Considering that the simulator is only based on a manipulator mathematical model errors of this magnitude are quite insignificant. As the number of loop traversals progresses the error becomes less significant due to the fact that over time the information being fed into each stage becomes more accurate, and thus the output data from each stage also increases in accuracy. In a typical simulation run over several seconds the simulation loop is traversed several hundred times. Despite the initial start-up error the pipelined simulator still provides a good method for evaluating the manipulator performance over longer periods of operation and\or for evaluating joint response to various controllers under typical conditions of time-span and sample period.

5.5.3 Introduction of Hardware Constraints

Naturally, having considered optimised performance of the simulator and having assumed ideal hardware conditions, one is inclined to ask what would happen if these favourable conditions were replaced by more restrictive limitations on hardware.

The unpipelined case is considered first, and the results are used in the similar formulations needed for the pipelined case. The two important performance indices which change under varying hardware constraints are speedup and processor efficiency, so the investigation will be focussed on the behaviour of these two parameters. Before graphically examining these though, it will be necessary to specify a processor allocation scheme, in both the pipelined and unpipelined case, so as to evaluate speedup and efficiency as functions of the number of processors.

5.5.3.1 Effects on speedup and processor efficiency

In the unpipelined case, since the optimal number of processors is three, only three instances need to be considered for speedup evaluation: a uni-processor system, a dual-processor system, and a three-processor multiprocessor system. The first case is trivial and the speedup is unity, since only one processor is available, thus the simulator is implemented sequentially. The second case requires the rescheduling of processes given in Figures 28 and 27. In the function set-up-mat(), (see Figure 27), when calculating the forces on the three joints one processor must evaluate for two joints. Fortunately, the calculations for joints 2 and 3 combined (63.5μs) are less demanding for joint 1 alone (106.9μs), and again in the evaluation of the force derivatives the computational path for joint 1 is longer (229.3μs) than for the other joints (63.6μs). Hence, despite only two processors being available the speedup of the function set-up-mat() remains unaltered. Looking at Figure 28, it can be seen that high-states also has a maximum parallelism of three. With only two processors available one processor executes the longest, or critical, path whilst the other executes the other path(s). The function define-pbar2(), is the longest of the first three parallel paths, thus this assigned to the first processor. The other processor is scheduled the other two paths, which combine to form a slower path, taking 4.112 milliseconds. The remaining operations which have a maximum parallelism of two, are schedule as before. The new execution time for this function, considering set-up-mat() remains unaltered, becomes 4.973 milliseconds. Speedup is re-evaluated, as before, to yield:

speedup $\cong$ 1.677

and from this efficiency is derived as:

Efficiency = Speedup÷2 $\cong$ 0.8385.

The last case, with three available processors, is the optimal assignment scheme alreadydescribed and for which speedup and efficiency have been calculated. Additional processors over and above this optimal number do not enhance speedup, since all tasks are already assigned to processors, but the processor efficiency rate decreases exponentially.

Although it is not mathematically correct to graphically represent speedup versus processor numbers, or efficiency versus processor numbers, as continuous curves, (since processornumbers only exist as whole numbers discrete points are a correct representation), it is nonetheless a good technique for illustrating the trend over the range of processor unit numbers with clarity.

Graphs 5.25 and 5.26 demonstrate the performance of speedup and processor efficiency against processor numbers respectively. The shape of the speedup curve is quite typical for parallelisation of a system with a reasonable degree of parallelism. As the number of processing units is increased the rate of returns on speedup decreases, until the point of no return when maximum speedup is reached. The processor efficiency curve is not particularly untypical but it is noticed that the efficiency does not decrease in as smooth a manner as the speedup increases. This is due to the fact that the speedup over the range zero to three processors has its least rate of increase when going from one to two processors. This can be observed as a slight dip in the speedup curve over this range at the two processor region and is reflected by this observed lack of decrease in processor efficiency in the same region. It is simply a peculiarity of this particular system.

The pipelined case is slightly more difficult to analyse as the range zero to twelve processing elements must be examined, and the processes rescheduled in a manner which optimises speedup under the prevailing circumstances. It is assumed that before the pipeline is operational there must be a minimum of four processors present, one for each pipelining stage. Less than four processors gives a performance identical to the unpipelined case, which has already been considered. For example a uni-processor system would have to calculate each pipeline stage individually which is merely the sequential case. With four processors each stage is scheduled to execute on a separate processor. Each stage takes as long as it would in the sequential case because there is only one processor available to it, but by merit of the fact that all four stages are executing concurrently on different processors a speedup of four is achieved. The processor efficiency is thus unity. The time between successive outputs from the pipeline is equal to the time of the slowest stage, which in this instance is any stage since all stages have equal execution times.

Despite adding a fifth processor to the system, which can be arbitrarily assigned to any of the four stages to speed up that stage by 1.677 (5.544 milliseconds), the slowest execution time remains the same. Thus so to does the time between pipeline outputs, and also the overall speedup. It is possible to assign a portion of the workload from each of the four stages to this processor and this would theoretically increase overall speedup. The practice, however, would be quite different. The overheads involved in dynamically scheduling and partitioning the excess workloads for this additional processor would overwhelmingly outweigh the theoretical gain in speedup, as experienced by Jones and Fleming [93] in their attempt to implement the inverse dynamics of a robot manipulator. Dynamic partitioning and scheduling of the workloads is thus avoided. As a consequent of unchanged speedup the processor efficiency decreases and is given by:

Efficiency = $4.0 \div 5 = 0.8$.

Addition of a sixth and then a seventh processor, although increasing the execution times of two more stages, are not reflected as an increase in overall speedup, as the slowest stage remains constant. The processor efficiency does however decrease. The addition of an eighth processor however brings all four processing stages up to the same speed, 5.544 milliseconds, due to the fact that each stage with two processors assigned to it has a localised speedup of 1.677. As there are four stages executing concurrently with individual speedups of 1.677 the overall speedup becomes:

Speedup = $4 \times 1.677 = 6.708$

and efficiency is formulated as:

Efficiency = $6.708 \div 8 = 0.8385$

In a likewise fashion the addition of three more processors is reflected in the speedup of three of the processing stages, but not in the overall speedup. The addition of a twelfth processor, however, produces a greater speedup, evaluated as follows:

Speedup = $4 \times 2.438 = 9.752$

and efficiency for twelve processors in the pipeline is:

Efficiency = $9.752 \div 12 = 0.81266$

These results are represented graphically in graphs 5.27 and 5.28, which show speedup and efficiency versus the number of processors respectively. The speedup graph, because of the particular scheduling scheme chosen to avoid large dynamic overheads, increases in discrete steps, from the initial speedup of four for a four processor system to a maximum of 9.752 for twelve processors and greater. The number of processors used to implement the simulator therefore should be a multiple of four up to a maximum of twelve, as no other amount of processors is justified (or is necessary), as can be seen from the discrete speedup 'plateaux' of the graph. Graph 5.28, which shows the relationship between efficiency and processor numbers, is derived directly from the speedup graph. It has the appearance of a sawtooth waveform, with the peaks denoting the localised maximum processor efficiency when the number of processors is either 4, 8 or 12. The alternating decreases and increases in the graph are attributed to the discrete levels of speedup which remain constant over given intervals along the x-axis of the graph. In Chapter seven several conclusions with regard to the parallel simulator, both pipelined and unpipeliend, are presented and the overall performance is analysed in the context of an accurate representation of the actual manipulator dynamics. The following section summarises the procedures, the propositions and the prominent aspects of this chapter.

5.6 SUMMARY

The objective of this chapter is to create an accurate parallelised simulator for the PUMA 560 robotic manipulator. To achieve this a validated third order model must first be derived. It is possible to extend the second order model to incorporate the effects of a first order motor model, the relationship between joint voltage and torque, and the substantial joint frictional forces, although it must be noted that the effects of link elasticity and gearing backlash have been ignored apropos in the assumption of their negligible contribution to the manipulator dynamics. In Section 5.1 the complete third order model is derived. By matrix representation the model is placed in a third order state space format, and furthermore by incorporating into this representation the relationships between joint positions and velocities, joint velocities and accelerations and, joint accelerations and their derivatives a more complete ninth order state space model is accomplished.

The robotic model is then incorporated into a complete simulation routine, which includes the Runge Kutta technique of numerical integration necessary to extrapolate the lower states from their derivatives. In Section 5.2.3 the complete computational model is presented. An exploration of the inherent coarse grained parallelism in the simulator is undertaken in Section 5.2, in a piece-wise fashion. Each sub-function within the simulator is examined individually, and then collectively at a more coarse grained level. The parallel functional models and overall parallel model are established, and in Section 5.3 a coding scheme based on these models is presented in Parallel C. A method of pipelining the simulator over four stages is considered and its effect on simulation accuracy is investigated. In the following section, Section 5.4, a specifically tailored Transputer topology, necessary to host the parallel simulator software, is described, and the IMS B008 TRAM motherboard, which hosts the Transputers, is introduced to the reader.

Finally, in the last section on simulator analysis, the performance of the simulator is validated under a series of tests which examine for characteristic PUMA 560 manipulator behaviour in the model dynamics. The overall speedup of the parallel simulator compared to the sequential implementation is evaluated in Section 5.5.2 for both the pipelined case and the unpipelined case and from the results a value for the processor efficiency is derived. The eventuality of reduced availability of processors is also considered and the effect on efficiency is inferred. The results are graphically illustrated for both the pipelined and unpipelined simulators in Appendix C (Graphs 5.25 to 5.31). In Chapter seven conclusions based on observations from these graphs are presented.

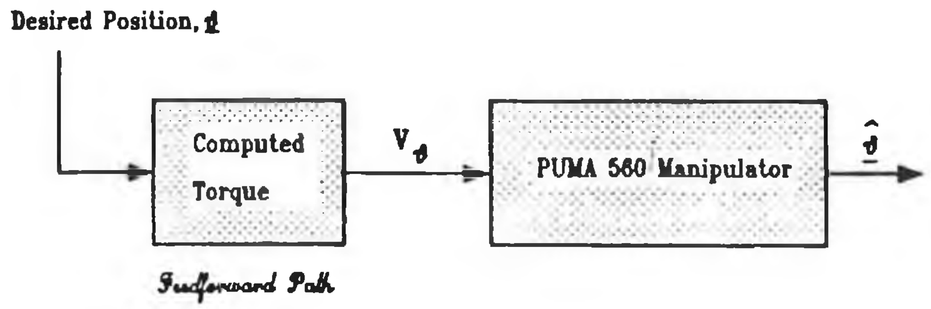
CHAPTER 6

Development of a Parallel Computed Torque Algorithm for Robotic Control Incorporating P-D Feedback

6.1 COMPUTED TORQUE ALGORITHM (FEEDFORWARD CONTROLLER)

Due to the phenemonal growth of robots in the last twenty years there has been a wealth of techniques and schemes developed for robotic control, often based on older and/or existing control algorithms. This has facilitated the usage of robots in demanding and *errorphobic* applications. One of the major obstacles in controlling a robot is not to devise a control regime, but to calculate the control signals to move the robot to the desired positions. There are two aspects to this problem: firstly a path must be determined to bring the robot end-effector to the desired location in an acceptable and efficient manner and secondly the voltages to move the manipulator joints through this chosen trajectory must be calculated. Neither task is trivial. Trajectory planning is an extensive field of study and involves the optimisation of several factors, which is classical problem to be found throughout all mathematical fields of study. The calculation of voltages to move the manipulator to desired set-points involves the evaluation of the inverse dynamics of the manipulator, which is recognised as being computationally intensive. The inverse dynamics of a robot are equations which describe the system input (voltages) in terms of the system outputs (joint positions, velocities and accelerations), as contrasted with the forward dynamics which express the robot output parameters in terms of the joint driving-voltage inputs. This allows the voltages, needed to generate the necessary torque to move the manipulator to a desired position, to be repetitively calculated, as the trajectory is traversed.

The Computed Torque feedforward algorithm is as its name suggests a technique whereby using the inverse dynamical model of the Puma 560 manipulator the torque required to move the manipulator to a desired position is calculated and applied in a feedforward fashion directly to the inputs of the robot. In practice not only are the torques calculated but so also are the voltages to generate these torque, using knowledge of the 100 Watt permanent magnet dc motors used in the joints. Figure 29 shows schematically how this control technique is configured. As there is no feedback mechanism any errors in the final end-effector position are not compensated for, and generally this technique requires an accurate model for acceptable performance. Because most robot models neglect higher order dynamics computed torque without feedback is not usually practical. In Section 6.2 a feedback scheme utilising P-D control for enhanced performance is introduced.


Desired Position, $\underline{\vartheta}$
Computed Torque
V_{ϑ}
PUMA 560 Manipulator
$\hat{\underline{\vartheta}}$
Feedforward Path

Block diagram representation of Computed Torque feedforward control of PUMA 560 manipulator

FIGURE 29

6.1.1 Inverse Dynamics of Robotic Manipulator

Derivation of the inverse dynamical model for the Puma 560 robot manipulator is basedon the equations presented in Section 5.1. Recall that the second-order Euler-Langrangian differential equation of motion for a manipulator with n degrees of freedom is given by:

$$F_i = \sum_{j=1}^{n} D_{ij}\ddot{q}_i + I_{ai}\ddot{q}_i + \sum_{j=1}^{n}\sum_{k=1}^{n} C_{ijk}\dot{q}_j\dot{q}_k + G_i + H_i\dot{q}_i$$

□ Eqn 47.

where,

q_i = position of joint i,

F_i = torque acting on joint i,

I_{ai} = actuator inertia of joint i,

D_{ii} = effective coupling of joint i,

D_{ij} = coupling inertia on i joint due to joint j,

C_{ijj} = centripetal force on i due to joint j,

C_{ijk} = coriolis force on joint i due to joints j and k,

G_i = gravity loading of joint i,

H_i = coefficient of friction for joint i.

and that the differential equation for the equivalent circuit which models the 100

Watt permanent magnet dc motors driving the manipulator joints is:

$$V_i = R_i . i_i + L_i . \frac{di_i}{dt} + k_i^e . \frac{d\omega_i}{dt}$$

□ Eqn 48.

The torque produced by a dc motor is proportional to the armature current of the dc motor:

$$F_i = k_i^t . i_i$$

□ Eqn 48a.

where F_i is the torque experienced at joint i.
The joint position be can related to the motor position by the following equation :

$$\omega_i = N_i . q_i$$

□ Eqn 48b.

where N_i is the gearing ratio of joint i.
Substituting equations 48a & 48b into equation 48 gives the following equation for joint voltage :

$$V_i = k_i^e . N_i . \frac{dq_i}{dt} + [\, R_i F_i + L_i . \frac{dF_i}{dt} \,] / k_i^t$$

□ Eqn 49.

The quantity $\dot{F}_i$ is the derivative of the joint torque and is given by :

$$\dot{F}_i = \sum_{j=1}^{3} (\, D_{ij}\dddot{q}_j + \dot{D}_{ij}\ddot{q}_j \,) + I_{ai}\dddot{q}_i$$

$$+ \sum_{j=1}^{3} \sum_{k=1}^{3} (C_{ijk}\ddot{q}_j \dot{q}_k + C_{ijk}\dot{q}_j \ddot{q}_k + \dot{C}_{ijk}\dot{q}_j \dot{q}_k)$$

$$+ \dot{G}_i + H_i\ddot{q}_i$$

□ Eqn 50.

The total model can then be written as :

$$V_i = k_i^e . N_i . \dot{q}_i + R_i . [\, H_i\dot{q}_i + G_i$$

$$+ \sum_{j=1}^{3} D_{ij}\ddot{q}_i + I_{ai}\ddot{q}_i + \sum_{j=1}^{3} \sum_{k=1}^{3} C_{ijk}\dot{q}_j\dot{q}_k \,] k_i^t \quad +$$

$$L_i . [\ \dot{G}_i + \sum_{j=1}^{3} (D_{ij}\dddot{q}_j + \dot{D}_{ij}\ddot{q}_j) + I_{ai}\dddot{q}_i$$

$$+ \sum_{j=1}^{3}\sum_{k=1}^{3}(C_{ijk}\ddot{q}_j\ \dot{q}_k + C_{ijk}\dot{q}_j\ddot{q}_k + \dot{C}_{ijk}\dot{q}_j\dot{q}_k) + H_i\ddot{q}_i\ \ ]k_i^t$$

□ Eqn 51.

This is the third order model equation for each primary joint of the PUMA 560. The scheme introduced in Section 5.1.2, whereby the above equation was modified to matrix form, again is useful in deriving the inverse dynamical model of the robot manipulator. Recall the following matrix definitions:

$\text{LMAT} = \text{Diagonal}(\ L_1/k_1^t,\ L_2/k_2^t,\ L_3/k_3^t\ )$

$\text{RMAT} = \text{Diagonal}(\ R_1/k_1^t,\ R_2/k_2^t,\ R_3/k_3^t\ )$

$\text{HMAT} = \text{Diagonal}(\ H_1,\ H_2,\ H_3\ )$

$\text{IMAT} = \text{Diagonal}(\ I_{a1},\ I_{a2},\ I_{a3}\ )$

$\text{KMAT} = \text{Diagonal}(\ N_1k_1^e,\ N_2k_2^e,\ N_3k_3^e\ )$

$\underset{\sim}{G}\ = \text{Gravity Vector}(\ G_1,\ G_2,\ G_3\ )$

D = matrix which contains all the effective and coupling inertial terms,

D^1 = matrix which contains the centripetal and coriolis forces experienced by joint 1,

D_2 = matrix which contains the centripetal and coriolis forces experienced by joint 2,

D^3 = matrix which contains the centripetal and coriolis forces experienced by joint 3.

Based on these definitions the following matrix model is derived:

$$\begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \text{LMAT}.[\ D + \text{IMAT}\ ].\begin{bmatrix} \dot{q}_7 \\ \dot{q}_8 \\ \dot{q}_9 \end{bmatrix} +$$

$$(\ \text{LMAT}.\dot{D} + \text{RMAT}.\{\ D + \text{IMAT}\ \} + \text{HMAT}\ +$$

$$\text{LMAT}.\begin{bmatrix} (q_4,\ q_5,\ q_6).D^1 \\ (q_4,\ q_5,\ q_6).D^2 \\ (q_4,\ q_5,\ q_6).D^3 \end{bmatrix}\ ).\begin{bmatrix} q_7 \\ q_8 \\ q_9 \end{bmatrix} +$$

$$\Big(\text{LMAT}.\begin{bmatrix} (q_7,\ q_8,\ q_9).D^1 \\ (q_7,\ q_8,\ q_9).D^2 \\ (q_7,\ q_8,\ q_9).D^3 \end{bmatrix} + \text{RMAT}.\begin{bmatrix} (q_4,\ q_5,\ q_6).D^1 \\ (q_4,\ q_5,\ q_6).D^2 \\ (q_4,\ q_5,\ q_6).D^3 \end{bmatrix}$$

$$+ \text{LMAT}.\begin{bmatrix} (q_4,\ q_5,\ q_6).\dot{D}^1 \\ (q_4,\ q_5,\ q_6).\dot{D}^2 \\ (q_4,\ q_5,\ q_6).\dot{D}^3 \end{bmatrix} + \text{RMAT}.\text{HMAT} + \text{KMAT}\Big).\begin{bmatrix} q_4 \\ q_5 \\ q_6 \end{bmatrix}$$

$$+ \text{LMAT}.\dot{\underset{\sim}{G}} + \text{RMAT}.\underset{\sim}{G}$$

□ Eqn 52.

The following quantities are defined to simplify the model equation :

1. $\underline{D} = \text{LMAT}.\big[\ D + \text{IMAT}\ \big]$

□ Eqn 53a.

2. $\underline{P}(q) =$

$$\Big(\text{LMAT}.\dot{D} + \text{RMAT}.\{\ D + \text{IMAT}\ \} + \text{HMAT}\ +$$

$$\text{LMAT}.\begin{bmatrix} (q_4,\ q_5,\ q_6).D^1 \\ (q_4,\ q_5,\ q_6).D^2 \\ (q_4,\ q_5,\ q_6).D^3 \end{bmatrix} \Big).\begin{bmatrix} q_7 \\ q_8 \\ q_9 \end{bmatrix} +$$

$$\Big(\text{LMAT}.\begin{bmatrix} (q_7,\ q_8,\ q_9).D^1 \\ (q_7,\ q_8,\ q_9).D^2 \\ (q_7,\ q_8,\ q_9).D^3 \end{bmatrix} + \text{RMAT}.\begin{bmatrix} (q_4,\ q_5,\ q_6).D^1 \\ (q_4,\ q_5,\ q_6).D^2 \\ (q_4,\ q_5,\ q_6).D^3 \end{bmatrix}$$

$$+ \text{LMAT}.\begin{bmatrix} (q_4,\ q_5,\ q_6).\dot{D}^1 \\ (q_4,\ q_5,\ q_6).\dot{D}^2 \\ (q_4,\ q_5,\ q_6).\dot{D}^3 \end{bmatrix} + \text{RMAT}.\text{HMAT} + \text{KMAT}\Big).\begin{bmatrix} q_4 \\ q_5 \\ q_6 \end{bmatrix}$$

$$+ \text{LMAT}.\dot{\underset{\sim}{G}} + \text{RMAT}.\underset{\sim}{G}$$

□ Eqn 53b.

Hence the model equation can be written as :

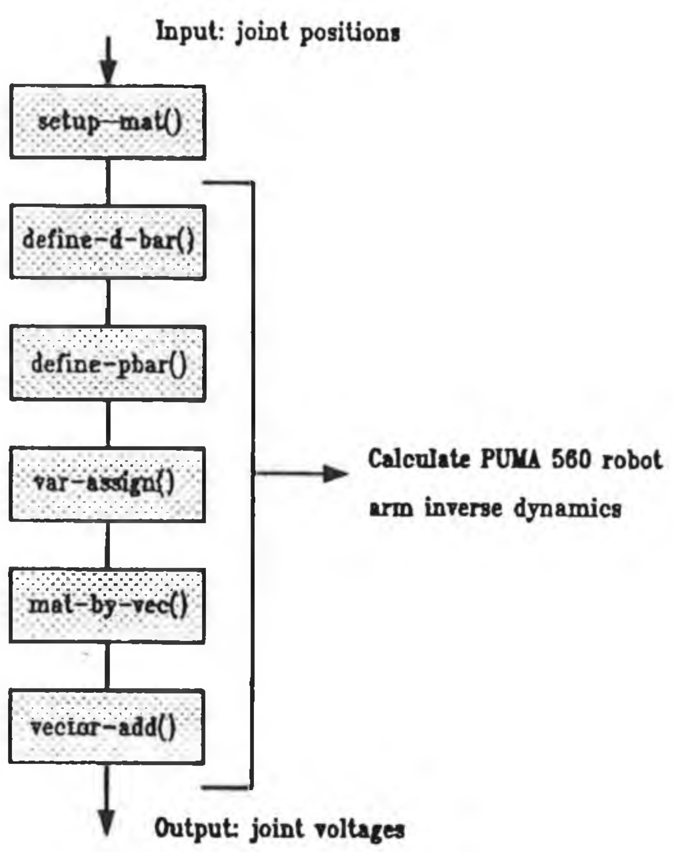
$$\begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \underline{D}.\begin{bmatrix} \dot{q}_7 \\ \dot{q}_8 \\ \dot{q}_9 \end{bmatrix} + \underline{P}(q)$$

□ Eqn 53c.

This model relates the voltage on the joint motors to the joint acceleration derivatives, and is defined as the inverse dynamical model of the PUMA 560 robotic manipulator, upon which the computed torque algorithm is founded.

6.1.2 Sequential Flow Chart of Software Model of Algorithm

The sequential flow of operations for the computed torque software routine is best illustrated by a flow diagram. Figure 30 shows the sequence of computations necessary to evaluate the inverse dynamics of the robot manipulator. The function notation developed in Section 5.3 holds true in this chapter, as do the function definitions, although these will be described for completeness.



Sequential flow diagram of Computed torque function { com-tor() }

FIGURE 30

The input to the algorithm is defined as joint positions. These set-points are derived by trajectory planning. This is necessary as the algorithm always attempts to generate a signal to reach the desired end-effector position in one step and without intermediate points the algorithm attempts to reach the final location in one sample period, which in general is not possible. The set-points along the path are stored in a data file, from which the computed torque function reads values from specific data fields. The first function executed is setup-mat(). This function is basically the initialising and assignment routine needed to establish the matrices and vectors used in the inverse dynamical model evaluation. Following this function is the define-d-bar() routine. This involves the evaluation of equation 53a. Likewise the next function, define-pbar(), involves implementing equation 53b. The ensuing three functions, var-assign(), mat-by-vec() and vector-add(), are a simple variable assignment routine, a matrix by vector multiplication and an elementary vector addition respectively. These latter three functions constitute only a small portion of the overall function execution time, as will be seen in Section 6.6. The outputs of the function are the voltages required to generate the joint torques to achieve the new set-point positions. These signals are made available to the actual robot manipulator, or are applied to the model during controller evaluation.

6.1.3 Flow Diagram Exhibiting Medium/Coarse Grain Parallelism

Establishing the overall parallelism inherent in the computed torque controller is best done by examining each individual sub-function for parallelism separately, and then, by inspection, determining whether complete sub-functions may be run concurrently. It is found that it is in fact possible to decompose two of the more significant functions into parallel computational paths. The first function, setup-mat(), is practically identical to that described in Section 5.2, and may be decomposed as shown in Figure 31.

This decomposition is based on the separation of calculations in a *joint-wise* fashion. The maximum parallelism reflects this and is equal to three. The various tasks shown in the schematic diagram for setup-mat() are explained completely in Section 5.2 and it is deemed not necessary to repeat these explanations. This parallel model may be incorporated into the overall parallelised algorithm. Of the remaining functions define-p-bar() may be decomposed into two processes, of almost equal execution times, thus giving a localised speedup of almost two. The two sub-processes are defined to calculate $\underline{P}_1(q)$ and $\underline{P}_2(q)$. The expressions for $\underline{P}_1$ and $\underline{P}_2$ are given as follows:

$\underline{P}_1(q)$ =

$$\Big(\mathrm{LMAT}.\dot{D} + \mathrm{RMAT}.\{ D + \mathrm{IMAT} \} + \mathrm{HMAT} \; +$$

$$\mathrm{LMAT}.\begin{bmatrix} (q_4,\ q_5,\ q_6).D^1 \\ (q_4,\ q_5,\ q_6).D^2 \\ (q_4,\ q_5,\ q_6).D^3 \end{bmatrix} \Big\} . \begin{bmatrix} q_7 \\ q_8 \\ q_9 \end{bmatrix} +$$

$$+ \ \mathrm{LMAT}.\dot{\underset{\sim}{G}} + \mathrm{RMAT}.\underset{\sim}{G}$$

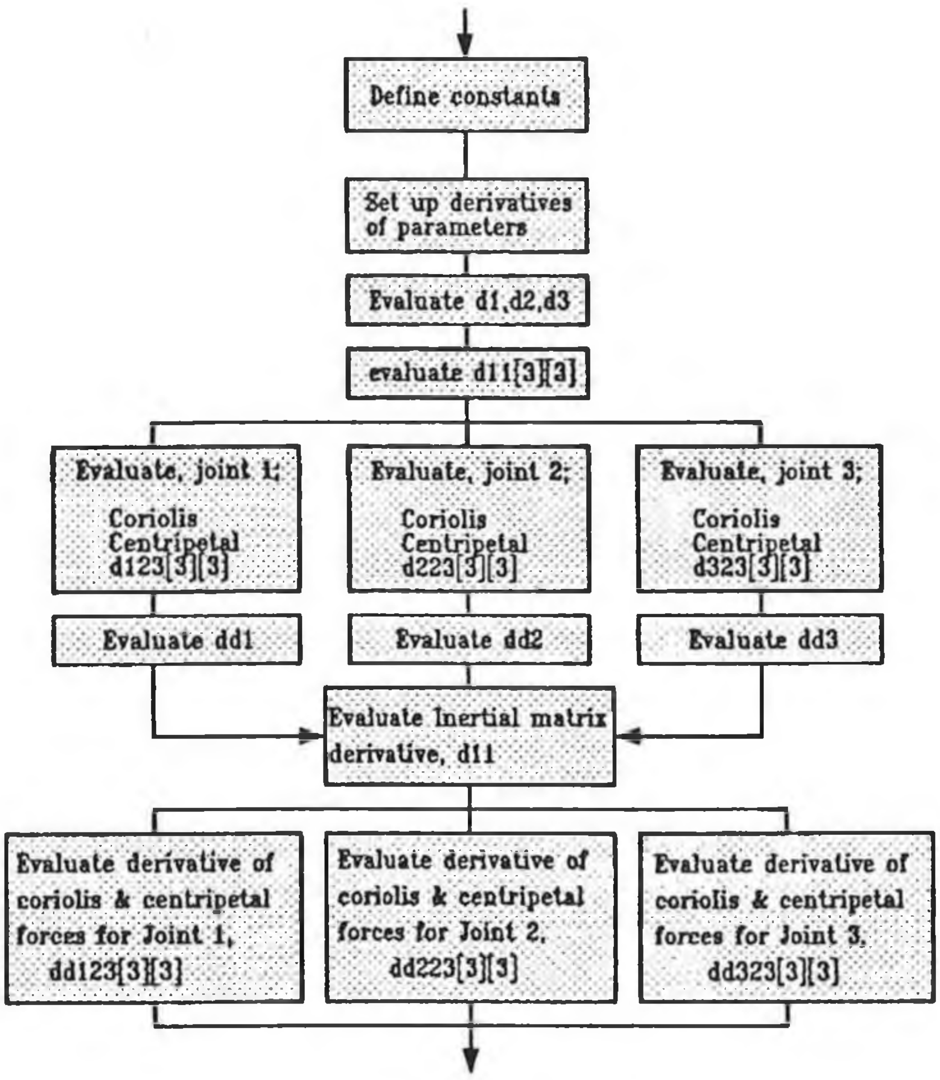
□ Eqn 54.

$$\underline{P}_2(q) =$$

$$\left(\text{LMAT}.\begin{bmatrix} (q_7,\ q_8,\ q_9).D^1 \\ (q_7,\ q_8,\ q_9).D^2 \\ (q_7,\ q_8,\ q_9).D^3 \end{bmatrix} + \text{RMAT}.\begin{bmatrix} (q_4,\ q_5,\ q_6).D^1 \\ (q_4,\ q_5,\ q_6).D^2 \\ (q_4,\ q_5,\ q_6).D^3 \end{bmatrix} \right.$$

$$\left. + \text{LMAT}.\begin{bmatrix} (q_4,\ q_5,\ q_6).\dot{D}^1 \\ (q_4,\ q_5,\ q_6).\dot{D}^2 \\ (q_4,\ q_5,\ q_6).\dot{D}^3 \end{bmatrix} + \text{RMAT}.\text{HMAT} + \text{KMAT} \right).\begin{bmatrix} q_4 \\ q_5 \\ q_6 \end{bmatrix}$$

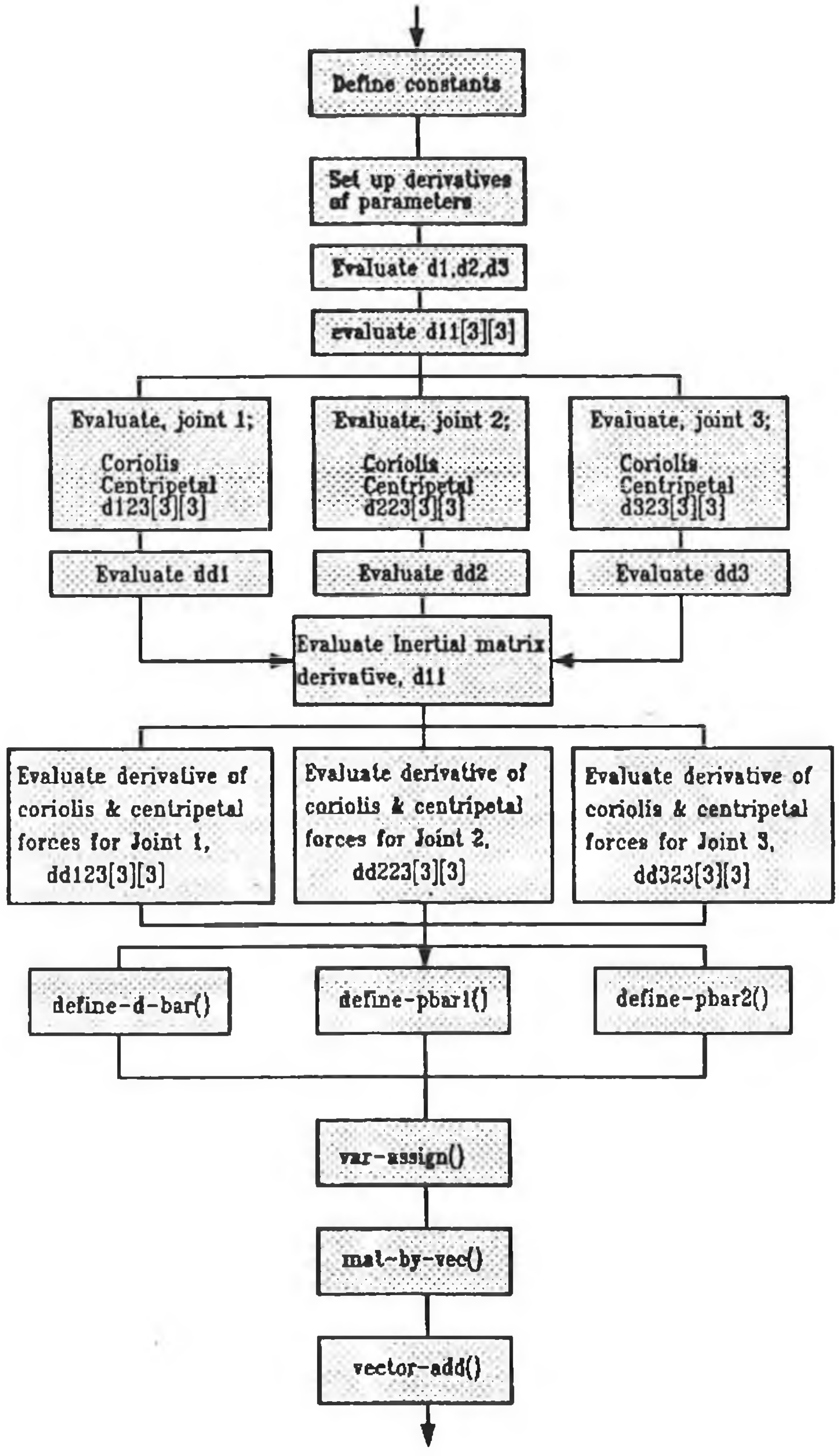
□ Eqn 55.


Define constants
Set up derivatives of parameters
Evaluate d1,d2,d3
evaluate d11[3][3]
Evaluate, joint 1; Coriolis Centripetal d123[3][3]
Evaluate, joint 2; Coriolis Centripetal d223[3][3]
Evaluate, joint 3; Coriolis Centripetal d323[3][3]
Evaluate dd1
Evaluate dd2
Evaluate dd3
Evaluate Inertial matrix derivative, d11
Evaluate derivative of coriolis & centripetal forces for Joint 1, dd123[3][3]
Evaluate derivative of coriolis & centripetal forces for Joint 2, dd223[3][3]
Evaluate derivative of coriolis & centripetal forces for Joint 3, dd323[3][3]

Block diagram illustration of parallel form of function set-up-mat().

FIGURE 31

The complete parallel software model may be represented in block diagram form as shown in Figure 32.


Define constants
Set up derivatives of parameters
Evaluate d1,d2,d3
evaluate d11[3][3]
Evaluate, joint 1; Coriolis Centripetal d123[3][3]
Evaluate, joint 2; Coriolis Centripetal d223[3][3]
Evaluate, joint 3; Coriolis Centripetal d323[3][3]
Evaluate dd1
Evaluate dd2
Evaluate dd3
Evaluate Inertial matrix derivative, d11
Evaluate derivative of coriolis & centripetal forces for Joint 1, dd123[3][3]
Evaluate derivative of coriolis & centripetal forces for Joint 2, dd223[3][3]
Evaluate derivative of coriolis & centripetal forces for Joint 3, dd323[3][3]
define-d-bar()
define-pbar1()
define-pbar2()
var-assign()
mat-by-vec()
vector-add()

Parallel Decomposition of computed torque function, com-tor()

FIGURE 32

Once again the maximum parallelism is found to be three. This creates an upper bound on speedup of three, although it would be impossible to achieve this due to the presence of other computational paths of finite execution time, with a parallelism which is less than three. The define-d-bar(), define-pbar1() and define-pbar2() functions may be executed concurrently, as illustrated. It will be found in Section 6.6 during the speedup analysis that the processor executing the define-d-bar() function spends considerable time awaiting the other two computational paths to complete and this contributes to overall processor inefficiency. The evaluation of speedup is based on this model of execution and the results of timing measurements.

6.2 PD FEEDBACK CONTROL

As mentioned in Section 6.1 control of the Puma 560 manipulator solely by the computed torque algorithm is deficient in many respects. An error occurring in joint position is not compensated for in this method, and for applications which have limited workspace environments this could result in damage due to mispositioning of the robot arm on the assumption that the set-point has been achieved. Furthermore, errors occurring in end-effector position every sample period would accumulate as time progressed, since no feedback signal is available to indicate the misalignment in position. There is also the problem of non-linearity in the robot model. This requires varying degrees of compensation, due to inaccuracies in the inverse dynamical model, depending on the location, velocity and acceleration of the joints.

These problems may be overcome by incorporating a feedback compensation mechanism which adds to the feedforward control signal a compensation term which is based on the error and the behaviour of the error between the desired set-points and the actual joint locations and velocities. The feedback scheme used is PD control.

The PD controller behaves in the same manner as a phase lead compensator. A phase lead compensator is used to improve the stability margins of a system. It also increases system bandwidth, thus giving the system a faster speed of response. The drawback of this technique is naturally the likelihood of increased high-frequency noise problems due to the increased high frequency gains. In general, if the generation of a system output depends on the rate of change of the actuating signal then that system is said to possess derivative error compensation. Comparably, if the output generation depends on a scalar times the actuator signal then the system employs proportional control. Combination of these two techniques provides the basis for PD control or *proportional plus derivative control.* The general forms of the proportional and derivative terms are as shown below, respectively:

$$C_p = K_p.e_n$$
$$C_d = K_d(e_n-e_{n-1})/h$$

Where,

C_p: Proportional control term

C_d: Differential control term

K_p: a constant scaling term for proportional control

K_d: a constant scaling term for derivative control

e_n: the error signal at sample time n

h : the sample time.

The following explains in detail the actual algorithm used, and how the feedback terms are calculated. It also describes how each joint is modeled as a linear SISO system and is continuously parameterised by an identification routine, in order to make the PD feedback mechanism more effective.

6.2.1 The P-D Algorithm & Recursive Least Squares Identification

Before examining the PD feedback controller, there are two other important aspects to be considered: firstly, the algorithm is used to control a process represented by a linear model preferably in transfer function form, so it will be necessary to specify a linearisation scheme for the Puma 560 manipulator dynamics so that each joint may be represented by a linear approximation during each sample period. Secondly, assuming the form of the linear model has been established, an identification routine is needed to recursively update the parameters of this model as the manipulator changes position and loading effects, forces, etc. vary.

To obtain a linear model for each joint one must once again consider the equations governing the behaviour of the joints.

Taking the torque equation (Eqn 47),

$$F_i = \sum_{j=1}^{n} D_{ij}\ddot{q}_i + I_{ai}\ddot{q}_i + \sum_{j=1}^{n}\sum_{k=1}^{n} C_{ijk}\dot{q}_j\dot{q}_k + G_i + H_i\dot{q}_i$$

if the coupling and gravity terms are ignored then :

$$F_i = I_{ai}\ddot{q}_i + H_i\dot{q}_i$$

□ Eqn 56.

and,

$$\dot{F}_i = I_{ai}\dddot{q}_i + H_i\ddot{q}_i$$

□ Eqn 57.

Substituting equations 56 and 57 into equation 48 gives :

$$V_i = L_i I_{ai} q_i / k_i^t + (R_i I_{ai} + L_i H_i).\ddot{q}_i / k_i^t$$
$$+ (k_i^e.N_i.k_i^t + R_i H_i).\dot{q}_i / k_i^t$$

□ Eqn 58.

This is a linear model for each of three primary joints of the PUMA 560 robot. It ignores the nonlinear terms which are present in the comprehensive model, so there is
no coupling or gravity terms present. It can also be represented by the following transfer function :

$$\frac{Q(s)}{V(s)} = \frac{b}{s^3 + a_1.s^2 + a_2.s}$$

where,
q = joint position,
v = armature voltage,

$$b = k_i^t/(L_i.I_{ai})$$

$$a_1 = (L_i.H_i + R_i.I_{ai})/(L_i.I_{ai})$$
$$a_2 = (R_i.H_i + k_i^e.N_i.k_i^t)/(L_i.I_{ai})$$

Computing the coefficients of the transfer function results in the following three models,

<u>Linear model for Joint 1</u>:

$$\frac{Q(s)}{V(s)} = \frac{687.1058}{s^3 + 333.46s^2 + 11219.45s}$$

Linear model for Joint 2:

$$\frac{Q(s)}{V(s)} = \frac{225.9552}{s^3 + 333.47s^2 + 6380.87s}$$

Linear model for Joint 3:

$$\frac{Q(s)}{V(s)} = \frac{915.7552}{s^3 + 333.58s^2 + 12853.34s}$$

These transfer function models, which have constant coefficients values, are sufficient for use with simple controllers, but unfortunately due to the nature of robotic movement their accuracy varies, so it is necessary to continuously perform parameter estimation on the models in anticipation of small coefficient changes. This ensures that the errors in the linear model are minimised and that the compensation signals generated by the PD algorithm are made more accurate. The identification routine employed is the recursive least squares algorithm.

In many practical cases of identification the observations are obtained sequentially and it is desirable to obtain least-squares estimates sequentially as N, the number of observations, increases. The process of obtaining estimates sequentially is called *Recursive identification*. Recursive identification is apt to show an improvement in the parameter estimates as N is increased. In obtaining a recursive formula the result obtained for N observations is used to calculate the estimate for N+1 observations. In general the equation representing the estimation procedure is given by [97]:

$$y(N) = C(N)x(N) + \epsilon(N)$$

Where,

y(N): The vector of outputs at time N

$$C(N) = \begin{bmatrix} y(n-1) & \dots & y(0) & u(n) & \dots & u(0) \\ y(n) & \dots & y(1) & u(n+1) & \dots & u(1) \\ \vdots & & \vdots & \vdots & & \vdots \\ y(N-1) & & y(N-n) & u(N) & & u(N-n) \end{bmatrix}$$

$\epsilon(N)$: The error vector

Assuming the Nth estimate is available then the (n+1)st estimate is formulated as follows:

$$\begin{bmatrix} y(N) \\ y(N+1) \end{bmatrix} = \begin{bmatrix} C(N) \\ c(N+1) \end{bmatrix} x(N+1) + \begin{bmatrix} \epsilon(N) \\ \epsilon(N+1) \end{bmatrix}$$

Where,

$$c(N+1) = [y(N):y(N-1): \dots :y(N-n+1):u(N+1):u(N): \dots :u(N-n+1)]$$

The optimal value of x(N+1) may be extracted from this equation to yield (refer to [97] for a complete derivation):

$$x(N+1) = \left| [C^*(N):c^*(N+1)] \begin{bmatrix} C(N) \\ c(N+1) \end{bmatrix} \right|^{-1} [C^*(N):c^*(N+1)] \begin{bmatrix} y(N) \\ y(N+1) \end{bmatrix}$$

By using the elementary matrix inversion lemma

$$(A+BD)^{-1} = A^{-1}-A^{-1}B(I+DA^{-1}B)^{-1}DA^{-1}$$

and by equating terms this equation may be simplified further to give:

$$x(N+1) = \left| [C^*(N)C(N)]^{-1} - \frac{[C^*(N)C(N)]^{-1}c^*(N+1)c(N+1)[C^*(N)C(N)]^{-1}}{1+c(N+1)[C^*(N)C(N)]^{-1}c^*(N+1)} \right| [C^*(N)y(N)+c^*(N+1)y(N+1)]$$

Based on the previous estimation evaluation, the equation for x(N) is:

$$x(N)= [C^*(N)C(N)]^{-1}C^*(N)y(N)$$

Before combining this equation with the equation for x(N+1), to obtain a recursive relationship between estimates, the following definition is made. Let:

$$K(N+1) = \frac{[C^*(N)C(N)]^{-1}c^*(N+1)}{1+c(N+1)[C^*(N)C(N)]^{-1}c^*(N+1)}$$

The equation for x(N+1) now becomes:

$$x(N+1) = x(N) + K(N+1)[y(N+1)-c(N+1)x(N)]$$

This is the desired recursive formula. It provides updated estimates provided one or more observations are available. The present estimate is obtained by adding to the last estimate a correction term, which is proportional to the difference between the observed output, y(N+1), and the estimated output, c(N+1)x(N).

The PD algorithm is supplied with a linear model for each joint from the recursive least squares routine, in the form:

$$T(z) = \frac{b'z + c'}{az^2 + bz + c}$$

Based on this model a compensation signal is calculated. The generalised form of the algorithm is defined as follows:

```
if $b^2 - 4ac < 0$
        $pole1 = -b/2$
        $pole2 = -b/2$
Else
        $pole1 = -b-\sqrt{(b^2-4ac)}$
        $pole2 = -b+\sqrt{(b^2-4ac)}$

If $pole2 > 1$
        $pole2 = 1$
        $pole1 = -b+1$
```

$$K_d = \frac{K_v.T}{(b'+c')}$$

$$K_p = \frac{K_d.(1-pole1)}{h.pole11}$$

$$Q_0 = (K_p.h+K_d)/h$$

$$Q_1 = K_d/h$$

$$\Delta V = Q_0 e_n - Q_1 e_{n-1}$$

$$= K_p e_n + K_d(e_n - e_{n-1})/h$$

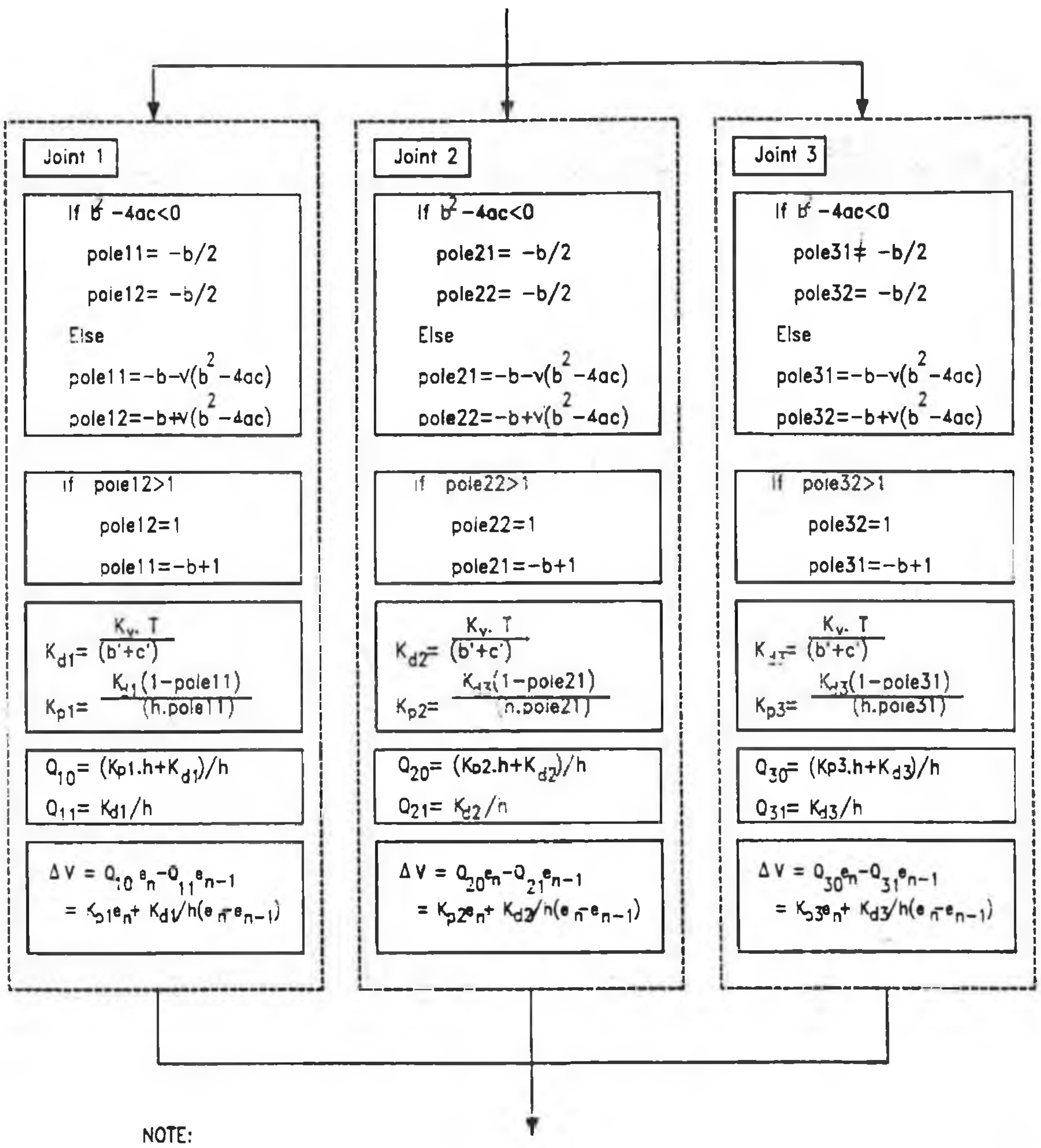
The final control signal is based on both the values of the present and the previous error. Initially the system poles are calculated (i.e. the values which set the denominator of T(z) to zero) and a test is performed to determined whether these are complex or real. If complex the imaginary part is neglected and if real they are left unchanged. The next step is to ensure that the model is stable, by checking whether the magnitudes of the poles are greater than unity, and setting them to unity if the check proves positive. The next task is to evaluate the error-proportional feedback term and the error-derivative feedback term. This is achieved by the equations shown. The proportional term, K_p, and the derivative term, K_d, are multiplied by the error are the error differential over one sample period respectively, to give the final compensation voltage, ΔV. This voltage is added to the voltage signal being applied to the actual robot manipulator joints, to account for the error in end-effector position. This procedure is applied to the three joint linear models, and as will be seen in the next section, Section 6.2.2, the algorithm can be executed separately for each joint.

6.2.2 Coarse Grain Decomposition of Algorithm

The PD algorithm is suitable for both coarse grain and medium grain decomposition. The decomposition of the algorithm in a coarse grain manner will first be considered. As a consequence of the decoupling of the three joints into three separate linear models, it is also possible to generate control signals separately for each joint during each sample period. The PD algorithm consists effectively of three separate controllers, one for each joint, and their calculations are completely uncoupled. Figure 33 illustrates this point more clearly by showing the three concurrent computational paths, and their identical calculations.

Aside from the three control signal computations there are very few overheads in the algorithm, thus ensuring an average parallelism in the algorithm of almost three, as will be seen in Section 6.6 when examining the algorithmic speedup. The use of uncoupled linear models to represent the joints introduces errors due to modelling inaccuracies. These errors are reflected in marginal mis-positioning of the joints, but this produces an increased output error which drives the PD routine. The derivative term is then adjusted accordingly, and also prevents overshoot in the response. Likewise, the proportional term in the PD control evaluation scheme responds in a manner which acknowledges the error magnitude, and proportional control provides the action necessary to reduce steady-state errors.

As a direct result of the need for three separate linear models there are also three separate identification routines to needed to update the models. Figure 34 demonstrates the parallel decomposition of the identification routine.



NOTE:

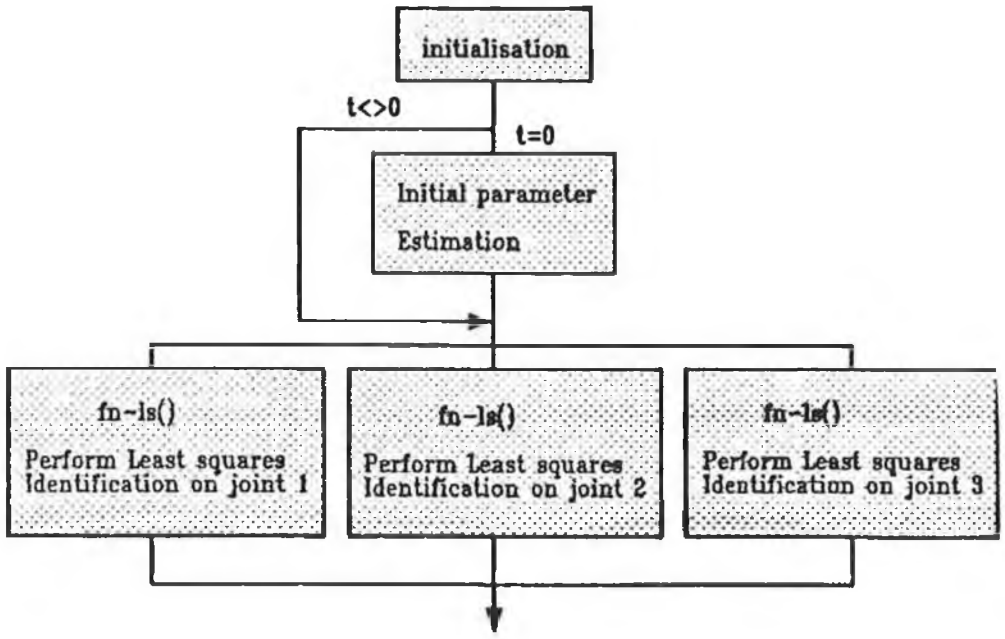
The linear model for each joint is given by the following 2nd order transfer function:

$$\frac{b'z + c'}{az^2 + bz + c}$$

Flowchart illustration of the pd controller showing parallelism inherent in calculations.

FIGURE 33

The first task is to initialise the necessary variables, parameters and matrices. Secondly, a test is performed to see if the routine is making its first run. If so then the linear models must be given initial estimates, and if not then this is not necessary. The computational path of the function can then be split three ways, and each of these paths estimates a new linear model parameter set for each of the joints. The three identification routines employ the recursive least squares method and require past estimates, a system input vector and a system output vector to optimally estimate the current system transfer function parameters.


initialisation
t<>0
t=0
Initial parameter Estimation
fn-ls()
Perform Least squares Identification on joint 1
fn-ls()
Perform Least squares Identification on joint 2
fn-ls()
Perform Least squares Identification on joint 3

Parallel decomposition of Recursive Least Squares routine, rls().

FIGURE 34

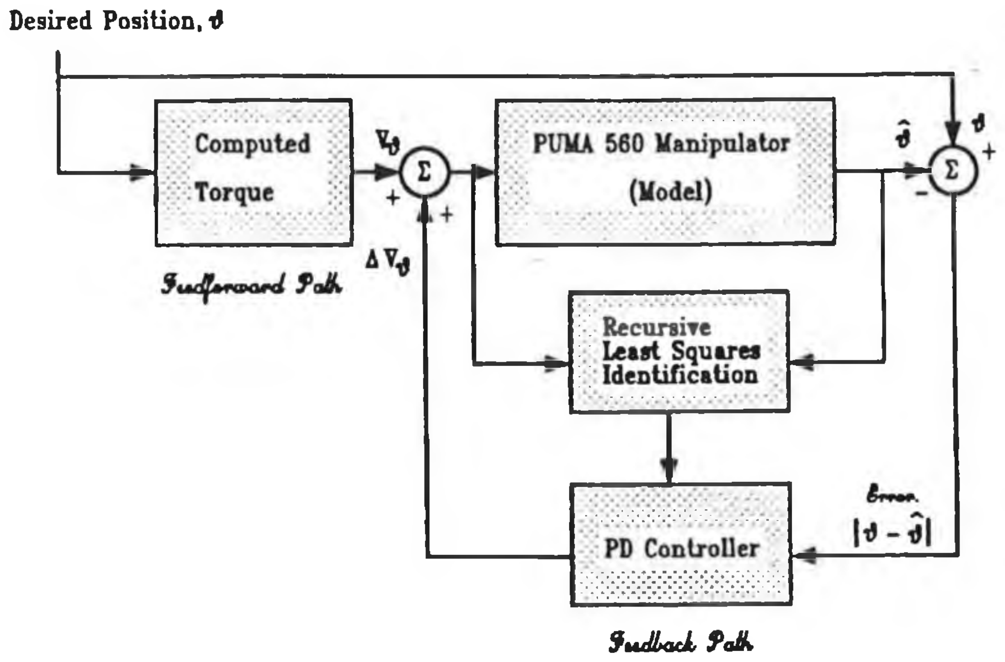
6.2.3 Medium Grain Parallelism

The larger grain parallelism which is found in the PD algorithm is due to the fact that the feedback control of the three joints results in three separate PD controllers. In a sense the parallelism is a result of the system design rather than actual parallelism in the algorithm itself. Examining the algorithm however shows that there is a considerable degree of medium grain parallelism. This parallelism, which manifests itself as a concurrency among the simple arithmetic instructions, is best implemented on an medium grained array architecture. Implementation on any larger grain parallel architecture would be of little benefit due to mismatch in granularity between process and processor. In Section 6.5 this parallelism is examined more closely and a medium grain parallel architecture is proposed.

6.3 COMBINING FEEDFORWARD AND FEEDBACK CONTROL

The preceding sections have separately examined both the feedforward and feedback paths and their effectiveness. It now remains to describe the combination of both these to form a complete controller for the Puma 560 robot manipulator. Figure 35 illustrates the complete controller in block diagram fashion.

The feedforward path consists of the computed torque algorithm which computes a control signal based on the inverse dynamical model of the Puma 560 manipulator, as described in Section 6.1.1. The user initially specifies the desired final destination of the manipulator end-effector. The computed torque algorithm receives an input from a trajectory planning scheme, at each sample period, which defines the new end-effector position. This position serves as input to the inverse dynamical calculations, and the solution of these equations yields the torque necessary to achieve this position in one sample period. The algorithm also calculates the voltage needed to generate the torque required. This voltage is applied to the robot joints, via specialised hardware, or to the robot simulator in the case of controller performance appraisal.



Block diagram representation of Computed Torque feedforward and PD feedback control of PUMA 560 manipulator

FIGURE 35

Naturally enough, inadequacies in the inverse model dynamics manifest themselves as misalignment of the joints to the desired positions. This mismatch in actual position and desired position may be represented by an error signal, $\Delta\theta$. $\Delta\theta$ is used to drive the PD controller in a feedback loop. The PD controller, as described in Section 6.2, is designed to account for the errors typical of modelling inaccuracies, such as positional overshoot of the end-effector. In order for the PD algorithm to calculate a satisfactory compensation signal it must posses an accurate linear model of the system(s) being controlled, in this case the three joints. A recursive least squares estimation is performed on the three joints every sample period and the linear model is supplied to the PD controller. The compensation voltage is then superimposed, or added, to the actual voltage being generated by the feedforward controller. This assumes linearity, and is valid if the control signal updates are applied every sample period and if the sample periods are short enough so that the manipulator is adequately represented by the linear model, which is updated every sample period. For the purposes of controller appraisal the validated simulator for the Puma 560 robot manipulator may be used, so that the full range of inputs may be applied without the possibility of physical damage to the robot.

6.4 SOFTWARE MODEL OF PARALLELISED FEEDBACK & FEEDFORWARD CONTROLLER

The parallel C parallel programming language was introduced in Chapter 5, Section 5.3 and the various C language extension constructs, needed to specify concurrency, were also examined. It is not necessary therefore to repeat this introduction to this language (refer to Appendix B for further reference). However, for completeness and clarity, recall the following language extensions.

The channel datatype:

```
Channel chan1,chan2,chan3;
```

The Transputer link interface:

```
channel *Link0Out = LINKOUT;
```

The system timer:

```
main()
    {
     int t;
     t = timer;
    }
```

The par construct:

```
par{
        statement1;
        statement2;
        statement3;
       }
```

The alt construct:

```
alt{
        guard  guardexpression1:
                code;
        guard  guardexpression2:
                code;
        guard  guardexpression3:
                code;
      }
```

Some of these constructs are used in the coding scheme for the parallel versions of both the computed torque feedforward and PD feedback controllers.

6.4.1 Parallel C Implementation of Computed Torque Algorithm

The flowchart of the computed torque algorithm is given in Section 6.1.2 (Figure 30). The functions are given in sequential format and the notation developed in Section 5.3 is maintained. The first function is the setup–mat function, which may be decomposed quite significantly into parallel form. Figure 31, Section 6.1.2 demonstrates the parallelism in the function. The sequential C programming of the function setup-mat() is given in simplified form as:

```
set-up-mat( x1,x2,x3,x4,x5,x6 ) /*system states*/
 {
        define-constants(...);          /* Define sine and cosine constants*/
        par-derivative(...);            /* Set up parameter derivatives */
        eval-d(...);                    /* Evaluate d1, d2, d3 */
        eval-d11(...);                  /* Evaluate d11[3][3], inertial matrix */

        eval-d123(...);           /* Evaluate coriolis and centripetal */
                                  /* forces for joint 1 */

        eval-d223(...);           /* Evaluate coriolis and centripetal */
                                  /* forces for joint 2 */
```

```
          eval-d323(...);            /* Evaluate coriolis and centripetal */
                                     /* forces for joint 3 */

          eval-dd1(...);             /* Evaluate derivative of d1 */

          eval-dd2(...);             /* Evaluate derivative of d2 */

          eval-dd3(...);             /* Evaluate derivative of d3 */

          eval-dd11(...);            /* Evaluate derivative of inertial */
                                     /* coupling matrix */
          eval-dd123(...);           /* Evaluate derivative of coriolis and */
                                     /* centripetal forces on joint 1 dd123[3][3]*/
          eval-dd223(...);           /* Evaluate derivative of coriolis and */
                                     /* centripetal forces on joint 2 d223[3][3]*/
          eval-dd323(...);           /* Evaluate derivative of coriolis and */
                                     /* centripetal forces on joint 3 d323[3][3]*/

     }
```

The simplified sub-functions given above represent blocks of coding in the actual programme. Based on Figure 31 the processes can be reorganised in parallel form, and the software rewritten to accommodate this specification:

```
setup-mat( x1,x2,x3,x4,x5,x6 ) /*First 6 system states*/
{
seq{
     define-constants(...);          /* Define sine and cosine constants*/
     par-derivative(...);            /* Set up parameter derivatives */

     eval-d(...);               /* Evaluate d1, d2, d3 */
     eval-d11(...);             /* Evaluate d11[3][3], inertial matrix */

     par{
       seq{
          eval-d123(...);            /* Evaluate coriolis and centripetal */
                                     /* forces for joint 1 */
          eval-dd1(...)              /* Evaluate derivative of d1 */
          }
       seq{
          eval-d223(...);            /* Evaluate coriolis and centripetal */
                                     /* forces for joint 2 */
          eval-dd2(...)              /* Evaluate derivative of d1 */
          }
       seq{
          eval-d323(...);            /* Evaluate coriolis and centripetal */
                                     /* forces for joint 3 */
          eval-dd3(...)              /* Evaluate derivative of d1 */
          }
        }

          eval-dd11[3][3]            /* Evaluate derivative of inertial */
                                     /* coupling matrix */
     par{
          eval-dd123(...);           /* Evaluate derivative of coriolis and */
                                     /* centripetal forces on joint 1 dd123[3][3]*/
          eval-dd223(...);           /* Evaluate derivative of coriolis and */
                                     /* centripetal forces on joint 2 d223[3][3]*/
          eval-dd323(...);              /* Evaluate derivative of coriolis and */
                                        /* centripetal forces on joint 3 d323[3][3]*/
        }
    }
}
```

In this listing the parallelism is made explicit, and reflects the parallelism determined by inspection in Section 6.1.3. The next step is to develop a parallel software model for the complete feedforward controller. The readers attention is drawn to Figure 30 in Section 6.1.3 which shows the sequential model for the computed torque algorithm and Figure 32, again in Section 6.1.3, which illustrates clearly the concurrency to be found in the algorithm. The sequential coding for the algorithm is given in generalised form as:

```
    com-tor()
        {
            setup-mat();
            define-d-bar();
            define-pbar();
            var-assign();
            mat-by-vec();
            vector-add();
        }
```

Based on Figure 32 this can be coded in parallel C, to exploit the inherent parallelism, as follows:

```
com-tor()
{
 seq{
     define-constants(...);         /* Define sine and cosine constants*/
     par-derivative(...);           /* Set up parameter derivatives */

     eval-d(...);              /* Evaluate d1, d2, d3 */
     eval-d11(...);                 /* Evaluate d11[3][3], inertial matrix */

     par{
      seq{
         eval-d123(...);            /* Evaluate coriolis and centripetal */
                                    /* forces for joint 1 */
         eval-dd1(...)              /* Evaluate derivative of d1 */
         }
      seq{
         eval-d223(...);            /* Evaluate coriolis and centripetal */
                                    /* forces for joint 2 */
         eval-dd2(...)              /* Evaluate derivative of d1 */
         }
      seq{
         eval-d323(...);            /* Evaluate coriolis and centripetal */
                                    /* forces for joint 3 */
         eval-dd3(...)              /* Evaluate derivative of d1 */
         }
      }

         eval-dd11[3][3]            /* Evaluate derivative of inertial */
                                    /* coupling matrix */
```

```
        par{
            eval-dd123(...);            /* Evaluate derivative of coriolis and */
                                        /* centripetal forces on joint 1 dd123[3][3]*/
            eval-dd223(...);            /* Evaluate derivative of coriolis and */
                                        /* centripetal forces on joint 2 d223[3][3]*/
            eval-dd323(...);            /* Evaluate derivative of coriolis and */
                                        /* centripetal forces on joint 3 d323[3][3]*/
           }

        par{
            define-d-bar();
            define-pbar1();
            define-pbar2();
           }
        var-assign();
        mat-by-vec();
        vector-add();
    }
}
```

The functions given above may be broken down into more elementary functions, but there are no further traces of parallelism, other than localised medium grain parallelism such as matrix operations and vector algebra. In Section 6.6 the execution time for the functions are listed. The performance of the parallel controller, incorporating both feedforward and feedback, is evaluated in terms of speedup over the sequential case, processor efficiency and other performance indices.

6.4.2 Parallel C Implementation of the PD Feedback Controller

The PD controller decomposes effectively into three computational paths. Section 6.2 illustrates this clearly and specifies the individual algorithmic computations. The sequential software model in generalised C coding is given as:

```
pd-control()
    {

        rls();          /* Perform identification on joint1,          */
                        /* joint2 and joint3 sequentially.            */

        /***  Linear models for each joint are now available ***/

        /**   Joint 1, calculate compensation feedback voltage **/
        eval-pole();        /* Evaluate poles of linear model               */
        pole-test();        /*          Test for stability                  */
        eval-kd();          /* Evaluate derivative constant                 */
        eval-kp();          /* Evaluate proportional constant      */
        eval-derivative-control();          /* Calculate differential       */
                                            /* control term                 */
        eval-proportional-control();        /* Calculate proportional       */
                                            /* control term                 */
        delta-voltage();                    /* Derive compensation voltage */

        /**   Joint 2, calculate compensation feedback voltage **/
        eval-pole();        /* Evaluate poles of linear model               */
        pole-test();         /*          Test for stability             */
        eval-kd();           /* Evaluate derivative constant            */
        eval-kp();           /* Evaluate proportional constant     */
        eval-derivative-control();         /* Calculate differential        */
                                           /* control term                  */
        eval-proportional-control();       /* Calculate proportional        */
                                           /* control term                  */
        delta-voltage();                   /* Derive compensation voltage */

        /**   Joint 3, calculate compensation feedback voltage **/
        eval-pole();         /* Evaluate poles of linear model          */
        pole-test();         /*          Test for stability             */
        eval-kd();           /* Evaluate derivative constant            */
        eval-kp();           /* Evaluate proportional constant     */
        eval-derivative-control();         /* Calculate differential        */
                                           /* control term                  */
        eval-proportional-control();       /* Calculate proportional        */
                                           /* control term                  */
        delta-voltage();                   /* Derive compensation voltage */
    }
```

The initial task is to call the identification function, rls(), so as to obtain a linear model for each joint. The recursive least squares function performs a least squares estimation of the parameters of the three linear models in sequence, and the C code is quite straightforward:

```
rls()
 {
    init();
    first-run();
    fn-ls();            /* Identify joint 1 */
    fn-ls();            /* Identify joint 2 */
    fn-ls();            /* Identify joint 3 */
 }
```

The initialisation of variables and the general linear model is performed by the function init(). When the rls() routine is run for the first time, there are no past estimates upon which to base the present estimates, so approximations are used. The first-run() function checks to see if it is a first run and if so generates approximations for the estimates. A least squares routine is called for each of the three joints. This function is denoted as fn-ls() in the code. The generalised C code for the rls() function, which may be modified to explicitly declare concurrency, in parallel C is as follows:

```
rls()
 {
    seq{
        init();
        first-run();
        par{
                fn-ls();       /* Identify joint 1 */
                fn-ls();       /* Identify joint 2 */
                fn-ls();       /* Identify joint 3 */
            }
        }
    }
```

Following the call to rls() in the function pd-control() are the three PD controllers for each joint in sequence. The eval-pole() function simply evaluates the *poles*, or roots of the transfer function denominator, of the linearised system. This is followed by pole-test(), which examines the poles to determine whether the linear model represents a stable or unstable system, and adjust the pole values accordingly. The two ensuing functions, eval-kd() and eval-kp(), are as their names suggest evaluation routines for K_d and K_p. Kd and Kp are the two controller constants for derivative and proportional control respectively, and determine the weighted response of each of the two types of control response. Having computed the two control constants it is then possible to evaluate the derivative and proportional control terms, and this is achieved by calls to the functions eval-derivative-control() and eval-proportional-control(). On completion of these two functions the compensation voltage which is fed back into the robotic manipulator joint actuator inputs is calculated by the function delta-voltage(). This whole routine, which is completed for each of the three joints, is reflected in the parallelised coding of pd-control():

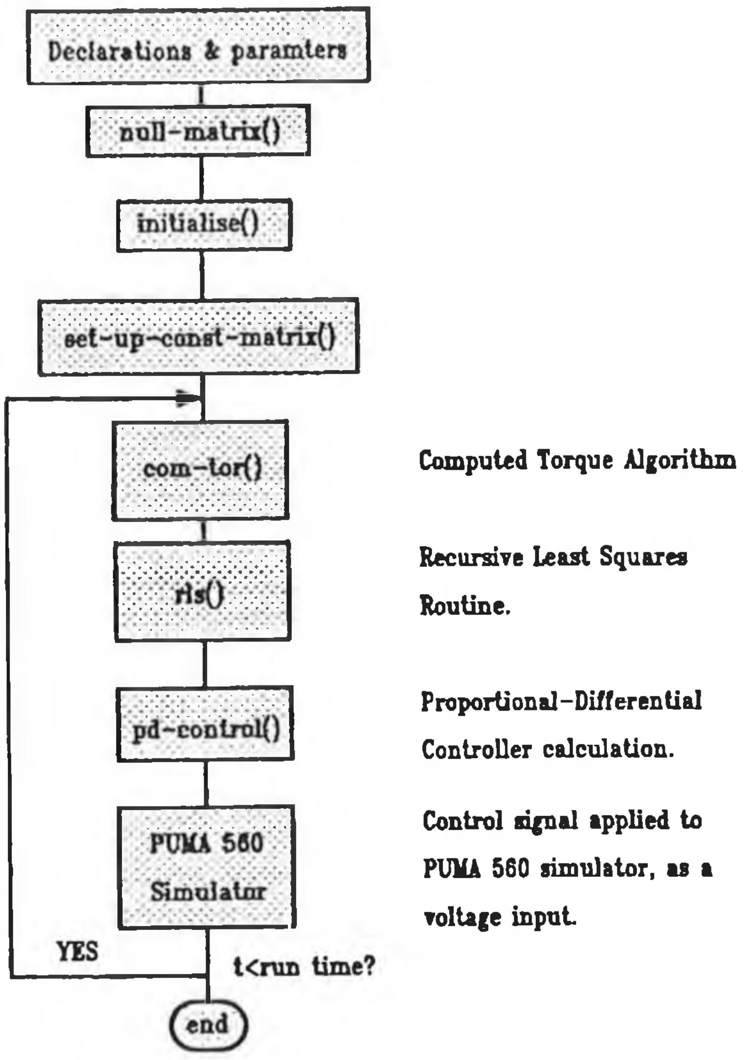
```
pd-control()
 {
     seq{
          init();
          first-run();
          par{
               fn-ls();
               fn-ls();
               fn-ls();
             }

          /***  Linear models for each joint are now available ***/

          par{
           seq{
               /**   Joint 1, calculate compensation feedback voltage **/
               eval-pole();       /* Evaluate poles of linear model          */
               pole-test();       /*          Test for stability          */
               eval-kd();         /* Evaluate derivative constant          */
               eval-kp();         /* Evaluate proportional constant     */
               eval-derivative-control();          /* Calculate differential
*/
                                                   /* control term          */
               eval-proportional-control();        /* Calculate proportional
*/
                                                   /* control term          */
```

```
                delta-voltage();                     /*  Derive  compensation  voltage
*/
             }
          seq{
                /**   Joint 2, calculate compensation feedback voltage **/
                eval-pole();          /* Evaluate poles of linear model          */
                pole-test();          /*           Test for stability          */
                eval-kd();            /* Evaluate derivative constant            */
                eval-kp();            /* Evaluate proportional constant      */
                eval-derivative-control();           /* Calculate differential
*/
                                                     /* control term              */
                eval-proportional-control();         /* Calculate proportional
*/
                                                     /* control term             */
                delta-voltage();                     /*  Derive  compensation  voltage
*/
             }
          seq{
                /**   Joint 3, calculate compensation feedback voltage **/
                eval-pole();          /* Evaluate poles of linear model          */
                pole-test();          /*           Test for stability          */
                eval-kd();            /* Evaluate derivative constant            */
                eval-kp();            /* Evaluate proportional constant      */
                eval-derivative-control();           /* Calculate differential
*/
                                                     /* control term              */
                eval-proportional-control();         /* Calculate proportional
*/
                                                     /* control term             */
                delta-voltage();                     /*  Derive  compensation  voltage
*/
             }
          }
       }
```

This coding specifies that the identification routines on each of the joints should be executed concurrently, and so also should the three PD feedback control signal evaluation routines. In Section 6.5 it will be shown that the three PD controllers may in fact be implemented on a finer grain hardware topology than a Transputer known as the PACE chip. The implications for the generalised software scheme given above are not significant. The PD control of each joint is still performed in parallel and evocation of the controller can still be represented by a call to one function, instead of the several functions above, albeit the coding of the function will be very different to accommodate the interface between the Transputer and the PACE architecture. This will be discussed in more detail in Section 6.5. The maximum parallelism of the algorithm is seen to be three, as a direct consequence of three joints being controlled. In Section 6.6 during analysis of the performance of the parallel controller the average parallelism, which is equivalent to the speedup, will be calculated.

6.4.3 Software Model of Complete Controller

There are effectively three main functions within the complete feedforward/feedback controller: the computed torque algorithm, the recursive least squares estimation routine and the PD controller. In the sequential model these three function are executed in sequence every sample period, thus ensuring that a control signal is generated every sample period. Figure 36a illustrates the sequence of execution of the overall controller, with the parallel simulator used in proxy of the actual robot.



Declarations & paramters
null-matrix()
initialise()
set-up-const-matrix()
com-tor()
Computed Torque Algorithm
rls()
Recursive Least Squares Routine.
pd-control()
Proportional-Differential Controller calculation.
PUMA 560 Simulator
Control signal applied to PUMA 560 simulator, as a voltage input.
YES
t<run time?
end

Sequential model of Computed torque controller incorporating pd feedback

FIGURE 36a

The overall controller is contained in a software loop which continues until the run time has expired. The purpose of a controller is to regulate and direct the robotic manipulator along a fixed trajectory to a desired end position. The main object of a control algorithm is to generate control signals which achieve this as accurately as is possible, under various restrictions. Feedback is often used to reduce the error and add to the stability of the controlled system by making it closed loop. In this instance the user determines the desired end position and a path planning algorithm determines the trajectory of the manipulator, accommodating any special considerations which may exist such as obstacles or limited workspace, and stores the path in a data file. The computed torque algorithm then reads data from this file and calculates the voltages necessary at each step.

The rls() function performs an identification on the robot model, and produces three linear models, which it supplies to the pd-control() routine. This routine, as shown in Section 6.2, calculates voltages to compensate for the error and rate of change of error in the end-effector's position. The computed torque algorithm can be run independently of the rls() routine and the PD controller, thus it may be specified in the software to run concurrently with these two functions. The PD controller must await the outcome of the identification routine before it can proceed to calculate the compensatory control voltages. The PD controller and the identification routine must be executed in sequence. It is possible to run the identification routine in parallel with the PD controller, by supplying the controller with the most recently estimated linear models (i.e. those calculated in the previous sample period). There would be a need to investigate the degradation in performance as a result of doing this. Fortunately it is not necessary to implement this proposal since the combined execution time of the identification routine and the PD controller evaluation function is shorter than that of the more computationally intensive computed torque algorithm. The critical path is therefore unaffected by the running of the functions rls() and pd-control() in parallel and thus the speedup also remains unaffected. As a result the two functions remain in sequential execution order. The software for achieving the overall controller is written in general form as:

```
main()
{
do{
seq{
 par{
    seq{
         define-constants(...);          /* Define sine and cosine constants*/
         par-derivative(...);            /* Set up parameter derivatives */

          eval-d(...);              /* Evaluate d1, d2, d3 */
          eval-d11(...);            /* Evaluate d11[3][3], inertial matrix */

          par{
           seq{
              eval-d123(...);            /* Evaluate coriolis and centripetal */
                                         /* forces for joint 1 */
```

```
               eval-dd1(...)              /* Evaluate derivative of d1 */
               }
            seq{
               eval-d223(...);            /* Evaluate coriolis and centripetal */
                                          /* forces for joint 2 */
               eval-dd2(...)              /* Evaluate derivative of d1 */
               }
            seq{
               eval-d323(...);            /* Evaluate coriolis and centripetal */
                                          /* forces for joint 3 */
               eval-dd3(...)              /* Evaluate derivative of d1 */
               }
             }

               eval-dd11[3][3]            /* Evaluate derivative of inertial */
                                          /* coupling matrix */
         par{
               eval-dd123(...);           /* Evaluate derivative of coriolis and */
                                          /* centripetal forces on joint 1 dd123[3][3]*/
               eval-dd223(...);           /* Evaluate derivative of coriolis and */
                                          /* centripetal forces on joint 2 d223[3][3]*/
               eval-dd323(...);           /* Evaluate derivative of coriolis and */
                                          /* centripetal forces on joint 3 d323[3][3]*/
             }

          par{
               define-d-bar();
               define-pbar1();
               define-pbar2();
             }
         var-assign();
         mat-by-vec();
         vector-add();

      }
    seq{
     seq{
         init();
         first-run();
         par{
               fn-ls();
               fn-ls();
               fn-ls();
            }

         /***  Linear models for each joint are now available ***/

         par{
          seq{
               /**   Joint 1, calculate compensation feedback voltage **/
               eval-pole();       /* Evaluate poles of linear model             */
               pole-test();       /*            Test for stability           */
               eval-kd();         /* Evaluate derivative constant             */
               eval-kp();         /* Evaluate proportional constant      */
               eval-derivative-control();        /* Calculate differential*/
                                                 /* control term              */
               eval-proportional-control();      /* Calculate proportional*/
                                                 /* control term              */
               delta-voltage();                  /* Derive compensation voltage*/
             }
          seq{
              /**   Joint 2, calculate compensation feedback voltage **/
              eval-pole();        /* Evaluate poles of linear model          */
              pole-test();        /*            Test for stability          */
              eval-kd();          /* Evaluate derivative constant             */
              eval-kp();          /* Evaluate proportional constant       */
              eval-derivative-control();           /* Calculate differential*/
                                                   /* control term           */
              eval-proportional-control();         /* Calculate proportional*/
                                                   /* control term          */
              delta-voltage();                     /* Derive compensation voltage*/
            }
          seq{
              /**   Joint 3, calculate compensation feedback voltage **/
              eval-pole();        /* Evaluate poles of linear model          */
              pole-test();        /*            Test for stability          */
              eval-kd();          /* Evaluate derivative constant             */
              eval-kp();          /* Evaluate proportional constant       */
              eval-derivative-control();           /* Calculate differential*/
                                                   /* control term           */
              eval-proportional-control();         /* Calculate proportional*/
                                                   /* control term          */
              delta-voltage();                     /* Derive compensation voltage*/
            }
           }
        }
     apply-control-signal();
   }while(time<run–time);
     }
    }
}
```

The function apply-control-signal() is a fictitious function use to represent the process of relaying the control signals to the Puma 560 controller board or to the robotic simulator. As is evident, one can see that at no point does the parallelism exceed the value of three. The average parallelism of the complete controller will be calculated in Section 6.6, based on this software model, and on the assumption firstly of unlimited processor supply and secondly in the eventuality of limited processor availability. In the following section an optimal hardware scheme is developed in an attempt to exploit the inherent parallelism of the controller fully to achieve higher performance margins.

6.5 OPTIMISED HARDWARE STRUCTURE FOR IMPLEMENTATION OF PARALLEL CONTROLLER

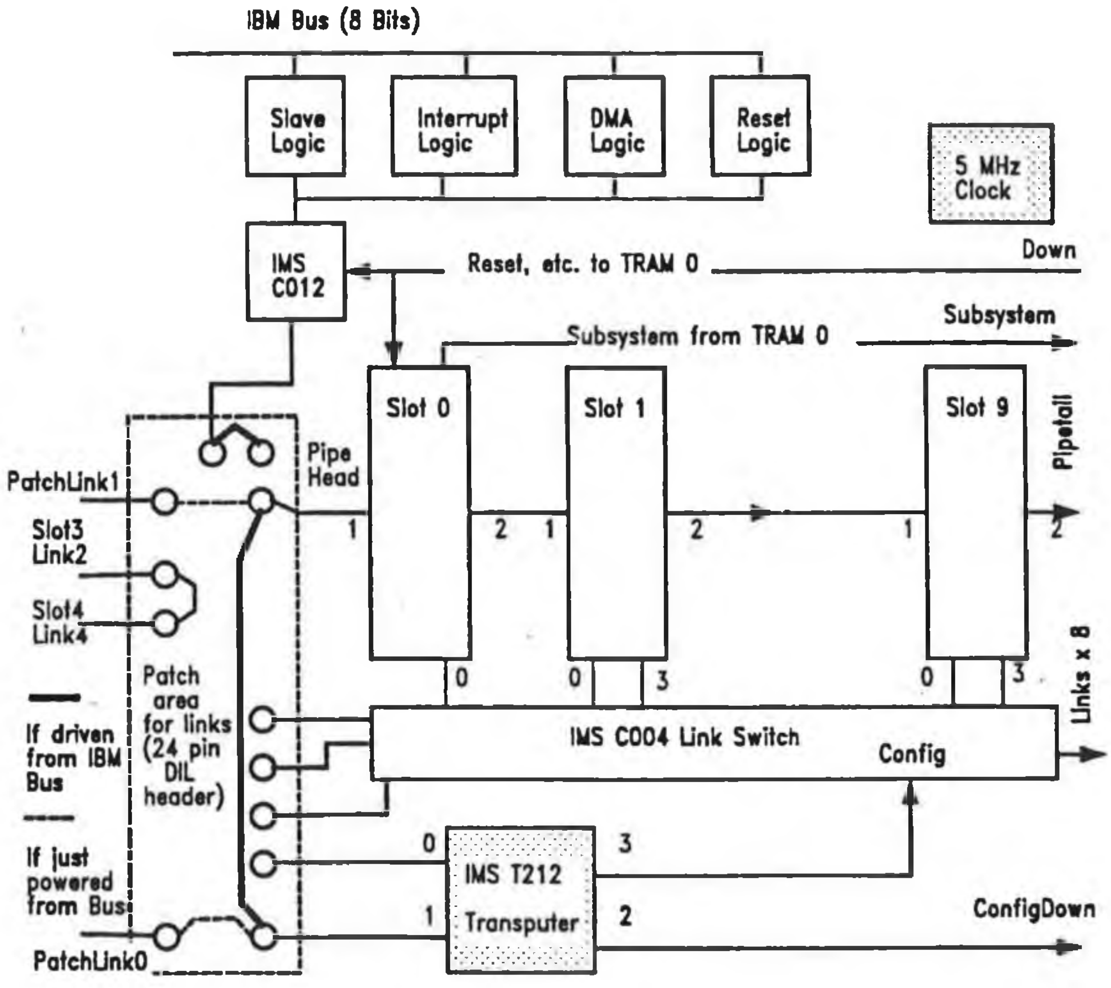
The coarse grain parallel decomposition of the feedforward and feedback algorithmsoutlined above is ideally suited to implementation on a Transputer multiprocessor system, due mainly to matched granularity and speed considerations. The medium grain decomposition of the PD controller algorithm, as mentioned in Section 6.2.3, is better suited to a more finer grain topology than a Transputer system, and in Section 6.5.2 the PACE processing architecture is evaluated for this purpose. By combining a multiprocessor system, of large granularity, with a finer grain array processor one can tailor the hardware to the specific software application, thus exploiting the full range of inherent parallelism. The major drawback is of course the probable loss of flexibility in the multiprocessor architecture to accommodate significant alterations or modifications in the software and the inability to expand the number of processors without having to change the software to re-establish optimality between the application and the hardware topology. In order to reduce scheduling overheads each concurrent process is assigned to a single processor, thus avoiding the need to employ complex scheduling algorithms which are necessary in applications where the one-to-one relationship between processes and processors is not specified.

In Chapter 5, Section 5.4.1, the Transputer processor architecture is profiled in detail, and in Section 5.4.2 a technical description is given of the IMS B008 motherboard, which can house several Transputers. It shall therefore not be necessary to give as detailed a description of either the Transputer or the IMS B008 motherboard in this section, and the reader is referred to the respective sections and references [89] & [94] for further information.

6.5.1 The Transputer and Coarse Grain Parallelism

The Parallel C coding of the computed torque and PD algorithms is given in Section 6.4. The Parallel C language is written specifically for the Transputer. As with the parallel Puma 560 manipulator simulator the parallel controller for the same is also proposed for implementation on a multi-Transputer system.

The software of the overall controller specifies two main computational paths, each of which is subdivided into a maximum of a further three computational paths. The two main paths are the computed torque feedforward algorithm and the PD control feedback algorithm incorporating the identification routine. Both algorithms have a maximum parallelism of three, thus it is possible to have a maximum of six processes concurrently active at particular times. Naturally, to achieve optimum performance one processor should be made available to execute each of these six processes. A six Transputer system must be designed to host the controller. As with the parallel simulator an IMS B008 motherboard is proposed to house the Transputer network, and since only six Transputers are needed one motherboard is sufficient. Recall the hardware topology of the IMS B008 Transputer management board, as shown in Figure 36b.



Simplified Block diagram representation of IMS B008 Motherboard

FIGURE 36b

A maximum of ten TRAMs (TRAnsputer Modules) may be accommodated on the one motherboard, although to actually achieve this physical alterations, such as extension boards, must be made to the board. The TRAM slots are hardwired so that TRAM(N), link 2 is connected to TRAM(N+1), link 1. This leaves links 0 and 3 available for user defined connections, and to create various topologies. In Figure 36 it can be seen that links 0 and 3 of the TRAMs (except for link 3 of TRAM0) are brought to a switch network known as the IMS C004 link switch. The IMS C004 is a 32 link-in 32 link-out switch, which facilitates the connection of links from different TRAMs. This is achieved by sending the configuration data along the IMS C004 configuration link. The switch is controlled by a T212 16-bit Transputer, housed permanently on the IMS B008 board. It is possible to pipeline the T212's of different IMS B008 motherboards to permit the interconnection of TRAMs on different boards, which fortunately in this instance is not necessary.

The slots chosen for the six TRAMs on the IMS B008 motherboard are slots 0, 1 & 3 for the feedforward controller and slots 4, 8 and 9 for the PD feedback controller and identification routine. In this manner both the *pipehead* (TRAM0, link1) and the *pipetail*(TRAM9, link 2) are included in the configuration. To run an application over the Transputer network it is necessary to establish a control hierarchy among the TRAMs.

TRAM0 is chosen as the application host TRAM. This means that the application runs under the control of TRAM0, and on completion of all the sub-processes on other TRAMs control is returned to TRAM0. TRAM0 is also defined as the source of the **notReset** and **notAnalyse** signals for all other TRAMs in the network. This is achieved by the setting of jumpers on the motherboard. Incidentally, this prevents the control TRAM, TRAM0, from being reset every time the application network is reset, which saves considerable time during debugging. The subsystem ports of the TRAMs must be connected to the subsystem line, see Figure 36, to place the reset, analyse and error functions under the control of TRAM0 and to realise the desired TRAM hierarchical structure. In order for the configuration data to be sent to the IMS C004 switch it is imperative that the **configUp** link of the IMS T212 on-board Transputer (link 1) be connected to the pipehead, which is defined as link 1 of TRAM0 (the TRAM running the MMS[95]). This is accomplished by connection via the patch area, as shown in Figure 36. The IMS C012 link, which is connected to control lines for TRAM0, is also connected to link 1 via the patch area. This facilitates the booting down of this link by TRAM0. Unlike the case of the parallel simulator, there is no need to perform inter-board connections of the various control signals, notSubSystemReset, ConfigDownLinkOut, etc., as the complete system is housed on one motherboard. The remaining unoccupied TRAM slots are bypassed by the use of *pipejumpers*. These are simple cross connectors which are inserted into the TRAM slots to connected the locations for link1 and link2.

It is necessary to specify a softwire file, as described in Chapter 5, to list the number and nature of desired connections between TRAMs. The connection between TRAM3 (link2) and TRAM4 (link1) may be optionally disconnected via the patch area. In this application this connection is removed to break the continuity of the pipeline and create two separate TRAM systems, under the control of TRAM0. The softwire file is specified as:

```
SOFTWIRE
     PIPE 0
           SLOT 0, LINK 3 TO SLOT 3, LINK 3
           SLOT 4, LINK 3 TO SLOT 9, LINK 3
END
```

Along with the softwire file the Module Motherboard Software (MMS) must also be supplied with a hardwire file. This file contains a list of the connections physically present on the board. It is necessary to produce this file so that the MMS can decide whether the connections specified in the softwire file are legitimate. If the softwire connections are legal they are implemented by the MMS by sending the configuration data to the IMS C004 via the IMS T212 Transputer. The hardwire file is written below:

```
DEF boarda
     SIZES
       T2 1
       C4 1
       SLOT 10
       EDGE 10
END

T2CHAIN
     T2 0, LINK C4 0
END

HARDWIRE
     SLOT 0,LINK 2 TO SLOT 1,LINK 1
     SLOT 1,LINK 2 TO SLOT 2,LINK 1
     SLOT 2,LINK 2 TO SLOT 3,LINK 1
     SLOT 3,LINK 2 TO SLOT 4,LINK 1
     SLOT 4,LINK 2 TO SLOT 5,LINK 1
     SLOT 5,LINK 2 TO SLOT 6,LINK 1
     SLOT 6,LINK 2 TO SLOT 7,LINK 1
     SLOT 7,LINK 2 TO SLOT 8,LINK 1
     SLOT 8,LINK 2 TO SLOT 9,LINK 1

     C4 0,LINK 10 TO SLOT 0,LINK 3

     C4 0,LINK 1 TO SLOT 1,LINK 0
     C4 0,LINK 11 TO SLOT 1,LINK 3

     C4 0,LINK 2 TO SLOT 2,LINK 0
     C4 0,LINK 12 TO SLOT 2,LINK 3

     C4 0,LINK 3 TO SLOT 3,LINK 0
```

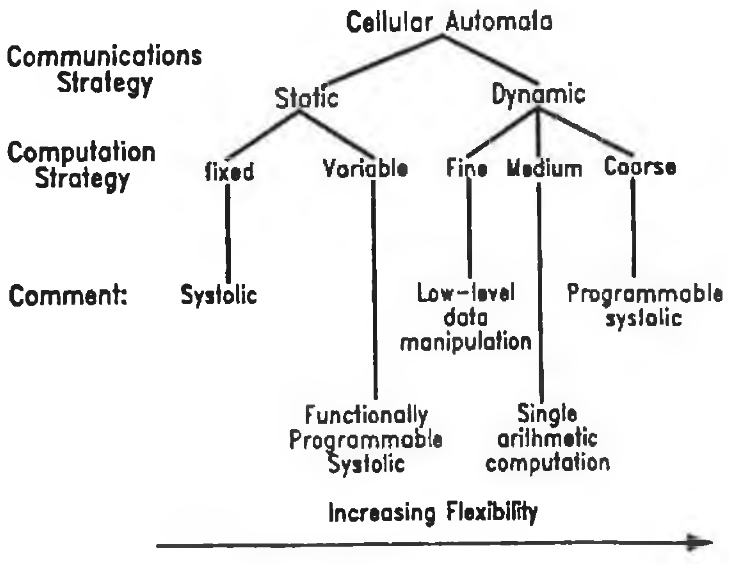
```
    C4 0,LINK 13 TO SLOT 3,LINK 3

    C4 0,LINK 4 TO SLOT 4,LINK 0
    C4 0,LINK 14 TO SLOT 4,LINK 3

    C4 0,LINK 5 TO SLOT 5,LINK 0
    C4 0,LINK 15 TO SLOT 5,LINK 3

    C4 0,LINK 6 TO SLOT 6,LINK 0
    C4 0,LINK 16 TO SLOT 6,LINK 3

    C4 0,LINK 7 TO SLOT 7,LINK 0
    C4 0,LINK 17 TO SLOT 7,LINK 0

    C4 0,LINK 8 TO SLOT 8,LINK 0
    C4 0,LINK 18 TO SLOT 8,LINK 3

    C4 0,LINK 9 TO SLOT 9,LINK 0
    C4 0,LINK 19 TO SLOT 9,LINK 3

    C4 0,LINK 20 TO EDGE 0
    C4 0,LINK 21 TO EDGE 1
    C4 0,LINK 22 TO EDGE 2
    C4 0,LINK 23 TO EDGE 3
    C4 0,LINK 24 TO EDGE 4
    C4 0,LINK 25 TO EDGE 5
    C4 0,LINK 26 TO EDGE 6
    C4 0,LINK 27 TO EDGE 7
    C4 0,LINK 28 TO EDGE 8
    C4 0,LINK 29 TO EDGE 9
END
```

Incidentally, the speed of the serial TRAM links may be set at several different values. For this application a speed of 20Mb/sec is chosen for all links and is accomplished by setting the three-toggle speed switch to 0x000. In this hardwire file only one board is defined. The board is specified to posses one IMS T212 Transputer and only one IMS C004 reconfigurable switch. The T2CHAIN command specifies how the T212's are connected to the IMS C004 switches. In this instance this command is trivial as only one T212 Transputer and one IMS C004 switch are present and only one connection is specified. The main body of the hardwire file describes all the fixed connections present on the motherboard.

The following section describes how the more finer parallelism inherent in the controller (specifically the PD algorithm) may be implemented on an array processor.

6.5.2 PACE <<Programmable Adaptive Control Engine>> Architecture

The PACE chip, which is being developed at the University of Nottingham by Spray and Jones[81,82,83], is a medium grain array processor. It has been designed to support irregularly structured algorithms on a regularly structured array.



Taxonomy of Cellular Automata

FIGURE 37

Figure 37 shows a taxonomy of cellular automata. These autonomous cellular arrays (ACA) have individual PE autonomy. To cope with both regular and irregular flows within algorithms, the requirement for programmability in the communications topology of an array becomes considerable, through the desire for efficient support of general algorithms within low-complexity processing elements. Hence the category of architectures possessing a programmable interconnect is of particular interest when discussing the PACE architecture. The multi-processing element systems shown in Figure 37 are explained further below:

- The fine grained processors operate at bit level, and exploit the finest degree of parallelism
- The medium grained processors consist of PE's which operate at the word level and are suitable for exploiting the parallelism present in algorithmic calculations.
- The coarse grained systems are created by the combination of several processors, all of which are capable of hosting individual programmes.

Based on the above scheme of classification the PACE architecture is categorised as a synchronous medium grained ACA array. Contrasting with a systolic array, which is suitable for exploiting fine grain parallelism in a regular function on a regular array, the PACE architecture is suitable for exploiting medium grain parallelism in both regular and irregular functions. To do this it must support the following constructs:

- □ Sequential operation of instructions,
- □ Iteration,
- □ Alternation

The advantage of the array structure of the PACE is that not only is the medium grain parallelism of a function exploited, but so too is the pipelinability of the function. In supporting irregularity in the application function several aspects have to be considered. The first concern is programmability. A sufficient range of operators must be present to allow for algorithm flexibility, such as multiply/add, logical AND/OR, and division. Too large a range would cause excessive PE complexity, so only the most important and most common operators are included. Related to this concern is the aspect of programmable on-board communications. The optimum communications strategy is generally very application specific, but based on Snyder's scheme [98] an 8-way interconnectivity, in a general purpose scenario, is the maximum most commonly required and this is used in PACE.

An important consideration in array architectures is the need for data synchronisation, for both regular and irregular functions. It is necessary to ensure that the operands meet at the right time at the right processing element and it should independent of the nature of the data. Finally, one must ensure that conditionality is supported. This is one of the basic constructs listed above.

With regard to hardware the PACE architecture can be described at chip level as multiple PEs arranged in a rectangular, 8-way fully duplex locally interconnected array. Programming and testing are achieved through a series of communications lines which pass through all elements. Individual elements within the array may be chosen for either downloading information or for testing by use of the column and row select lines. Where two enabled lines cross, the processing element is selected to input from or output to the global communications lines mentioned above. Global clock and control lines ensure that all elements are operating in the same mode (reset, load, test, run) and are all in phase with the same clock beats.

At the processing element level, each structure is subdivided into the four basic functional blocks:

- □ Communications unit: inter-PE transfer of data and operand selection
- □ Processing unit: stores and decodes instructions, performs tasks on an ALU.
- □ Test unit: makes internal states and operating mode visible to user
- □ Control unit: controls the other 3 units, decodes global clock and select signals

The overall control of the PACE array is of paramount importance to the application under consideration, parallel control of the PUMA 560 manipulator. Each PE must obey the globally broadcast control and clock signals. The globally broadcast control mechanisms are:

□ Reset: All processors are halted, and registers are reset.
□ Load: The selected processor in the array downloads the port communication configuration, the operand source information and the function to be implemented.
□ Test: The selected processing element outputs test results onto the shared data lines, whilst continuing to operate as in the 'process' state.
□ Process: The processor performs the specified function on the data supplied by the operands, and permits data communication from ports, otherwise known as the 'run' state of the processor.
□ Halt: In the absence of any of the above modes the processor is held in a halted state, where information is neither lost nor received.

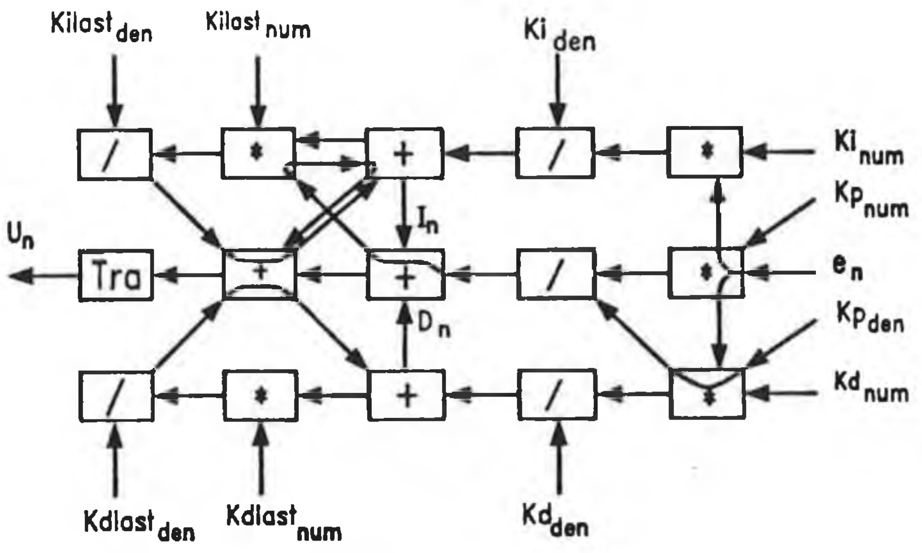
There are four phases to every clock cycle, denoted by the beats $\Phi_0,\Phi_1,\Phi_2,\Phi_3$. Before being supplied to the processing elements each signal is completely decoded. The different beats are explained as follows:

Φ_0: Prepare the processor for input latching.
Φ_1: If in load mode download the configuration data, otherwise latch input data into ports, and operand registers.
Φ_2: Prepare the processor for output data latching.
Φ_3: If not in load mode, latch results from ALU, and latch the outputs of the ports, otherwise do nothing.

The four beats are used to avoid glitching effects in the PACE array processors. For a performance assessment of the PACE array and an analysis of PACE applications the reader is referred to [82]. The following section describes how this architecture may be availed of to implement a standard PID algorithm in an efficient manner, by exploitation of medium grain parallelism. Based on this scheme an adapted PACE configuration is developed in Section 6.5.2.2 for the more elementary PD controller, and this is in turn tailored to suit the feedback requirements of the parallel controller for the PUMA 560 robot manipulator arm.

6.5.2.1 Implementation of PID control on the PACE chip

It was mentioned in the last section that the PACE architecture is being developed in an attempt to realise irregular algorithms on regular array type structures. The PID first order controller has an irregular algorithmic structure, since in the course of its execution it requires multiple interspersed multiply and addition operations and it also 'remembers' and uses previous parameters (i.e. through feedback of the previous modelling error). In a PID controller a control signal is generated according to a function of the current error, an accumulated derivative error term and an integral term over previous errors. Figure 38 illustrates the topology of a PACE realisation of the PID controller.



$$D_n = \frac{Kdlast_{num}}{Kdlast_{den}} \cdot D_{n-1} + \frac{Kd_{num}}{Kd_{den}} \cdot e_n \qquad P_n = \frac{Kp_{num}}{Kp_{den}} \cdot e_n$$

$$I_n = \frac{Kdlast_{num}}{Kdlast_{den}} \cdot I_{n-1} + \frac{Kd_{num}}{Kd_{den}} \cdot e_n \qquad U_n = P_n + I_n + D_n$$

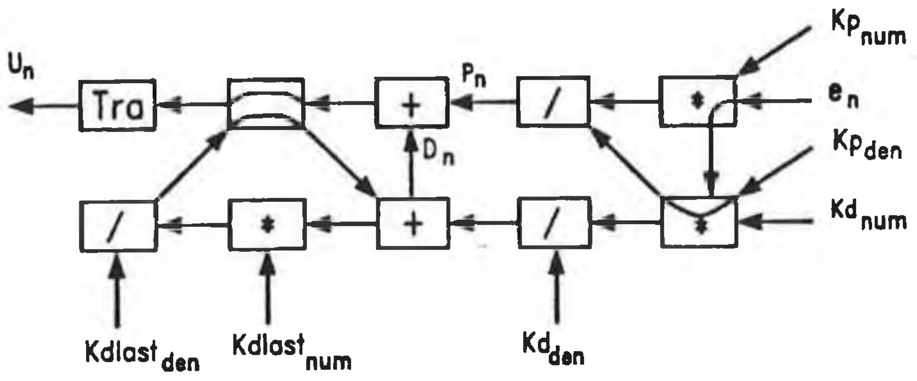
PID Controller Implementation, on the PACE architecture.

FIGURE 38

The array supports two parallel branches of feedback loops, one for the integral term evaluation and the other for the derivative term. The schematic diagram of the PACE array shows how data paths may be directed through other PEs in order to join two PEs separated by more than one row or column. The various numerators and denominators for the three controller terms are fed into the array through a PE on the peripheral of the array (i.e. a PE contained in the first or last row, or in the first or last column). Both the integrative derivative terms include a weighted component based on the last evaluation. The proportional term is a scaled version of the modelling error. The output is sent to an edge PE via a transfer command (TRA). The next section, Section 6.2.2.2, modifies this implementation to accommodate a PD controller, by dismissing the integral term. The resulting topology is further modified to realise the PD control scheme used in the feedback control of the robot arm.

6.5.2.2 Adaptation of the PID implementation to a PD algorithm

Adaptation of the PID controller to a PD controller simply involves the removal of the *integrative* term from the control signal summation, U_n, in Figure 38. Integral control is often used to meet high accuracy requirements and in this instance proportional plus derivative control is sufficiently accurate to account for modelling errors. Figure 39 illustrates the PACE configuration proposed to implement a standard PD controller.



$$D_n = \frac{Kdlast_{num}}{Kdlast_{den}} \cdot D_{n-1} + \frac{Kd_{num}}{Kd_{den}} \cdot e_n$$

$$P_n = \frac{Kp_{num}}{Kp_{den}} \cdot e_n \qquad U_n = P_n + D_n$$

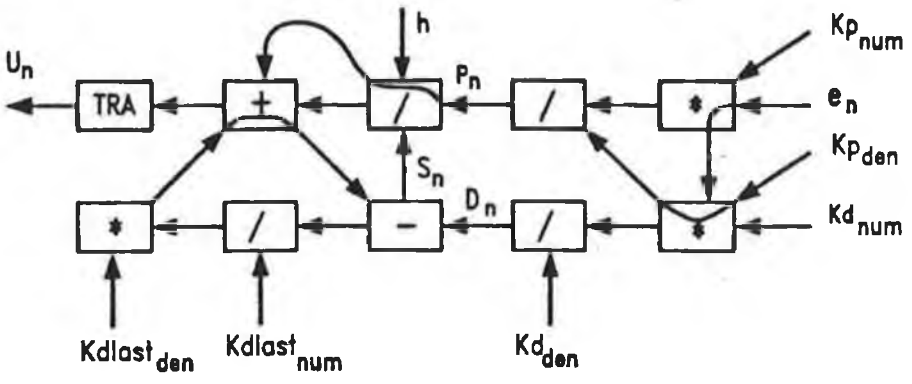
PACE architecture implementation of standard pd controller.

FIGURE 39

The layout is derived directly from the PID realisation, and involves only ten processing elements in a two by five array format. The output from the controller is evaluated by the addition of the proportional and derivative terms, whilst the input remains unchanged as e_n, the current joint positioning error. This error represents the error between actual joint position and the desired joint position. The desired joint positions serve as input to the computed torque feedforward controller and this controller applies the voltages needed to generate the necessary torques to attain these positions. The errors arise mainly due to inaccuracies in the model of the robotic inverse dynamics used by the computed torque feedforward controller.

This PACE implementation of the standard PD controller may be modified to implement the PD feedback controller used in the controller under consideration. The computational model of the PD controller is illustrated in Figure 33. Based on this model the modified PACE implementation proposed is shown in schematic form in Figure 40.

The previous array inputs and outputs remain unaltered and at the same locations, but there is an additional input, the sample period h. This is provided as a separate inputto facilitate future modification in an efficient manner, and doesn't add significantly to the complexity.



$$S_n = \frac{Kd_{num}}{Kd_{den}} \cdot e_n - \frac{Kdlast_{den}}{Kdlast_{num}} \cdot D_{n-1}$$

$$P_n = \frac{Kp_{num}}{Kp_{den}} \cdot e_n \qquad U_n = P_n + S_n/h$$

Where,

$$D_{n-1} = \frac{Kdlast_{num}}{Kdlast_{den}} \cdot e_{n-1}$$

$Kdlast_{num}$ = Previous value of Kd_{num}

$Kdlast_{den}$ = Previous value of Kd_{den}

PACE architecture designed to implement the PD controller used in feedback control of the PUMA 560 robotic manipulator.

FIGURE 40

There is one feedback loop, whereby the scaled error term in the derivative calculation from the last iteration, D_{n-1}, is multiplied by quotient and then subtracted from the current value. The result is then divided by the sample period, h. The final controller feedback term (voltage) is evaluated by adding to the derivative term the proportional term which is scaled version of the current error. This PD control technique is performed for each joint in parallel, thus necessitating three separate PACE PD controller arrays. The following section contains a proposal as to how these PACE arrays may be controlled in a slave-master configuration within a multi-Transputer processing system.

6.5.3 Interfacing the Transputer and the PACE Chip

In order to implement the three joint feedback PD controllers a scheme must be devised to incorporate the PACE architecture with the multi-Transputer system, which hosts the main programme. The most appropriate direction is to assign a Transputer the task of controlling, transmitting to and receiving from the PACE array. Unfortunately, the PACE architecture is not yet generally available, and the interfacing specifications are only schematic. Despite these obstacles it was a strategically decided that for the purposes of illustration and investigation, which are among the thesis objectives, the PACE architecture should remain in the hardware scheme, at least in theory. The specifying of an interfacing prototype between PACE and the Transputer is not a sisyphean task but only a general framework is possible. It would not be difficult to extend this generalisation to incorporate the specific pin-outs, voltages and speed specifications of either architecture.

As specified the Transputer will generate the signals necessary to control the PACE array. A Transputer communicates with the outside world via it's four links. These links are tailored for inter-Transputer communications protocols, but it is possible to develop simple hardware to modify the output. A serial to parallel converter is necessary to bring the output signal into an appropriate format, and likewise a parallel to serial converter is needed to return data from the array. The four links are designated to different channels, each with a different task. One channel would be necessary to communicate with the control Transputer, specified to be TRAM0 in Section 6.5.1. The other three are used as the PE address line, the array control line and a two-way data line. The Transputer links are connected via the IMS C004 switch to the edge-connector and from there to the external interface hardware.

Since only ten PEs are necessary the first two rows and the first five columns need ever be activated. The remaining row and column lines are hardwired to a *unselect* signal. An eight bit address is used, the first two bits are for row selection, the next three are for column selection and the last three bits are redundant (these last three bits are present to conform to a serial to parallel chip standard). One byte may be sent via the appropriate channel to specify the address of the selected PE. Each of the required PEs can be programmed in sequence before operation begins, and each element will maintain its programmed function until the controller has completed operation, or until a global reset is sent. A generalised pseudo-coding scheme to realise the software interface between the Transputer and the PACE array is formulated as follows:

```
PD.BEGIN
    DEFINE address.Kdlast.num
    DEFINE address.Kdlast.den
    DEFINE address.Kd.num
    DEFINE address.Kd.den

    DEFINE address.Kp.num
    DEFINE address.Kp.den

    DEFINE address.h
    DEFINE address.error.in
    DEFINE address.control.out

    DEF CHANNEL chan1 = address.channel
    DEF CHANNEL chan2 = control.channel
    DEF CHANNEL chan3 = data.channel
    DEF CHANNEL chan4 = link.to.TRAM0

    chan2 = RESET                  -- Send global reset to PACE array

         ROW=0:COLUMN=0
         chan1 = ROW.CHANNEL.000 -- Select Processing Element (1 Byte)
         chan2 = LOAD            -- Specify LOAD control command
         chan3 = TRA             -- Send function type
         chan2 = LOAD            -- Specify LOAD for data
         chan3 = PORT.CONFIG     -- Send the comm.s port configuration data

         ROW=0:COLUMN=1
         chan1 = ROW.CHANNEL.000
         chan2 = LOAD
         chan3 = ADD
         chan2 = LOAD
         chan3 = PORT.CONFIG

         ROW=0:COLUMN=2
         chan1 = ROW.CHANNEL.000
         chan2 = LOAD
         chan3 = DIV
         chan2 = LOAD
         chan3 = PORT.CONFIG

         ROW=0:COLUMN=3
         chan1 = ROW.CHANNEL.000
         chan2 = LOAD
         chan3 = DIV
         chan2 = LOAD
         chan3 = PORT.CONFIG

         ROW=0:COLUMN=4
         chan1 = ROW.CHANNEL.000
         chan2 = LOAD
         chan3 = MULT
         chan2 = LOAD
```

```
          chan3 = PORT.CONFIG

          ROW=1:COLUMN=0
          chan1 = ROW.CHANNEL.000
          chan2 = LOAD
          chan3 = MULT
          chan2 = LOAD
          chan3 = PORT.CONFIG

          ROW=1:COLUMN=1
          chan1 = ROW.CHANNEL.000
          chan2 = LOAD
          chan3 = DIV
          chan2 = LOAD
          chan3 = PORT.CONFIG

          ROW=1:COLUMN=2
          chan1 = ROW.CHANNEL.000
          chan2 = LOAD
          chan3 = SUB
          chan2 = LOAD
          chan3 = PORT.CONFIG

          ROW=1:COLUMN=3
          chan1 = ROW.CHANNEL.000
          chan2 = LOAD
          chan3 = DIV
          chan2 = LOAD
          chan3 = PORT.CONFIG

          ROW=1:COLUMN=4
          chan1 = ROW.CHANNEL.000
          chan2 = LOAD
          chan3 = MULT
          chan2 = LOAD
          chan3 = PORT.CONFIG

     LOOP
          kdlastnum     = chan4()   -- Receive constants from TRAM0
          kdlastden     = chan4()   -- in a specified order
          kdnum         = chan4()
          kdden         = chan4()
          kpnum         = chan4()
          kpden         = chan4()
          h             = chan4()
          en            = chan4()

          chan1 = address.Kdlast.num          -- Set up parameters
          chan2 = LOAD
          chan3 = kdlastnum
          chan1 = address.Kdlast.den
          chan2 = LOAD
          chan3 = kdllastden
          chan1 = address.Kd.num
          chan2 = LOAD
          chan3 = kdnum
          chan1 = address.Kd.den
          chan2 = LOAD
          chan3 = kdlden
```

```
            chan1 = address.Kp.num
            chan2 = LOAD
            chan3 = kpnum
            chan1 = address.Kp.den
            chan2 = LOAD
            chan3 = kpden

            chan1 = address.h
            chan2 = LOAD
            chan3 = h
            chan1 = address.error.in
            chan2 = LOAD
            chan3 = en

            chan3 = PROCESS                  -- Instigate PE processing of data

            chan1 = address.control.out
            chan3 = DOWNLOAD
            chan4 = chan3                    -- Send control signal back to TRAM0
      END LOOP
      chan2 = RESET                          -- Send global reset to PACE array
PD.END
```

The above pseudo code serves to illustrate the interfacing protocol between the two processing hardwares. The actual coding, in Parallel C for example, would be highly dependent on the specific addressing, control and data connection protocols of the PACE array. The initial programme instructions are the definitions of the addresses of the various parameters used in the PD algorithm. This is performed only once, as these are fixed addresses. Next, each processing element is assigned an operator using the addressed LOAD command, and again this assignment is performed only once per PE, although were it necessary it would be possible to re-assign the PE with a new operator in a dynamical fashion, which is one of the major advantages of the PACE architecture. A second LOAD command is executed on each PE in the two by five array, this time to supply the configuration information for the PE communications port. This specifies the manner in which the PE is connected to its neighbouring PEs, and is basically a description of the localised topology.

The main body of the programme consists of a loop, containing a list of functions. The first task is the channel based updating of values. Fresh data is read from TRAM0 via chan4, and assigned in a set order to the proportional and derivative terms' parameters. Having received and assigned these values the next step is to transmit these values via the interface to the PACE array, to the originally assigned addresses. This is accomplished by addressing the correct location on chan1, followed by a LOAD command over chan2, and completed by sending the updated value via chan3. This procedure is used to relay all of the parameters in sequence. On completion of this collection of assign/transmit routines the PEs are globally commanded to begin execution of the algorithm, using the PROCESS command broadcast across the array. The PE process the data based on a clock signal and this signal is subdivided into four beats to enhance synchronisation and avoid localised processor glitching. The four beats are described in Section 6.5.2. The final routine is to retrieve the output data from the PACE array, defined as *control.out* and found at the location given by the address *addres.control.out*. The command channel transmits a DOWNLOAD command over the control line, with the above address. The output value is transmitted back to TRAM0 via chan4. It is possible that the feedforward calculations may not yet be complete (on TRAM0, TRAM1 & TRAM3) at this point in time, and the PD controller may spend time waiting for completion to allow transmission over chan4. This processor idling does not effect overall speedup, as the PD controller has completed at this stage, and has to await the end of the sample period to rebegin the loop in any case. The loop is terminated on completion of the controller task. This will be determined by TRAM0, the application host TRAM. The following section considers the performance of the overall system under variable conditions, from unlimited processor availability to a more restrined hardware scenario. For the purposes of performance evaluation, the execution time for the PD controller is taken as being equal to its execution time on a TRAM.

6.6 ANALYSIS OF PERFORMANCE OF PARALLEL CONTROLLER

This previous sections in this chapter illustrate how the parallelism inherent in the computed torque feedforward/PD feedback controller may be exploited at both the medium and coarse grain level to yield higher execution speeds. The actual speedup and processor efficiency, based on the hardware scheme and software model, are calculated in this section.

6.6.1 Evaluation of Controller Output Performance

Before proceeding to analyse the improved performance margins of the controller, the integrity of the controller must first be validated and the accuracy of the control strategy must also be examined. The most appropriate method to illustrate the controller effectiveness is through the use of graphical data. All graphs referenced in this section are contained in Appendix C.

The primary purpose of a robotic controller is to direct the manipulator along a trajectory in a specified manner, by the use of predetermined set-points. Before proceeding to appraise the controller accuracy, one must first choose a desired start-point and end-point for the robotic manipulator end-effector, and then subsequently devise a trajectory of set-points along which the manipulator is directed by the particular control technique. The use of intermediate set-points is a sine qua non for most robotic control strategies, and allows positional targets for each of the joints to be set for each sample period. In the absence of an intermediate set-point the controller may attempt to generate the control signal to move the end-effector to the desired end-position in ***one*** sample period. Obviously, this is generally not possible. Graph 6.1 illustrates the chosen trajecctory, and note that the path is not just ageometric route but also has a time-scale. The objective of the feedforward/feedback controller is to direct the PUMA 560 manipulator along a path which mimics this trajectory as closely as possible, bearing in mind also the path's location in the time domain.

The trajectory is stored in a data file, containing set-points for every 5 milliseconds increment over a total time span of ten seconds, or a total of 2000 points. The controller reads these set-points sequentially and attempts to generate a control signal every sample period to achieve the necessary joint positions, via the computed torque feedforward control strategy. Added to this signal is a compensation control signal driven by the previous positional error and generated by the PD feedback controller. Graph 6.2 illustrates the actual manipulator path, which seems to closely follow the desired trajectory. Note that there are only two time zones when the manipulator joints move significantly: the initial movement from zero-position to the hold positions and the final return to the zero-position. Before examining the error in the joint positioning, there are several items of graphical data to be considered.

The velocity, acceleration and acceleration derivative are illustrated in graphs 6.3, 6.4 and 6.5. The velocities, or the joint positional derivatives, for joint one and three are at a minimum at the point of inflection during the initial manipulator region of movement, and likewise are at a maximum at the point of inflection during the second region of manipulator movement, and vice versa for joint two, which moves in the opposite sense of direction to the other two joints. As a result of the sample period being small (0.005 sec.) compared to the simulation time (10 sec.) the existence of small scale random perturbations in joint velocities at each sample period, due to noise and error compensation control signals, are not evident from the graph and the velocity graphs appear to be apparently smooth curves. Examination of the joint accelerations, however proves the existence of these perturbations. This can be seen from the fact that whilst the graphical representation of the joint accelerations follows a well defined low frequency large amplitude pattern, there is an obvious disruption of the signal by a high frequency component, but of a much smaller amplitude. For an Ideal controller, there should be no very high frequency acceleration components and the manipulator should accelerate in a smooth manner, but in a physical situation jarring of the joints when stopping or starting prevents this being realised. The derivative of the acceleration of a joint should, ideally, be small. The graphical representation of the joint acceleration derivatives, Graph 6.5, further demonstrates the existence of noise in joint positioning, and has significantly large components due to the existence of the high frequency acceleration terms riding on the waveform of the more dominant lower frequency acceleration terms. The voltages generated by the controller, every sample period, to achieve the necessary motor torques to drive the manipulator along the desired trajectory are presented in graphical format in Graph 6.6. The low frequency component of the voltage curve represents the ideal voltages needed to control an ideal manipulator, with a non-noisy motor response, smooth dynamics at start-up and stop, etc. The high frequency component, which over a shorter time span and shorter voltage cross-section would naturally be more evident, is a representation of the controller's response to anomalies in the inverse dynamical model used by the controller, the error in the linearisation technique, the stop/start error in joint behaviour and other less significant related factors.

The final graph shows the error performance over the simulation time. The most noticeable feature of the curve is the absence of a steady state error, which vindicates the usage of PD feedback control. This is a desirable property for most robotic controllers, where the accurate placement of the joints in rest positions and/or the maintaining of fixed positions is important. A second feature is the apparent absence of error overshoot, thus implying that the response of the controller is sufficiently damped to avoid overshoot in joint position, at least in this instance. The prevention/minimisation of overshoot can be attributed to the derivative component of the PD controller which provides action in anticipation of system output (i.e. joint positions) overshoot. The joint position error is most positive at the point where the joint velocity is a minimum, and most negative at the point where joint velocity is a maximum. By numerical analysis of the error response data and the joint velocities it is observed that the error is almost directly proportional to the joint velocity. There is a delay in the change in the error as a function of the velocity, but over the 2000 sample period simulation time it is practically instantaneous. The constant of proportionality is given by:

$$\mathbf{K} \cong -5.166.10^3$$

Furthermore the rate of change of joint positioning error is equal to a scaled version of the joint acceleration curve, with the scaling factor again equal to **K**. The significance of this observation is that the slower the joints are moved then the more accurately the manipulator adheres to the given trajectory, and more specifically, if the manipulator motion is speeded up by a factor of two then the dynamic error approaches a value twice its previous value at a rate almost identical to the instantaneous rate of change of the joint velocity, or acceleration. Thus attempts to impose rapid changes in joint position on the manipulator should be avoided. This can be accomplished by generating the trajectory set-points under certain differential constraints. The overall performance of the controller proves to be acceptable, with a maximum recorded error of 0.009 radians for joint three, which constitutes 0.6% of maximum joint extension, and is an error of 1.8% in joint position at that point.

6.6.2 Speedup and Processor Efficiency in the Parallel Controller

One of the most important objectives, if not the most important objective, of parallelisation is to decrease the execution time of an algorithm, and thus to achieve a speedup greater than unity. Figure 32, in Section 6.1.3, demonstrates clearly the parallel computational paths that exist in the computed torque routine, com-tor(). By examining this model speedup and processor efficiency, in the case of unlimited processor availability, may be evaluated, based on the function execution times for both the INTEL 80286 multiprocessor system, and the multi-Transputer system housed in the IMS B008 motherboard. The times quoted in the following analysis are based on the execution times on the IMS T414-20 Tranputer. Table 10 contains a listing of the execution times for the functions com-tor(), rls() and pd-control(). These times are for sequential execution. Returning our attention to Figure 32, Section 6.1.3, it now possible to evaluate the speedup for the com-tor() function. The initial task in the function is to execute the sub-function setup-mat(). This is not unlike the set-up-mat() routine used in the PUMA 560 simulator, and is decomposed in the same manner. The function is decomposed in *joint-wise* fashion, (i.e. the operations are divided between, and performed separately on, the three joints), yielding a maximum parallelism of three.

	Execution time [ms]	
Function	INTEL 80286 μP	IMS T414-20
com-tor()	28.650	6.101
rls()	10.451	2.226
pd-control()	04.512	0.961

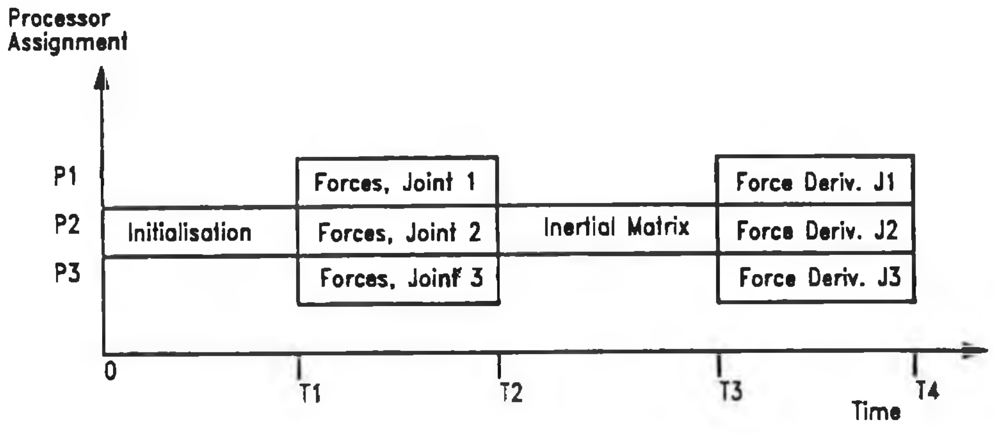
Where, com-tor() : Computed Torque Routine
rls(): Recursive Least Squares Algorithm
pd-control(): PD Feedback Controller

Table showing the execution times of the main functions of the controller

TABLE 10

Recall the schedule of process execution for set-up-mat(), described in Section

5.5, shown in Figure 41. The scheduling sequence for the subtasks for the function setup-mat() is exactly the same as for the function set-up-mat(). Although this function, setup-mat(), is identical in format to the set-up-mat() the sequential execution time in this instance is less, since the subtasks computations are less intensive, but because the algorithmic structure and decomposition technique remain the same the speedup for both functions are practically equal. The length of the longest computational path in the parallel version of setup-mat()is 0.629 milliseconds, about 1.25 times faster than the function set-up-mat().



Schedule of processes for function set-up-mat()

FIGURE 41

This parallel execution time can be incorporated into the parallel version of the feedforward controller com-tor(). The reader's attention is drawn to Figure 32 once more. Having evaluated the setup-mat() function parallel execution time, it remains to evaluate the complete execution time of the parallelised com-tor(). This is achieved by appraising the speed of execution of the define-d-bar(), definep1bar() and define-p2bar() functions in parallel and, the speed of execution of var-assign(), mat-by-vec() and vector-add() in sequence, and finally adding to these two times the execution time of the parallel setup-mat() routine. The slowest of the three functions in parallel is define-p2bar(), which takes 1.815 milliseconds. The three sequential functions, var-assign(), mat-by-vec() and vector-add(), have execution times of 10.6, 87.1 and 29.2 microseconds respectively, giving a combined run-time of 126.9 microseconds. The overall execution time for the computed torque routine, which is equal to the run-time of the longest or critical path, is thus 2.570 milliseconds. The speedup, as defined in Chapter3, may now be calculated on the assumption of unlimited processor availability:

$$\text{Speedup} = \frac{\text{sequential execution time}}{\text{parallel execution time}} = \frac{6.101}{2.570} = 2.374$$

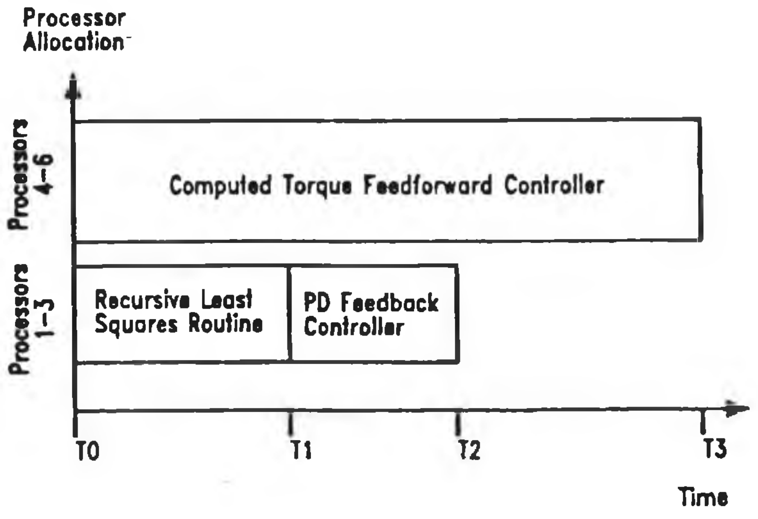
This figure of speedup, which is also equal to the average parallelism of the algorithm, shows a reasonably high degree of parallelism in the computed torque controller when one considers that the function has a maximum parallelism of three. For all systems of equal maximum parallelism the maximum speedup is given as equal to the value of the maximum parallelism. The speedup given above is almost eighty percent of the maximum in this instance. The processor efficiency is derived from the speedup figure:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of processors}} = \frac{2.373}{3} = 0.791$$

These values of speedup and processor efficiency for the inverse dynamical calculations of the computed torque algorithm compare with a speedup and a processor efficiency of 2.438 and 0.8127 respectively for the parallel PUMA 560 simulator, which evaluates the forward dynamical equations of the robot. The speedup of the forward dynamical equations is 1.03 times greater than the speedup of the inverse dynamical equations. This is only a marginal difference reflecting the similarities in the nature of the decomposition of the two problems, including the fact that three manipulator joints are under consideration in both cases. Likewise the processor efficiency, since it is a scaled version of speedup, is also greater for the forward dynamical parallel realisation by the same factor.

The execution of the two functions in the feedback path of the complete controller, rls() and pd-control(), may also be calculated in the same manner. In Section 6.2 both these functions were examined for parallelism. The recursive least squares routine, denoted rls(), is shown in block diagram parallel form in Figure 34, Section 6.2. Note that the first task is to initialise the variables and vectors used in the routine and has an execution time of 60.2 microseconds. The next task is to ascertain whether the call to rls() is the first call in the controller run. If no previous estimates are available, then it is necessary to initiate the parameter estimates. This routine takes 99.2 microseconds to execute. The main body of the programme consists of the least squares estimation of each of the joints. These identifications may be performed concurrently, and the execution time for each routine is 707.7 microseconds. The parallel execution time for the complete function takes 848.0 microseconds in the first call and 748.9 microseconds subsequently, after the initial estimates have been established.

In the feedback loop of the controller each joint is identified by a linear model and controlled by a separate PD controller, implemented by the function pd-control(). The three PD controllers have been specified in Section 6.5.2.2 to run on a PACE array under the supervision of a master programme on a host Transputer. The software modelfor this realisation is also presented in Section 6.5.2.2. The sequential combination of the identification routine and PD controller runs in parallel with the feedforward controller, com-tor(). Note that the combined recursive least squares routine and PD controller must complete execution before the computed torque controller to avoid becoming the critical path of of the overall controller, and lowering the speedup. Considering the recursive least squares routine has a parallel execution time of 748.9μs the maximum execution time permitted for the PD controller is therefore 1.8211ms. The total sequential time for the PD controller is only 0.961ms, and the faster parallel implementation is one third of this at 0.320ms. Because of the general unavailability of the PACE architecture, and the withdrawal of the PACE software simulator [99] for revision, it is not possible to perform timing measurements for the medium-grained PACE array realisation of the PD algorithm, but it is plausible to assume that by incorporating the PACE architecture as specified in Section 6.5.3, the critical path of the overall parallelised controller, consisting of the computed torque feedforward control strategy, remains unaltered. Otherwise if the rls()/pd-control() combination were to become the critical path then the PACE array would have to execute the PD control algorithm at least a speed over four and a half times slower than the Transputer realisation, even allowing for a Transputer-PACE interface communications overhead of 20%. For the purposes of controller evaluation the PD controller may be run on a Transputer, since the overall speedup remains unaffected. A schematic representation of the schedule of process execution in the feedforward/feedback controller is given in Figure 42.



Graphical representation of the block processing schedule in the Feedforward/Feedback controller

FIGURE 42

The graph gives a time-scale representation for each of the three main processes, although not precisely to scale. The critical path is evidently the feedforward path.

There are two main computational paths in the controller, with each of these paths decomposing into a maximum of three subpaths, giving a maximum parallelism of six. The only path of interest in the speedup evaluation is the longest path. With the proviso of unlimited processors this path is equal to the parallel execution time of the com-tor() function, 2.570ms. The overall speedup is given by the expression:

$$\text{Speedup} = \frac{9.287}{2.570} = 3.614$$

Despite a the system having a maximum parallelism of six the speedup falls well short of its upper bound. Among the factors causing this are algorithmic sequentiality, processor idling and a critical path which is significantly longer than the next slowest path, all of which are all effectively inter-related to one another. The processor efficiency is derived from the speedup as follows:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of processors}} = \frac{3.614}{6} = 0.6023$$

The relatively low processor efficiency reflects the presence of a comparitively slow critical path, which also ensures that processor efficiency will remain effectively constant despite a reduction in the number of processors in the system. This is because the second slowest path requires much less processing power to execute in the same time as the critical path and the removal of the first few processors does not therefore alter this critical path. This will be discussed and graphically illustrated in the following section, when the eventuality of reduced processor availability is considered.

6.6.3 Introduction of Hardware Constraints

A natural extension to the analysis of a system under ideal conditions is the consideration of the system performance under less favourable circumstances and possibly the establishment of thresholds below, or above, which the system performance indices remain constant. In this particular case the optimum performance of the controller is achieved in the presence of six Transputers. What now needs to be considered is the nature of the degradation in performance resulting from a reduction in Transputer numbers: how does the speedup decrease, and how does processor efficiency behave as a function of processor numbers?

The following section attempts to graphically answer these questions and furthermore it gives explanations as to why the performance indices behave in the manner they do.

6.6.3.1 Effects on speedup and processor efficiency

A reduction of one in the number of processors effectively means that the work previously executed by this processor must now be shared among the remaining processors. The manner in which this work is shared or allocated is vitally important. An ad-hoc method of dividing the work-load among the processors would be a futile scheme and a travesty of software management. The specific application generally determines the way in which the extra work is consumed and it would be very difficult to programme a multi-purpose compiler to optimise the work-load and speedup simultaneously, without introducing considerable dynamic management overheads. The scheme employed here involves maintaining the modularity of the processes and re-assigning complete processes to single processors. The diagrams showing the parallelism of the controller functions, contained in Sections 6.1.3, 6.2.2 and 6.2.3, clearly illustrate the processes by the use of blocks, and these processes are treated as indivisible tasks. Removal of a processor creates extra processes, available for execution, and the most important concern during re-allocation of these processes is the preservation of the critical path at its present length. Provided the critical path remains constant the process allocation is arbitrary. During the following analysis the PD controller execution time is defined as its parallel run-time on the T414-20 Transputer (32.0µs for each PD controller), for the purposes of illustration (in the absence of timing measurements on the PACE array).

In the optimal hardware configuration six Transputers are assigned to the controller. Three are assigned to the feedforward inverse dynamical equations calculation, and three are devoted to the feedback path, which involve a recursive least squares routine in sequence with a PD controller. By removing one Transputer a five Tranpsuter multiprocessing system is left to implement the controller. Despite this the speedup remains unchanged, although processor efficiency increases. Three Transputer are still dedicated to the critical path of the controller, or the computed torque evaluation, but only two Transputers execute the feedback path. This reduces the maximum parallelism of both the parallel recursive least squares algorithm and the three-joint PD controller to two. By inspection this increases the execution time of both algorithms by 688.6µs and 320.3µs respectively (due to the reassignment to another processor of one of the three least squares routines and one of the three PD controllers). The feedback computational path is increased from 1.0692ms to 2.0781ms, which implies that the critical path remains unaffected by the one-processor reduction.

Removal of a second Transputer, to leave a four Transputer system, is the next stage of analysis. Previous to this removal the feedback path has two Transputers assigned to it whilst the feedforward path has three dedicated Transputers. If a further

Tranpsuter is removed from the feedback path the one remaining Transputer has an additional workload which adds 1.009ms to the computational time for the feedback path, equal to its sequential execution time of 3.087ms. Thus this path becomes the critical path and reduces speedup to a value of 3.01 and processor efficiency becomes 0.752.

It is possible however to avoid such a severe degradation in speedup, when one considers the fact that of the three Transputers assigned to the feedforward path one or two are often idle, awaiting the completion longer path. This is apparent from Figure 32, Section 6.1.3, which shows the computed torque algorithm in parallel form. It is possible to assign the idle Transputers to execute complete processes from the feedback path, with the minimal effect on the critical path length. By evaluating several combinations of process allocation the optimum scheme was determined. After approximately 790μs the three Transputers in the forward path begin to execute the functions define-d-bar(), define-p1bar() and define-p2bar() in parallel. About 100μs previously the one Transputer in the feedback path would have completed executing the second of the three recursive least squares routines, which take 688μs each. The define-d-bar(), define-p1bar() and define-p2bar() function have execution times of 315.6μs, 1.76ms and 1.815ms respectively, so the Transputer hosting the define-d-bar() function on completion idles for 1.499ms, which is sufficient time to execute the two least squares routines, allowing for scheduling overheads. If this processor therefore is assigned these two functions then the uni-Transputer in the feedback path has its processing time reduced by approximately 1.377ms and the critical path remains unaltered. Speedup also remains unchanged from the original value achieved with optimal processor assignment and processor efficiency is increased

Theeventuality of only three Transputers being available creates a more complicated scenario. In the four Transputer system the Transputer in the feedback path hosts one least squares routine and three PD controllers. The solution in finding the optimal process assignment is again achieved by evaluating different allocation schemes and selecting that which yields minimum critical path length. It is found that the most effective method is to remove the Transputer from the feedback path and integrate the feedback calculations with those of the feedforward path on the same three Transputers with an amended scheduling scheme. The appropriate assignment criteria in the case of three Transputers is restricted in that the three individual PD controllers must be executed sequentially with their respective least squares routine (i.e. the identification routine must be complete before the PD controller can begin to generate a compensation signal). Note that in the computed torque evaluation the final three functions, var-assign(), mat-by-vec() and vector-add() are executed sequentially and two Transputers are idle. The optimal assignment is to use the two idle Transputers to host the extra processes. One Transputer executes the final least squares routine and a PD controller while the other hosts the remaining PD controller. The reason why one of these two PD controllers is placed on the same Transputer as the identification routine is to avoid the necessity of channels. Since it must await the final least squares routine to complete before beginning execution channels communicating the data to the PD controller would be necessary were it on a separate TRAM. The extended critical path takes 3.422ms to complete. Speedup takes on a new degraded value of 2.7136 and processor efficiency is a respectable 0.904, which is to be expected considering the reduced processing power.

Only two more Transputer topologies remain to be analysed: a uni-Transputer system and a dual-Transputer system. The case of only one Transputer is the benchmark standard used to evaluate all other cases and has a speedup of unity and a processor efficiency also of unity. The performance indices for the case of a dual-Transputer system requires a little more thought. The availability of two Transputers to implement the controller creates the possibility of many different processor allocation schemes, but as before that which minimises the critical path is the only one of practical interest. The initial four sequential functions of the computed torque algorithm, see Figure 32, are executed on one transputer as before. The following three parallel processes, which evaluate the joint centripetal and coriolis forces, have the same execution time on two Transputers as three, since the two fastest processes combined still execute quicker than the slowest process. On completion of the slowest process the outputs are collated and the inertial matrix is calculated as before. Following this again there are three parallel processes to be executed, and once more the slowest process takes longer to execute than the other two processes. The controller up until this stage has preserved its value of speedup. The remaining processes to be executed have a major impact on the final value of speedup, however. The following three parallel computational paths, (a) define-d-bar() and the two identification routines, (b) define-p1bar() and (c) define-p2bar() must be implemented on two processors. The summation of the two fastest paths gives an execution time of 3.458ms. The final operations, which again are distributed among three parallel computational paths, must also be assigned in the most efficient manner to two processors. Fortunately, in this instance the two fastest paths combined are computationally shorter than the slowest path, so no extra time is added to the overall controller execution time. Combining the above observations and timing measurements the controller execution time is modified to a new time of 5.254ms. This execution time yields a moderate value for speedup of 1.77 and consequently processor efficiency for the dual-Transputer system becomes 0.884, a decrease on the previous case, (three Transputers), because the process modularity prevented processes being decomposed equally among the two Transputers, and in two instances the longer of the two parallel computational paths was considerably slower than the other faster path and was reflected by an increase in processor idling.

This data is best expressed through the medium of a graph, and in Appendix C Graph 6.8 contains the piece-wise linear curve representing the performance of speedup as a function of the number of TRAMs available. The increase in speed from one to four processors is practically linear, and remains constant having reached its maximum at a value of 3.614. Likewise Graph 6.9 charters the progress of processor efficiency as the number of available TRAMS varies. A significant drop in processor efficiency appears at the two-processor stage. This reflects the fact that the increase in speedup from one to two processors is less than the theoretical linear increase from one to four processors. Note that at this value the efficiency curve decreases in a more acute manner than the speedup curve because the scales are different (the speedup scale spans a wider range). Based on the timing measurements presented in this section Graph 6.10 demonstrates the manner in which the overall controller execution time behaves as the number of TRAMs is increased from one to twelve. It must be mentioned that because the number of TRAMs may only increase in a discrete integer fashion the graphical data is correctly expressed as a series of points, and that these points are extended to a curve in the above graphs for the purposes of analysis and to illustrate the trend more clearly. The final section terminates this chapter by summarising the various sections and briefly concluding the overall results of the parallelised computed torque feedforward/PD compensator feedback controller. A more comprhehensive conclusion is made in Chapter 7, which is the concluding chapter of the thesis.

6.7 SUMMARY

Itisno understatement to say that the field of parallel processing holds tremendous possibilities for development of new, and the modification of existing, algorithms in the study of control engineering. In this chapter a scheme is proposed to utilise a parallel language software system and a multi-Transputer plus PACE array hardware system to exploit the inherent concurrency of operations in a robotic control strategy. In order to determine the coarse and medium grain parallelism in the overall controller the controller itself must be decomposed into well defined independent functions. These functions are examined and coded in a manner which explicitly declares the concurrency at an operational level. Furthermore, at a more coarse grained level, it is also possible to declare the concurrency between the functions themselves.

The overall controller may be divided into two independent paths: feedforward and feedback. The feedforward path consists of a computed torque algorithm. The

PUMA 560 robotic manipulator is adequately modeled by a set of third order dynamical equations, with joint motor voltage as input and the derivative of joint accelerations as output. By incorporating the relationship between joint position and velocity, joint velocity and acceleration and joint acceleration and its derivative a ninth order state-space model is achieved. In the computed torque algorithm the inverse dynamics of the robotic are modeled, as per equation 53c, Section 6.1.1. Whilst the forward dynamics model the response of the joints to an input voltage the inverse dynamics of a robot define the voltages necessary to achieve the desired joint loci. Therefore, when supplied with the desired joint set-points the computed torque algorithm attempts to generate the voltages necessary to accomplish these positions within one sample period. In Section 6.1 this algorithm is described and based on the inverse dynamical model the presence of coarse grained parallelism is identified.

The second computational path of the controller is the feedback path. The major drawback of using a computed torque controller is inability of the controller to respond to errors in joint positioning, in the absence of feedback. Any mispositioning due to modelling inaccuracies accumulates as time progresses and may become significant. By incorporating a feedback path a compensation voltage may be generated to offset the error. In Section 6.2 a PD feedback controller, which is driven by the error in the joint positions, is introduced. In order for the PD controller to generate accurate control signals the joints are represented by a continuously updated linear model generated by a recursive least squares identification routine. The presence of considerable medium grained parallelism in the PD algorithm is presented in Section 6.2.3, and the coarse grained parallelism illustrated in Section 6.2.2 is based on the observation that three joints each have a separate PD controller dedicated to them. The linear model identification routine may also be decomposed in a coarse grained manner, and this is shown in Section 6.2.2.

The software coding of the parallelised controller is given in Section 6.4, using the combined feedforward feedback controller model developed in Section 6.3. Parallel C is used to explicitly declare operational concurrency, and is designed specifically for compilation into Transputer machine code. The following Section describes the host hardware for the controller, which is proposed to incorporate the PACE programmable array. The PACE architecture is an ideal target system for the medium grained decomposition of the PD controller, and a standard PID control algorithm application on the array is modified to accommodate the less complex PD controller, which is described in Section 6.5.2.2. To avail of the PACE architecture one must devise a scheme to interface the Transputer and PACE topologies to harmonise their operations. By assigning chip control to a Transputer an interface methodology is presented in Section 6.5.3.

The final section deals with the analysis of the controller performance in parallel form, and considers the effects of reduced processor numbers. Speedup and processor efficiency are the two most prominent indices when considering the effect of a varying hardware environment. When the number of processors is at or below the optimum quantity then the maximum parallelism will always be equal to this number of processors. Furthermore, the average bounded parallelism for a given number of processors, as described in Chapter 3, is equal to the speedup at that same number of processors. The numerical findings of Section 6.6 are presented in their most eloquent form in Appendix C, which contains the graphical data.

CHAPTER 7

Conclusions

A comprehensive introduction to parallel processing is given in Chapter 2, and the general hardware and software manifestations of parallel processing theory are also included. The technique of concurrent processing provides a powerful platform for enhanced algorithmic performance and it is on this premise that an attempt is made to apply it to aspects of control engineering. To evaluate the effectiveness of the application a parallel decomposition scheme to an algorithm one must possess a thorough understanding of the formulation and interrelationships of relevant computational performance indices. In Chapter 3 the parallel performance indices of speedup, processor efficiency, parallelism, maximum parallelism and bounded parallelism are extensively considered and formulated and their relationship with the software construction and hardware topology is examined. The limitations and bounds of these performance *benchmarks* are sought and graphically illustrated.

Within an algorithm or a specified sequence of calculations there may be considerable scope for operational concurrency. In some instances, however, sequentiality may be ingrained in the nature of the computations and attempting to impose parallelism on them proves to be futile. There is a technique whereby concurrency may be created in the operations, by use of the *pipelining* technique. This is described in detail in Chapter 2, but in Chapter 4 the application to the field of control engineering is aptly illustrated by the pipelining of an explicit self tuning regulator (STR). The STR is part of the family of control strategies known as adaptive controllers. The STR controller generally decomposes into three tasks: identification of the controlled process with a linear model, controller parameter evaluation and control signal generation. It performs these three tasks every sample period. The identification routine is completely sequential, the parameter evaluation likewise and the control signal generation is a short and also sequential task. At a coarse grained level the three processes are highly sequential and offer no scope for parallelisation. By the use of pipelining the overall speed of the algorithm may be greatly enhanced at the cost of increased initial transient error and extra processing power, as described in Chapter 4. Graph 4.1 to Graph 4.6 illustrate the convergence of the controller parameters for a five stage pipeline, and vindicate its operational stability. Graph 4.7 shows both the controller and the system response to a desired unit step output. Some transient errors on the system output signal are evident but prove to be insignificant and the error is found to be less than 0.5% after five sample periods. The performance of the five-stage pipelined STR as a function of controller system order is chartered in Graph 4.8, Appendix C, page C1. It is found that the execution time increases in an exponential fashion as the system order increases, with a time constant approximately equal to 2.16.

Alternatively, in the instance where considerable scope does exist for parallelisation of an algorithm the benefits in enhanced performance can be quite significant.One of the most computationally intensive tasks in practical robot engineering is the software implementation of a mathematical model of the manipulator dynamics. In Chapter 5 an attempt is made to create a parallel simulator for the Puma 560 robot manipulator which is faster than the sequential model, but demonstrates the same degree of modeling integrity. The third order model presented in Chapter 5 is based on a second order model derived by Paul[70]. The second order model neglects some important dynamics of the Puma 560 arm and is therefore a poor model for use in a simulator when validating and testing a controller technique, especially one which operates within restricted error margins.

The open-loop performance of the model is tested by using a suite of different tests, which examine for various dynamics and responses, so that the simulator output may be compared with the actual manipulator response. From the graphical results, analysed in Section 5.5.1, it can be concluded that the dominant characteristics of the robot dynamics are adequately described by a third order dynamical model.

The mathematical model of the manipulator is examined for parallelism. The maximum parallelism was found to be 3 and the average parallelism was found to be equal to 2.438, which is equivalent to the ideal speedup in the instance of unlimited availabilty of processors, as shown in Chapter 3. It is interesting to note that the maximum parallelism in the model does not exceed the number of joints being simulated. This is to be expected since the computational model of each individual joint is quite sequential in nature and there is generally three computational paths in parallel throughout the simulator. There are some coupling terms which must be calculated for each of the three joints in sequence, which reduces the average parallelism for the joint calculations below a value of three. The remaining calculations and assignments (matrix calculation and set up, variable assignment, etc.) would require a average parallelism of much greater than three in order to give the system an overall parallelism of three or greater. These additional overheads are found to be mostly sequential, or at best have a maximum parallelism of three. The object therefore is to attempt to attain as near a speedup as possible to the ideal value. The factors which degrade actual speedup are mostly inter-processor communication and process partitioning and scheduling overheads. Although it is important that processor efficiency is as close to unity as possible to ensure a cost effective multiprocessor system the most important factor is obviously speedup, since this is the objective of parallelisation. As mentioned in Section 5.5, Jones & Fleming [93] found that through using a processor farm to share the workload and thus increase processor efficiency, the scheduling overheads became significant and in some instances took more processing time than the actual computations. In Chapter 5 the parallel model of the simulator is ideally implemented in parallel on three Transputers, which means that each concurrent process could be run on a separate processor. This ensured reduced scheduling overheads. Furthermore, the communications between processes was kept to a minimum to avoid unnecessary idling of processors.

The execution time of the simulator on a T414-20 Transputer network as described above is determined against processor numbers and is displayed in Graph 5.30. On an optimal system of three Transputers the simulator takes 15.256 milliseconds per iteration, compared to 37.194 milliseconds on a single Transputer. Increasing the number of Transputers beyond three has no effect on the performance of the simulator, other than enhanced integrity due to the availability of extra Transputers in the event of the *failure* of one the active Transputers. This is evident in Graph 5.25, which illustrates the performance of speedup as a function of the number of processors.

The introduction of pipelining enhances the execution time of the simulator even further. The iterative loop of the simulator naturally decomposes into four processing stages, with each containing a call to the function high-states() and a routine to evaluate the Runge Kutta numerical integration coefficients. In the unpipelined case the parallel model is optimally implemented by three processors. In the pipelined case, therefore, the optimal number of processors will be three for each of the four stages, or twelve overall. This is quite a costly hardware realisation but proves to be a relatively fast implementation of the forward dynamics of the Puma 560 robot manipulator. The execution times of the simulator for different numbers of T414-20 Transputers is illustrated in graph 5.31. Note however that the addition of extra processors is only permitted in blocks of four and a smooth curve is fitted to the points so that the natural trend of decreasing execution times is best illustrated. The fastest execution time, which occurs when twelve Transputers are used, is 3.8784 milliseconds. This is a speedup of almost 9.6 over the sequential case. The compares to an execution time of 17.9091 milliseconds for the pipeline of twelve INTEL 80286 μProcessors. The pipelining technique introduces errors due to data uncertainty in the initial stages of processing. After 5 loop iterations this percentage error reduces to an average of just over 1.5%, and on completion of ten iterations this error is no more than 0.2%. For most applications of the simulator, (controller validation, manipulator response, etc.), this initial margin of transient error is acceptable. This is graphically illustrated in Graph 5.31. There are many instances however, where the initial response is of vital importance, sometimes more so than the long term steady state response (e.g. impulse response of joints) and in these situations the pipelined simulator may prove to be inadequate.

In conclusion Chapter 5 has shown how the forward dynamics of the Puma 560 robot manipulator may be modeled adequately by third order differential equations. These equations may be transformed to matrix form and simulated in software. The software model, and not simply the equations themselves, may be examined for parallelism. By calculating the average parallelism of the model one also calculates the theoretical speedup of the parallel realisation over the sequential implementation. By optimising the hardware to the parallelised software model the speedup is also optimised. By increasing the number of processors above an optimal value the processor efficiency is decreased but speedup remains unchanged. The introduction of pipelining offers further scope for enhanced performance, but at the cost of extra hardware and decreased data accuracy in the initial stages. This can be acceptable in some applications where the initial data is not vitally important. Overall, though, it is seen how once again some of the concepts from the exciting area of *parallel processing* can be exploited to the betterment of performance indices in the field of control engineering.

In the same spirit as Chapter 5 Chapter 6 presents a parallelisation strategy for a PUMA 560 robotic manipulator controller. The controller is defined as having two paths: a feedforward path and a feedback path. The computationally more intensive feedforward path contains a computed torque algorithm, which is based on the inverse dynamical model of the manipulator. Ideally this path generates the necessary control signals to achieve the desired joint set-points, (supplied from a path planner routine), but in reality errors in joint positioning are unavoidable. Factors which contribute to the errors include inaccurate joint coefficients in the inverse dynamical model, joint stiffness and joint motor modeling errors, and to a lesser extent joint elasticity and gearing backlash. Without compensation these errors accumulate as time progresses, which may result in large final errors. A feedback path consisting of a linear model identification routine and a PD controller, driven by the joint positional error, is incorporated to generate the necessary compensatory voltages to offset the error. One of the reasons a PD control strategy is used is to avoid overshoot in the system output, which is desirable in this instance when you consider that the system output is the vector of joint positions. Graph 6.7, Appendix C, page C.12, illustrates the behaviour of the error over a given trajectory. By observation it can be seen that the error is proportional to the joint velocity (see Graph 6.3, page C.10) with a constant of proportionality equal to $-5.166\text{x}10^3$. Thus it is necessary to optimise the velocity of the joints under an error and time cost function. If the error is to be kept within a certain margin, then it is possible to incorporate a differential constraint into the path planning methodology used to generate the desired trajectory. This would prove to be an interesting optimisation problem, but unfortunately it is not within the scope of this thesis to postulate a solution.

The maximum parallelism in the computed torque algorithm is found to be equal to the number of joints being controlled, three. By further inspection the average parallelism of the algorithm is determined to be 2.374, which in the eventuality of an optimal number of processors being available yields a processor efficiency of 0.791. The speedup is quite respectable and represents almost eighty percent of the figure for maximum parallelism (recall that the value speedup may not exceed the figure for maximum parallelism). The values of speedup and efficiency for the Puma 560 simulator, which is based on the forward dynamical equations, are 2.438 and 0.8127 respectively, which are comparable to the values for the computed torque algorithm, which is based on the inverse dynamical model.

The feedback path is also examined for computational parallelism. Within this path there is scope for both medium grained and coarse grained parallel decomposition. Overall each joint is assigned an individual PD feedback controller, and each joint is separately identified by a recursive least squares routine. The sequential operation of identification and PD control may be performed on each of the three joints in parallel. This represents the coarse grained decomposition of the feedback path. At a finer grained level there is considerable scope for medium grained parallelisation of each of the three PD algorithms. Based on a PACE array PID algorithm realisation by Jones[81] the PD controller evaluations are mapped onto a medium grained array processor. This method exploits both parallelism and pipelinability.

The major obstacle of combining processors of two different granularity is the necessity to define and create an operational interface regime, to facilitate efficient inter-processor communication. In Chapter 6 such an interface definition is given for the PACE and Transputer topologies. Due to general unavailability of the PACE chip it is not possible to specifically code or build the interface. For the purposes of simulation and in the absence of a PACE array the PD algorithm is hosted on a TRAM on the IMS B008 motherboard.

The coarse grained decomposition of the feedback path yields a maximum parallelism of three, for the same reasons as explained for the forward path, and the execution time is found to be much less than the execution time of the feedforward path. The feedforward computational path is therefore defined as the critical path for the overall parallelised controller. Furthermore the maximum parallelism of the complete controller is found to be six, which occurs when the maximum number of processes are active simultaneously for both the feedforward and feedback paths. Naturally, six processors constitute the optimal hardware topology. It is interesting to observe the performance of speedup, equal to the bounded parallelism, for lesser numbers of processors, shown in graphical form in Appendix C, Graph 6.8. For four and five TRAM systems the speedup remains unaltered, and the degradation normally associated with reduced processing power is absent. This is explained by the fact that in the controller there are two computational paths (feedforward and feedback) which are completely independent, and whilst both have a maximum parallelism of three both paths have substantial sequential calculations. During the sequential calculations of, say, the feedforward path one or two processors may be idle, and it is possible to reassign these processors to processes in the feedback path, in the case where the feedback path has a number of processors less than the optimum. In the instance of three or less processors being assigned to the controller speedup degrades rapidly to unity. The performance of processor efficiency as a function of the number of processors may be inferred from the speedup curve. Graph 6.9 illustrates this performance index. Processor efficiency increses when the number of processors is increased from two to three. This reflect the fact that the slope of the speedup curve from one processor to two processors is less than over the range two processors to three processors. Although this slope difference is slight in Graph 6.8 it is amplified by the shorter y-axis scale of Graph 6.9 to give a much more dramatic change in the processor efficiency curve.

REFERENCES

[1] Lea, R. M. "The influence of technology on parallelism " in "Major Advances in Parallel Processing", Editor C. Jesshope, Technical Press 1987, pp.3-13

[2] Lea, R. M. "The influence of technology on parallelism " in "Major Advances in Parallel Processing", Editor C. Jesshope, Technical Press 1987, pp.3-5

[3] Jones, Dr. S "Parallel processing in control ", Editor P.J. Fleming, IEE Control Engineering Series 38, Publ. Peter Peregrinus Ltd 1988, page 5.

[4] Kinniment, D. J. "VLSI & Machine Architecture", Computer Science Dept., Technical report, University of Newcastle upon Tyne, UK, 1985.

[5] Jones, Dr. S. "Parallel processing in control ", Editor P.J. Fleming, IEE Control Engineering Series 38, Publ. Peter Peregrinus Ltd 1988, page 3.

[6] Jones, Dr. S. "Parallel processing in control ", Editor P.J. Fleming, IEE Control Engineering Series 38, Publ. Peter Peregrinus Ltd 1988, page 4.

[7] Lea, R. M. "The influence of technology on parallelism " in "Major Advances in Parallel Processing", Editor C. Jesshope, Technical Press 1987, pp. 8-10.

[8] DeCegama, A. L. "The Technology of Parallel Processing" Prentice Hall, Chap. 1.

[9] Sweazey, P. & Smith, A. J. "A class of compatible cache consistency protocols & their support by the IEEE Futurebus" 1986 IEEE conference on Computer Architectures, 1987, pp. 414-423

[10] Flynn, M. J. "Very high speed computing systems" Proceedings of IEEE, Dec. 1966, pp.1901-1909.

[11] Kuck, D. J. "The structure of computers and computations", Volume 1, New York Wiley, 1978.

[12] Basu, A. "A classification of parallel processing systems", Proceedings of IEE conference on computer design, 1984, pp.222-225.

[13] DeCegama, A. L. "The Technology of Parallel Processing" Prentice Hall, page 63.

[14] Keller, R. M. et al. "Simulated performance of a reduction based multiprocessor" IEEE Computer, July 1984, pp.70-82.

[15] Jones, Dr. S. "Parallel processing in control ", Editor P.J. Fleming, IEE Control Engineering Series 38, Publ. Peter Peregrinus Ltd 1988, p.5.

[16] Mc Cabe, A. "Systolic arrays and special purpose silicon ". in "Major advances in parallel processing" Editor C. Jesshope, Technical Press 1987, pp.13-33.

[17] Hudson, J. "Systolic array architectures for optimal filtering", IEE Computing and Control division conf. "Recent advances in Parallel Processing for Control", Bangor, Wales, 7th July 1988.

[18] Megson, G. M. "Transputer implementation of systolic arrays for model reduction", IEE Computing and Control division conf. "Recent advances in Parallel Processing for Control", Bangor, Wales, 7th July 1988.

[19] Irwin, G. "Occam simulation of a systolic architecture for parallel kalman filtering", IEE Computing and Control division conf., "Parallel Processing in Control - The Transputer and other architectures", 20-22 Sept. 1987.

[20] Gaston, F. Irwin, G. "The application of systolic and wavefront arrays to adaptive beamforming", IEE Computer and Control division conf., "Parallel Processing: A new direction for control", London, Feb. 1987.

[21] Gaston, F. "Systolic kalman filtering - An overview", Dept. of elect. engineering, The Queens University of Belfast, Northern Ireland, United Kingdom.

[22] Gaston, F. Irwin,G "VLSI architectures for square root covariance kalman filtering", Proc. SPIE conf., San Diego, Aug. 1989.

[23] Irwin, G. Gaston, F. "Systolic square root covariance kalman filtering" Dept. of elect. engineering, The Queens University of Belfast, Northern Ireland, United Kingdom.

[24] Irwin, G. Gaston, F. "Linear quadratic optimal controller realisation on systolic arrays" Dept. of elect. engineering, The Queens University of Belfast, Northern Ireland, United Kingdom.

[25] Irwin, G. Gaston, F "A systolic parameter estimator for real time control" Dept. of

elect. engineering, The Queens University of Belfast, Northern Ireland, United Kingdom.

[26] Kung, R. "A new approach to linear filtering and prediction problems" Trans. ASME, J. basic Eng. , 1960, 82D, pp34-45.

[27] Ward, C. "A novel algorithm and architecture for adaptive digital beamforming" IEEE Trans. Antennas and Propagation, 1986, AP-34, pp338-346.

[28] Kung, H. "Systolic communication" Proc. Int. Conf. on systolic arrays, May 1988, pp695-703.

[29] Guez, A. "Neuromorphic architecture for adaptive robot control : A preliminary analysis" 1987, Interml Conf. of Neur. Net. (ICNN).

[30] Guez, A. "Solution to the inverse kinetics problem in robotics by neural networks" March, 1988, International Conf. on Neural Networks, (ICNN)

[31] Hopfield, J. "Computing with Neural Networks - A model ", SCIENCE vol.233, 8 Aug. 1986, pp.625-633.

[32] Murtagh, F. & Adorf, H "Clustering Based on neural network processing", COMPSTAT '88, Copenhagen, Aug.-Sept. 1988.

[33] Materna, T. "Neural networks enter high speed marketplace " Computer Technology, Vol (vii)., no.7, June 1987.

[34] Zeidenberg "Modelling the brain", BYTE, December 1987, pp. 237-246.

[35] McKittrick, B. J. "Development and implementation of a correlation based stereo algorithm", School of Electronic Engineering, Dublin City University, 1989

[36] Hillis, W. D. "The Connection Machine" Cambridge, MA, The MIT press, 1985.

[37] Siegel, H. J. et al. "The PASM parallel system prototype", IEEE 1985 COMPCON, pp.429-434.

[38] Mc Kittrick, B. J. "Parallel Processing and Control: A Survey", Technical report, Control Technology Research Unit (CTRU), Dublin City University, March, 1990.

[39] Young, S. "An introduction to ADA" 2nd Edition.

[40] Griffiths, S. "Introduction to OCCAM Development", IEE Computing and Control division conf., "Parallel Processing in Control - The Transputer and other architectures", 20-22 Sept. 1987.

[41] Ahuja et al. "Linda and friends", IEEE Computer, August 1986, pp.26-34.

[42] James McGraw et al. "SISAL: Streams and Iteration in a Single Assignment Language", Language reference manual, Version 1.2, University of Texas, Computer Science Dept., March 1, 1985.

[43] Bigler, B. M. et al "Parallel dynamic storage allocation" 1985, IEEE Int. conf. on par. proc. pp.272-275

[44] Fox, G. C. & Otto, S. W. "Concurrent computation and the the theory of complex system" Knoxville Hypercube conf Aug '85

[45] Fox, G. C. "Load balancing and sparse matrix vector multiplication on the Hypercube" Caltech preprint C^3, page 327.

[46] DeCegama, A. L. "The Technology of Parallel Processing" Volume 1, Prentice Hall, New Jersey, 1990, page 44.

[47] Irwin, G. Gaston, F. "Systolic square root covariance kalman filtering" IT90, San Diego, March 1990.

[48] Ahmed, S. "Optimal design of multiple arithmetic-based robot controllers", Purdue University.

[49] Daniel, R. W. Sharkey P. M. "Transputer control of of a PUMA 560 robot via the virtual bus" IEE proc., Vol 137, Pt.D, No.4, July 1990.

[50] Katbab, A. "A multiprocessor architecture for robot arm control" Microprocessing & Microprogramming 24 (1988) 673-680, North Holland.

[51] "A multiprocessor-based controller for the control of mechanical manipulators" IEEE Journal of robotics and automation, Vol. RA-1, No.4, December 1985.

[52] Prof J. W. Ponton, McKinnel, R. "Nonlinear process simulation and control using

Transputers", IEE proc., Vol. 137, Pt.D, No.4, July, 1990.

[53] Skogestad, S. Morari, M. "The dominant time constant for distillation columns" Computing and Chem. Eng., 1987, 11, (6), pp.607-618.

[54] Sargent, R. W. H. Sullivan, G. R. "Development of feed changeover policies for refinery distillation units", I & EC, pre d&d, 1979, 18, (1), pp.113-124.

[55] Eager, D. L. Zahorjan, J. Lazowska, E. D. "Speedup versus efficiency in parallel systems" IEEE Trans on comp., Vol. 38, No.3, March 1989.

[56] Young, S. "An introduction to ADA" 2nd Edition.

[57] DeCegama, Angel L. "The technology of parallel processing" Volume 1, Prentice Hall, New Jersey, 1990.

[58] Minsky, M. Papert, S. "On some associative, parallel and analog computations" in Associative Information Technologies, E J Jacks, Editor, New York, Elsevier, North Holland, 1971.

[59] Kluck, D. J. et al "The effects of programme restructuring, algorithm change and architecture choice on programme parallelism" Proc. Int. Conf. on Par. Proc., 1984, pp.129-138.

[60] Heidelberger, P. Trivedi, K. S. "Queueing network models for parallel processing with asynchronous tasks" IEEE Trans. Comput., Vol C-31, No.11, 1982, pp.1099.

[61] Fayolle, G. King, P. J. B. Mitrani, I. "On the execution of programs by many processors" in Proc. 9th Int. Symp. Comput. Performance modeling, Measurement Evaluation, 1983, pp.217-228.

[62] Chen, T. C. "Overlap and pipeline processing" in Introduction to Computer Architecture, H stone, Editor, SRA, 1975, pp.375-431.

[63] Amdahl, G. M. "Validity of the single processor approach to achieving large-scale computing capabilities", Proc. AFIPS, Vol 30, 1967, pp.483-485.

[64] Denning, P. J. Buzen, J. P. "The operational analysis of queueing networks models" Comput. Survey, Vol 10, No. 3, pp.225-261.

[65] Muntz, R. R. Wong, J. W. "Asymptotic properties of closed queueing networks models" in Proc. of 8th Princeton conf. inform. sci. syst., 1974, pp.348-352.

[66] Kumar, B. Gonslaves, A. "Modelling & Analysis of distributed software systems" Proc. 7th ACM symp. oper. syst. principles, 1979, pp.2-8.

[67] Graham, S. L. Kessler, P. B. McKusick, M. K. "gprof: a call Graph execution PROFiler" Proc. of the ACM SIGPLAN '82 Symp. on Compiler Construction, June, 1982, pp.120-126.

[68] Anstrom, Wittenmark "Computer controlled systems" Prentice Hall Information and System Sciences series, 1984.

[69] DeCegama, Angel L. "The technology of parallel processing" Prentice Hall, New Jersey, 1990, page 9.

[70] Paul, R. "Robot manipulator mathematics, programming control" The MIT press, 1981

[71] Craig, J. "Introduction to robotics, mechanics and control" Addison-Wesley Publishing Company.

[72] Denivit, J. & Hartenberg, R. "A Kinematic notation for lower-pair mechanisms based on matrices" ASME Journal of applied mechanics, June, 1985.

[73] Leahy et al "Efficient dynamics for the Puma 600" Robotics and Automation, San Francisco, CA., 1986.

[74] Stone, H. et al "Dynamic modeling of a three DOF robotic manipulator" IEEE Transactions on System Man and Cybernetics, Vol. SMC-14, No.4, July/Aug., 1984.

[75] Bejczy et al "Nonlinear feedback control of a Puma 560 robot arm by computer" Proc. of 24th confer. on decision and cont., Dec., 1985.

[76] Goldstein, H. "Classical mechanics", 2nd edition. Addison-Wesley Publishing Company.

[77] Carr, S. Anderson, G. et al "An LQG approach to selftuning with applications to robotics" Robotic control: Theory and applications, Peter Pergrinus Ltd. for IEE, 1988.

[78] Tang, P. Yew, C. Fang, Z. Zhu, C. Q. "Deadlock prevention in processor self-scheduling for parallel nested loops" University of Illinois, CSRD report no.626,

Jan., 1987.

[79] Turbo C, User Guide, Borland International.

[80] Turbo C, Reference Guide, Borland International.

[81] Jones, S. Spray, A. "Implementing irregularly structured functions On regularly structured arrays" Dept. of Elect. & Electronic Eng., University of Nottingham, University Park, Nottingham, NG7 2RD, United Kingdom.

[82] Jones, S. Spray, A. "PACE: A regular array for implementing irregularly structured algorithms" Dept. of Elect. & Electronic Eng., University of Nottingham, University Park, Nottingham, NG7 2RD, United Kingdom.

[83] Jones, S. Spray, A. "PACE architecture & Kalman filtering" Dept. of Elect. & Electronic Eng., University of Nottingham, University Park, Nottingham, NG7 2RD, United Kingdom.

[84] Poplett, J. "The development of a parallel C", TCC-0910-1 to TCC-0910-7.

[85] "Parallel C", Publ. by 3L limited, Supplier: SENSION - Transputer products division, Denton Drive, Norwich, Chesire CW9 7LU.

[86] Taylor, R. "Survey of Transputer Applications". May 1986.

[87] May, D. "The Transputer", "Major Advances in Parallel Processing", Editor C. Jesshope, Technical Press 1987, pp. 33-47.

[88] May, D. & Shepherd, R. "The Transputer Implementation of OCCAM" in "Parallel Processing in Control" Editor P.J. Fleming, IEE Control Engineering Series 38, Publ. Peter Peregrinus Ltd 1988, p.85.

[89] Fleming, P.J. "OCCAM and the Transputer" in "Parallel Processing in Control", Editor P.J. Fleming, IEE Control Engineering Series 38, Publ. Peter Peregrinus Ltd 1988, pp.27-99.

[90] Daniel, R. W. Sharkey, P. M. "Tranpsuter control of a Puma 560 robot via the virtual bus", IEE Proc., Vol.137, Pt. D, No.4, July, 1990.

[91] Ahmad et al "Optimal design of multiple arithmetic processor-based controllers" School

of Electronic Engineering, Purdue University, West Lafayette, Indiana 47906, USA.

[92] Lee, C. S. G. Chang, P. R. "Efficient parallel algorithms for robot forward dynamics computation" School of Electronic Engineering, Purdue University, West Lafayette, Indiana 47906, USA.

[93] Jones, D. I. Fleming, P. J. "Inverse Dynamics equations for Robot Control" in Parallel Processing and Control, Editor P J Fleming, IEE Control Eng. Series 38, 1988, pp.117-125.

[94] "IMS B008 User guide & reference manual" Published by INMOS, Product support.

[95] "Module Motherboard Software - User guide" Published by INMOS, Product support

[96] INTEL 8237 DMA controller chip, IBM PC-AT, Data Sheet.

[97] Ogata, K. "Discrete time control systems" Prentice Hall International editions, 1987, Chapter 7.

[98] Snyder, L. "Introduction to the configurable highly parallel computer" IEEE comp. mag., January, 1982, pp 47-56.

[99] PACE software simulator, available on request from:
Dept. of Elect. & Electronic Eng., University of Nottingham, University Park, Nottingham, NG7 2RD, United Kingdom.

APPPENDIX A

Parallel Processors - An Evaluation

A.1 THE TRANSPUTER

Since it's development by INMOS in early 1986 the transputer [A1][A2][A3] has become one of the more widely used processors in parallel system realisation. In VLSI technology it is observed that communication between seperate devices is much slower than communications within a single device. Thus a processor can spend a lot of time accessing it's memory store. Based on this premise the transputer was designed with both processor and memory on the same integrated circuit.

Communication between transputers is executed in serial fashion using point to point connections. A transputer system consists of a number of transputers connected in some ordered fashion. Each transputer can be directly connected to a maximum of 4 neighbouring transputers. A notable feature of the transputer is that it can support concurrent processes internally, albeit by time sharing the processor between the processes.

The small number of registers, six, is testament to the availability of fast on-board memory. This fact, coupled with the prescence of a simple instruction set makes for fast data paths when executing a task. The six registers are as follows;

- 1. The workspace pointer which points to an area of store where local variables are kept.
- 2. The instruction pointer which points to the next instruction to be executed.
- 3. The operand register which is used in the formation of instruction operands.
- 4. The A,
- 5. B,
- 6. and C registers which form an evaluation stack, and are the sources and destinations for most arithmetic and logical operations.

One of the design decisions of the transputer is that it should be programmable in a high level language and therefore the small and simple instruction set makes for easy and efficient compilation. The instruction set is independent of processor wordlength so that the microcode is equally applicable to two transputers of differing wordlength. Each instruction is 1 byte. The first four most significant bits are the function code and the second four are the data. Included in the set are 13 of the most necessary functions for a computer to operate realistically, including ;

load constant	add constant	
load local	store local	load local pointer
load non-local	store non-local	load non-local ptr
jump	conditional jump	call

Two of the function codes are used to allow for the extension of the operand to any length. They are 'prefix' and 'negative prefix'. The 4 least significant bits of the operand register are used to hold the data bits which is then treated as the operand. In the case where the prefix instruction is used the 4 data bits are loaded as normal and then shifted up 4 places. The negative prefix is the same except the data is complemented before shifting. Thus operands of any length up to the size of the operand register can be represented. Even with this facility research shows that approximately 80% of executed functions are encoded in a single byte. Therefore in only 20% of cases is a prefix used. This augurs well for encoding efficiency. This means also that several instructions will be got during a memory fetch cycle.

Transputers not only operate in parallel but also support the OCCAM model [A4][A5][A6][A7][A8][A9][A10] of concurrency internally. There is a microcoded scheduler present which enables any number of processes to be executed concurrently by sharing the processor time. The actual processor itself does not have to dynamicaly allocate the storage space since this is handled by the OCCAM compiler. A feature of the scheduler is that it prevents inactive processes from consuming processor time.

OCCAM communications are point to point, synchronised and unbuffered. Between transputers the channel is implemented by a point to point serial link and internally the channel between two processes is realised by a single word in memory. Referto[A4] for a more detailed account of the internal communications of transputers.

The clock of the transputer operates with a period of 1 microsecond and it's current value can be accessed via a 'Read Timer' instruction. A process may arrange it's input against the timer so that it begins to execute a fixed point in time. This is done using a 'Timer Input' instruction. This is basically a descheduling of the process priority. On arrival of the specified time the process is then re-scheduled.

A.2 THE BBN BUTTERFLY

Of all the commercial parallel processors supporting shared memory the BBN Butterfly [A11][A12] is the most popular. It was developed in Edinburgh by the BBN laboratory whose active fields of research include ;

- Advanced control theory
- Process monitoring and data analysis
- Artificial intelligence
- Computer simulation systems
- Computer networks and communications
- Parallel processing

By drawing from these areas of expertise the Butterfly parallel processor was designed to be a powerful and flexible tool in the hardware realisation of real time control applications. The following are the main architectural features of the processor ;

- It is a MIMD machine
- It is a tightly coupled, shared memory machine
- It is expandable over a wide variety of configurations
- It provides an integrated environment for both numeric and symbolic computation
- The machine has a very large memory.

Shared memory on the Butterfly is implemented by a 'unique switch technology architecture' [A11]. Two of the pioneering control tasks of the Butterfly were the Falcon system and the analysis of airframe test data. Falcon is a system whereby, in a chemical plant, the production of paracetamol is both monitored and simulated whilst it proceeds through a four stage chemical reaction. The system runs through multiple batches to detect imperfections in the production process while pinpointing faults. Thus an operator, by careful monitoring, is capable of improving the production quality of the plant. The whole start to finish process takes twelve hours and process simulation takes about 10 minutes on the Butterfly. This enables process results for a particular set of plant parameters to be obtained quickly without having to actually produce any paracetamol. The Falcon system also incorporates a diagnostic facility based on AI techniques.

The second application was real time data acquisition and reduction in a system used for analysis of airframe test data. Because of the nature of such tests quite a considerable amount of data is generated at reasonably high rates. The Butterfly is used to multiplex, sift and store the data and proves to be cost effective for this application. Languages which are suitable for the processor include parallel versions of FORTRAN 77, C, Ada and Lisp. This means that AI techniques suitable for control purposes may be implemented on the Butterfly parallel processor.

A.3 AMDAHL VECTOR PROCESSOR SERIES

Amdahl have developed a range of four vector processors [A13]. The hardware and software emphasis are on reliability. All processors, as are most Amdahl computers, are IBM system/370 compatible. Main memory consists of 256 Megabytes which allows for the running of large memory tasks. There are 16 or 32 i/p channels. This means many external devices may be attached. Operations can be overlapped since both the scalar and the vector units of the VP are highly pipelined.

□ Operating system : IBM MVS/XA

□ Languages : FORTRAN 66
FORTRAN 77
(Libraries are in FORTRAN)

□ VP500	142 MFLOPS
□ VP1100	286 MFLOPS
□ VP1200	571 MFLOPS
□ VP1400	1142 MFLOPS

The Amdahl vector processor range is the only series of machines delivering more than a Gigaflop that is IBM compatible and can run IBM's mainline operating system.

A.4 CYBERPLUS, Control Data Corporation

The CYBERPLUS [A14] is a parallel processing system employing a multi-processor and pipelined functional unit concept. Each processor is connected to a CYBER 180 computer via a ring group, which allows up to 16 processors to be integrated on a single ring group to the CYBER. Multiple ring groups of processors may also exist in the same CYBER 180. These can be connected to the memory of the CYBER as well as the memory of each CYBERPLUS processor. Peak performance is 91 MFLOPS in 32-bit mode and 61 MFLOPS in 64-bit mode.

- Operating system : That of the CYBER 180 which is currently a dual state machine running two O.S.'s NOS & NOS/VE.
- Languages supported : FORTRAN, Pascal, Prolog, Lisp, Cobol and C. Software for CYBERPLUS runs on CYBER 180. Uses FORTRAN ANSI 77 compiler.
- Performance (Dongarra Benchmark [A14])

FORTRAN	2.5 MFLOPS per processor
CODED-BLAS	12 MFLOPS per processor

The CYBERPLUS system incorporates direct memory access which allows for inter-processor transfer rates of up to 2600 Megabits.

A.5 ASPRO (ASsociative PROcessor)

This computer is produced by Goodyear Aerospace Corporation (GAC) [A15] and is in fact quite small at only 0.44 cu.ft. It operates in SIMD fashion and can deliver enormous computational power. Initially designed for use in the Grumman E-2C AEW aircraft it has also found favour in some commercial applications. It's small size and low power consumption (260W) are attributable to the use of custom CMOS VLSI and multi-chip CMOS random access memory.

Memory consists of 16 CMOS RAM chips plus additional logic chips for buffering, in a single hybrid and the custom VLSI chip contains 32 processing elements or the equivalent of 8,000 transistors.

□ Operating system	: VAX/VMS or UNIX
□ Languages supported	: ASPRO-Assembler FORTRAN 77
□ Performance	: 40 - 50 MIPS

A.6 NCUBE Parallel Processing Systems

A NCUBE parallel processing system [A16] combines a serial host processor with a sophisticated parallel processor called a "hypercube" or "N-cube" with from 4 to 1024 processing nodes. These nodes are 100% VLSI, with 128 KB or 512 KB memory and a proprietary processor chip. Into this propietary processor are integrated IEEE 32-bit and 64-bit floating point, error correcting memory interface, and 22 DMA communications channels. Compatible systems range from 4-node PC/AT-based to a 1024-node supercomputer.

□ Operating system	:	AXIS ... runs on host
		VERTEX ... runs on node
□ Languages supported	:	FORTRAN 77 and C
		Medium grain parallelism
□ Performance	:	Up to 2000 MIPS
		400 MFLOPS
		8x180 MB/sec I/O
		8 GB/sec mem

References

[A1] Taylor, R. "Survey of Transputer Applications". May 1986.

[A2] May, D. "The Transputer", "Major Advances in Parallel Processing", Editor C. Jesshope, Technical Press 1987, pp. 33-47.

[A3] May, D. & Shepherd,R. "The Transputer Implementation of OCCAM" in "Parallel Processing in Control" Editor P.J. Fleming, IEE Control Engineering Series 38, Publ. Peter Peregrinus Ltd 1988, p.85.

[A4] Fleming, P.J. "OCCAM and the Transputer" in "Parallel Processing in Control", Editor P.J. Fleming, IEE Control Engineering Series 38, Publ. Peter Peregrinus Ltd 1988, pp.27-99.

[A5] Taylor, R. "Concurrent Programming in OCCAM" in "Major Advances in Parallel Processing" Editor C. Jesshope, Technical Press 1987, pp.221-236.

[A6] Fleming, P. J. "Programming in OCCAM", IEE Computing and Control division conf., "Parallel Processing in Control - The Transputer and other architectures", 20-22 Sept. 1987.

[A7] Fleming, P. J. "OCCAM model of Parallelism" IEE Computing and Control division conf., "Parallel Processing in Control - The Transputer and other architectures", 20-22 Sept. 1987.

[A8] Brain, S. "Transputer Implementation of OCCAM" IEE Computing and Control division conf., "Parallel Processing in Control - The Transputer and other architectures", 20-22 Sept. 1987.

[A9] Jones, D. I. "OCCAM Structures in Control" IEE Computing and Control division conf., "Parallel Processing in Control - The Transputer and other architectures", 20-22 Sept. 1987.

[A10] Griffiths, S. "Introduction to OCCAM Development", IEE Computing and Control division conf., "Parallel Processing in Control - The Transputer and other architectures", 20-22 Sept. 1987.

[A11] Hunt, K. O'Neill, G. "Real time data acquisition and control on the Butterfly parallel processor ", IEE Computing and Control division conf. "Recent advances in Parallel Processing for Control", Bangor, Wales, 7th July 1988.

[A12] BBN advanced computers " Butterfly parallel procesor " in "Major advances in parallel processing" Editor C. Jesshope, Technical Press 1987, pp.351-353.

[A13] Amdahl corp. "Amdahl vector processor series " in "Major advances in parallel processing" Editor C. Jesshope, Technical Press 1987, pp.347-348

[A14] ControlData corp. " CYBERPLUS ". in "Major advances in parallel processing" Editor C. Jesshope, Technical Press 1987, pp.356-358.

[A15] Goodyear Aerospace corp. " ASPRO " in "Major advances in parallel processing" Editor C. Jesshope, Technical Press 1987, pp.366-367.

[A16] NCUBE " Ncube parallel processing systems" in "Major advances in parallel processing" Editor C. Jesshope, Technical Press 1987, pp.379-381.

APPENDIX B

Parallel Languages - An Evaluation

Of paramount importance to the commercial growth of parallel processing is the competence of computer languages usable on parallel architectures. Fortunately, as parallel processing becomes more necessary in many fields most language writers have seen the wisdom in producing a 'parallel' version of their language compilers. This chapter contains sections outlining several of these languages and both their merits and demerits.

B.1 OCCAM

In a parallel programming structure several processes may be running simultaneously. If, as is often the case, some of the processes rely on others for data then some form of synchronisation must be implemented. In a conventional language this would create quite a burden for the programmer, but in occam [B1]-[B7] this problem is alleviated somewhat. An important concept in occam is the 'process'. This is the fundamental unit of programming. It is defined as a set of operations which begins execution at a point in time and likewise terminates at a point in time. Many of these processes may execute concurrently. In occam a process is built from three primitive processes or 'primitives' which are;

Assignment
Input
Output

These can be combined to form the following constructs ;

SEQ	...	Sequential
PAR	...	Parallel
ALT	...	Alternative

and

IF	...	Conditional
WHILE	...	Iteration

Communication between processes are done via channels which are point to point links. The channels are one-way and are self-synchronising since communications only take place when both the sender and receiver processes are ready. In OCCAM if one processes is ready before the other then it will wait til the other is ready also. Examples of the three occam primitives are shown below;

(i). Assignment

```
v := e
```

"v" is assigned the value of "e".

(ii). Input

```
c ? v
```

A value is sought from a channel "c" to be stored in "v".

(iii). Output

```
c ! e
```

The value of expression "e" is output to channel "c".

The SEQ and PAR constructs are an integral part of OCCAM. SEQ indicates that any proceeding operations are to be executed sequentially. The following example shows this;

```
SEQ
  ADC? meas.signal
  scaled.signal := meas.signal*scale.factor
  DAC! scaled.signal
```

A measurement signal is input through ADC, scaled and then output through DAC. This can be graphically represented by figure 6. Contrasting to this is the PAR construct which indicates parallel operation of the following processes. As an example consider two processes to be executed concurrently;

```
PAR
  ... process 1
  ... process 2
```

Indentation indicates constituents of the same process. In the above example process1 and process2 are both component processes of the parent PAR process.

Occam does not support transmission of values between processes by use of a shared variable. The use of channels is employed to overcome this. Finally the ALT construct is used where several concurrent processes must wait on input data. The process to be initiated is the one for which data becomes available first. It could be termed a 'first-come first-served' system.

B.2 ADA

The ADA language [B8] is not exclusively a parallel programming language as it can operate just as efficiently on sequential algorithms. The method by which extra processes are introduced in concurrent operation is by use of the 'task' command. For example, consider a scenario where several external devices must be serviced at regular, but unpredictable intervals. In sequential programming this could only be overcome by the use of polling. In ADA each of the device service routines would be branched from the main process using the task statement. The general structure of an ADA programme is as follows:

```
procedure NAME is
        -- specification of the data to
        -- be used by the programme
begin
        -- sequence of statements defining
        -- the actions to be performed
end  NAME
```

Say there are two devices, dev1 & dev2 which are keyboards, then the task operation is introduced as follows ;

```
task DEV1;

task body DEV1 is
        CH:CHARACTER;
begin
        loop
                READ(1,CH) ; WRITE(1,CH) ;
        end loop
end DEV1;

task DEV2;

task body DEV2 is
        CH:CHARACTER;
begin
        loop
                READ(1,CH) ; WRITE(1,CH) ;
        end loop
end DEV2;
```

The two processes may now run independently. The main programme from which the tasks originate is then termed the parent processes and the tasks the offspring. In order for the parent process to terminate the offspring must first be completed.

For communication between two processes shared variables may be used but as in most protocols it is not advisable for two reasons. Firstly, there is the synchronisation problem which occurs if the consumer process is operating at a different rate (either slower or faster) to the producer process. Secondly, access to the shared variable(s) must be limited to one process at any given time and this is known as the mutual exclusion problem. ADA does have a much more elaborate method of inter-process communication which is termed the 'rendezvous'. The caller makes an 'entry' call which is defined by some other task known as the server. This entry is similar to a procedure. The server may or may not respond immediately but when it does so it issues an accept statement and at this point the rendezvous commences. Data is transferred via parameters contained in the entry call and reply data in the accept statement. On completion of the accept statement the rendezvous is broken and both processes recommence their respective operations.

B.3 PARLOG

PARallel LOGic programming (PARLOG) [B9] is quite a novel concept. It is based on a technique of incorporating concurrency into standard logic programming. Most standard programming languages are 'imperative' because of their syntax structure. From operations information about machine state can be derived whereas in logicprogramming the state of the hardware is completely transparent to the programming code. In 1965 LISP was developed as the first declarative programming language and today it still remains the most widely used of it's type. It is what is termed a 'functional' programming language because it is based on abstract formalism. It has a strong relationship with human language thus making it very high level.

Another form of functional or declarative programming, a type with which is of most concern here, is logic programming where statements are predicate logic i.e. Horn clauses [40]. The most frequently used logic language is prolog [40] whichwas developed in 1972 as a human orientated language. Parallelism in declarative programming can be divided into two main catagories ;

(1). **RESTRICTED PARALLELISM** : The concurrent evaluation of several arguments of a function expression. This is quite straightforward to implement since arguments are essentially independent.

(2). **STREAM PARALLELISM** : The evaluation of a function expression concurrently with one of it's arguments. The value of the argument can be communicated incrementally to the expression.

It must be kept in mind however, before applying parallel techniques to PROLOG, that it is not a suitable language for this purpose and neither is any logic language for that matter but with the advent of parallel processing machines an attempt is being made to create a compatibility between software and hardware. With specific regard to logic programming the above classifications of parallelism may be further refined ;

(1). **Restricted AND parallelism** : Concurrent evaluation of several calls in a conjunction which are independent, i.e. don't share the same variables.

(2). **Stream AND parallelism** : Concurrent evaluation of 2 calls which share a variable, with the value of the shared variable communicated incremently between the calls.

Because of the non-deterministic nature of logic programming two further classifcations of parallelism arise;

(3). OR parallelism : Concurrent application of several clauses in a procedure while solving a call.

(4). All solutions AND parallelism : Concurrent evaluation of several calls in a conjunction, each working on a different solution.

Intuitively, types (3) and (4) are the easiest to implement for the simple reason that no inter-process communication occurs and several processes run concurrently and independently. In fact, it is method (2) that is applied to the logic programme structure to realise a parallel implementation known as PARLOG. This method is chosen since it allows for concurrent processes to communicate via shared variables and this gives a much broader definition to the parallelism than types (3) and (4). Information is then passed incrementally between processes. Each process is a set of clauses and each clause constitutes a sentence. Hierarchal communications is by incremental transfer of data from process to outer function.

B.4 PARALLEL C

As is stated in section 4, the advent of commercially succesful parallel processors and consequently the growth of a lucrative software market has enticed many language writers to develop parallel versions of their languages and the C language is no exception [B10][B11]. C has powerful expressional syntax and is well known to most software engineers. Vast libraries of software written in C exist and huge amounts of software tasks have been implemented by it also. These reasons make it desirable to have a parallel version usable on parallel architectures, such as the transputer. Another attractive feature is that C unlike Occam for example, has dynamic allocation of variables and permits recursion. What remains is to add in, in a logical way, several operators to make C implementable on a transputer system.

B.4.1 CHANNEL DATATYPE

The first and most vital addition is the incorporation of a 'channel' datatype. This provides a vehicle for inter-processor communications.

```
channel     chan1,     chan2,     chan3;
```

This example intialises three channels, chan1, chan2 and chan3. Fortunately the transputers specialised hardware handles all channels in an identical fashion regardless of whether the two processes are on the same chip or not. Each channel when declared occupies one word of memory. Consider the example;

```
int a;
a=chan1;
```

This assigns a value to 'a' by reading a value from channel chan1. It is important to note that channels may be used in a similar fashion to conventional variables, the only difference being that they read their values from other processes rather than a specific memory location. When a process outputs a value to a channel another process must likewise input that value from the channel, as in the following example;

```
channel AtoB;

A( )
 {
     ...
     AtoB = 1;
```

```
          ...
 }

B( )
 {
          ...
printf( "Received from A : %d\n ", (int)AtoB );
          ...
 }
```

B.4.2 LINK

A transputer has 4 high speed serial links, and internally each link is represented by two channels. Three differences between links and channels are :

- □ A link communicates between processes on different transputers,
- □ Links are DMA based, and thus consume no processor time,
- □ Links are placed at fixed addresses,

but otherwise they are essntially the same. The link can be accessed using a pointer as follows;

```
channel *Link0out = LINK0OUT;
```

This defines a pointer to the channel Link0out, which is initialised with the value of LINK0OUT (0x80000000 on a T414 transputer). Sending information across this link is as simple as;

```
*Link0out = val ;
```

which sends the value 'val' over the link.

B.4.3 TIMER

Also incorporated is the system timer. This may be accessed as if it were an integer variable and available to all concurrent processes, thus giving a universal point of reference to all processes. Processes can then be synchronised or may even be put to sleep.

B.4.4 PAR CONSTRUCT

Another important instruction is the 'par' construct which allows the programmer to instantiate many parallel processes.

```
par{
        statement 1;
        /* subprocess 1 */

        statement 2;
        /* subprocess 2*/
                .
                .
                .
        statement n;
        /* subprocess n */
    }
```

Any statements nested in the par construct are executed in parallel.

B.4.5 ALT CONSTRUCT

Not unlike the alt statement in occam, the alt construct is a direct result of the transputer hardware upon which it is implemented. It can, in fact be used in replacement of the switch statement found in conventional C.

```
alt {
        guard guardexp1 :
                code;
        guard guardexp2 :
                code;
        default :
                code;
    }
```

The guards are not unlike 'case' labels in the switch statement. But whereas case labels must contain constant values guards can be dynamic and contain expressions. The guards are examined in order of sequence and if active then the corresponding process is activated, but if all guards are inactive then the alt will be descheduled until one of the channel or timer guards becomes active. There are five types of guards ;

- Boolean guard
- Channel guard without boolean,
- Channel guard with boolean,
- Timer guard without boolean,

References

[B1] Fleming, P.J. "OCCAM and the Transputer" in "Parallel Processing in Control", Editor P.J. Fleming, IEE Control Engineering Series 38, Publ. Peter Peregrinus Ltd 1988, pp.27-99.

[B2] Taylor, R. "Concurrent Programming in OCCAM" in "Major Advances in Parallel Processing" Editor C. Jesshope, Technical Press 1987, pp.221-236.

[B3] Fleming, P. J. "Programming in OCCAM", IEE Computing and Control division conf., "Parallel Processing in Control - The Transputer and other architectures", 20-22 Sept. 1987.

[B4] Fleming, P. J. "OCCAM model of Parallelism" IEE Computing and Control division conf., "Parallel Processing in Control - The Transputer and other architectures", 20-22 Sept. 1987.

[B5] Brain, S. "Transputer Implementation of OCCAM" IEE Computing and Control division conf., "Parallel Processing in Control - The Transputer and other architectures", 20-22 Sept. 1987.

[B6] Jones, D. I. "OCCAM Structures in Control" IEE Computing and Control division conf., "Parallel Processing in Control - The Transputer and other architectures", 20-22 Sept. 1987.

[B7] Griffiths, S. "Introduction to OCCAM Development", IEE Computing and Control division conf., "Parallel Processing in Control - The Transputer and other architectures", 20-22 Sept. 1987.

[B8] Young, S. "An introduction to ADA" 2nd Edition.

[B9] Gregory, S. "Parallel logic programming in PARLOG"

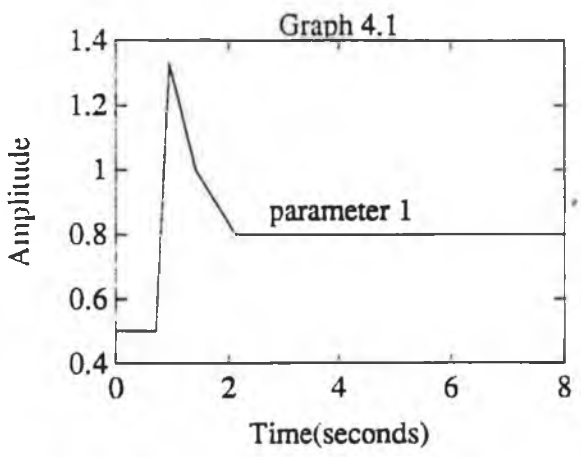
[B10] Poplett, J. "The development of a parallel C", TCC-0910-1 to TCC-0910-7.

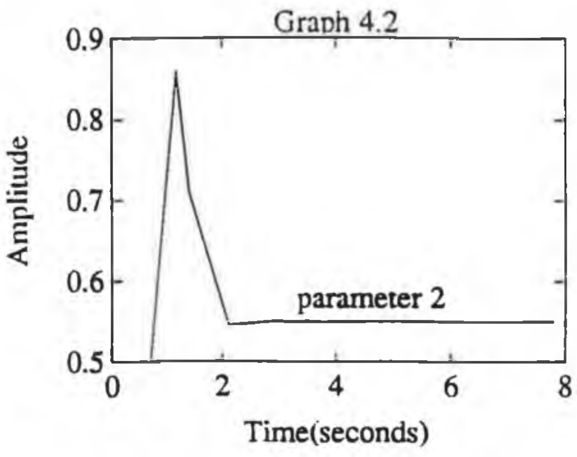
[B11] "Parallel C", Publ. by 3L limited, Supplier: SENSION - Transputer products division, Denton Drive, Norwich, Chesire CW9 7LU.

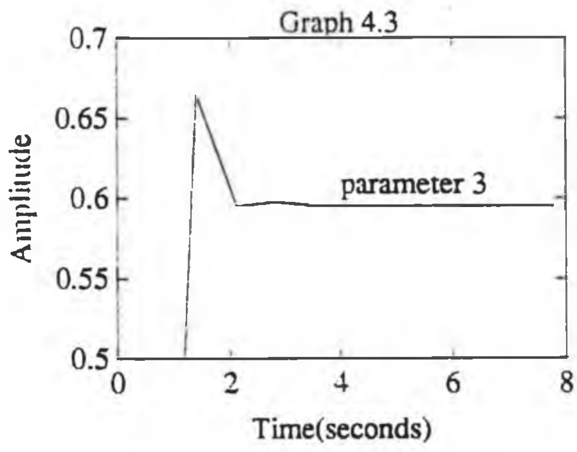
Appendix C
Graphical Data

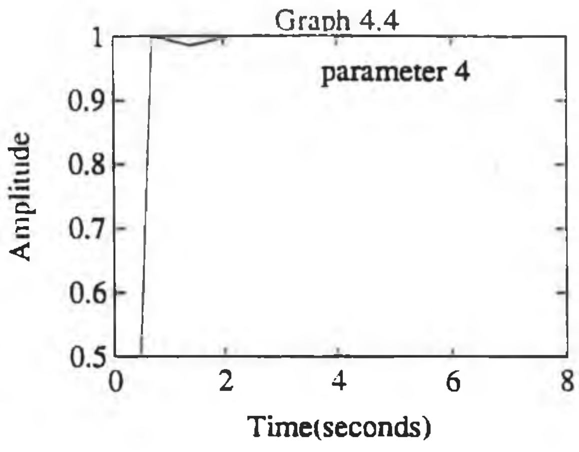
Graphical Index

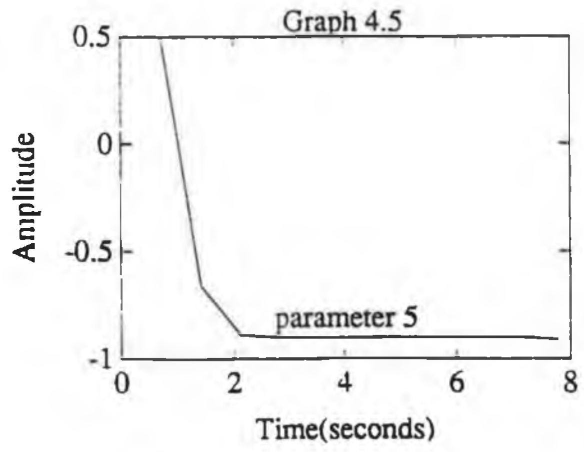
Appendix C: *graphical data*

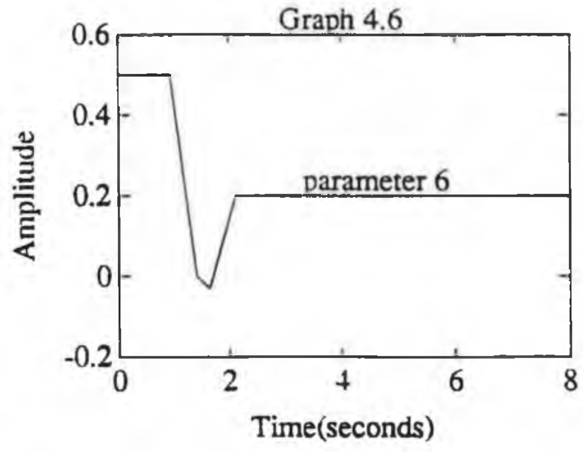


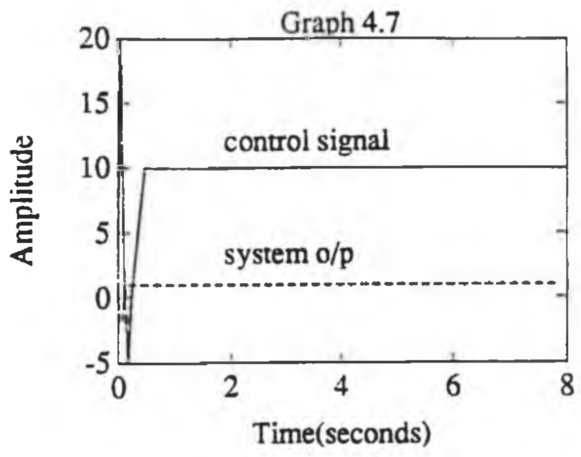


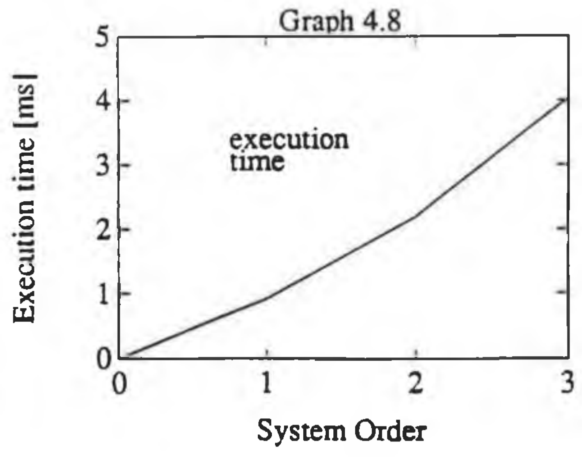


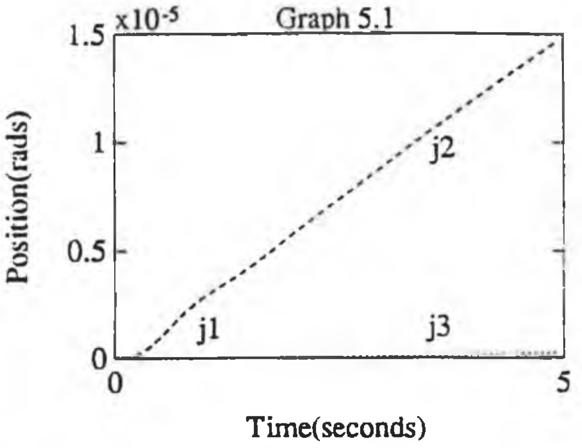


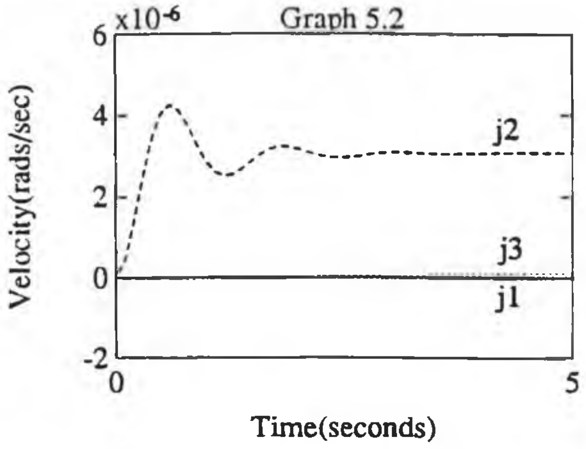


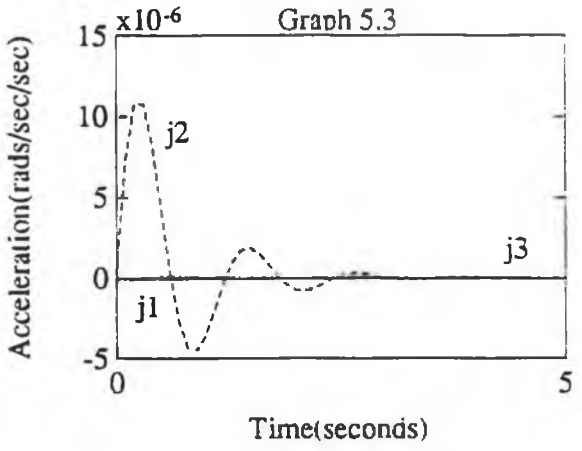


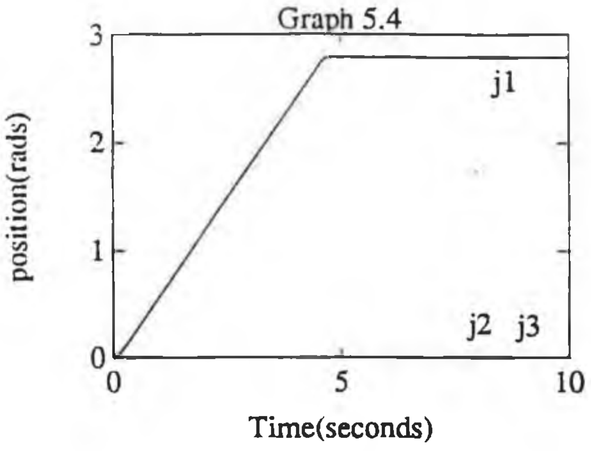


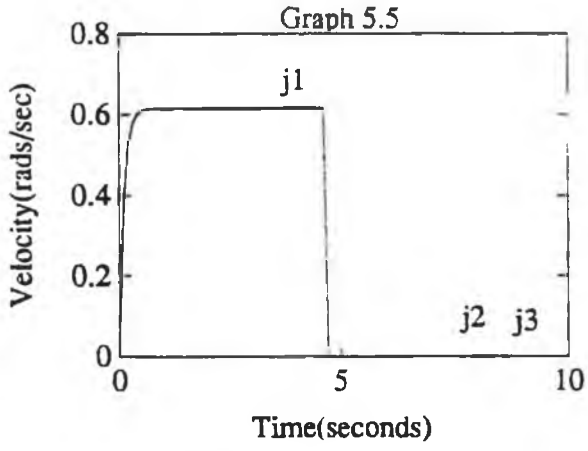


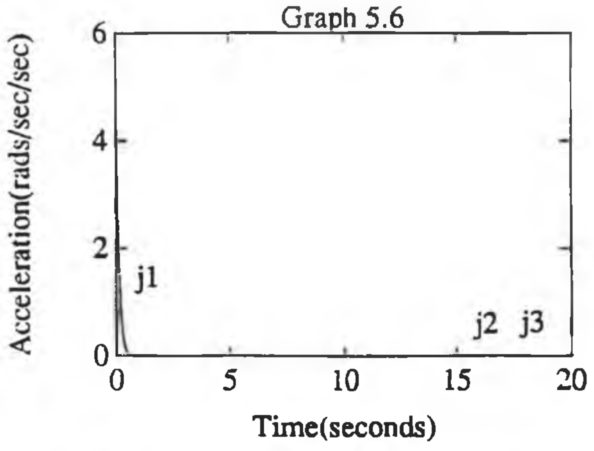

Graph 5.1

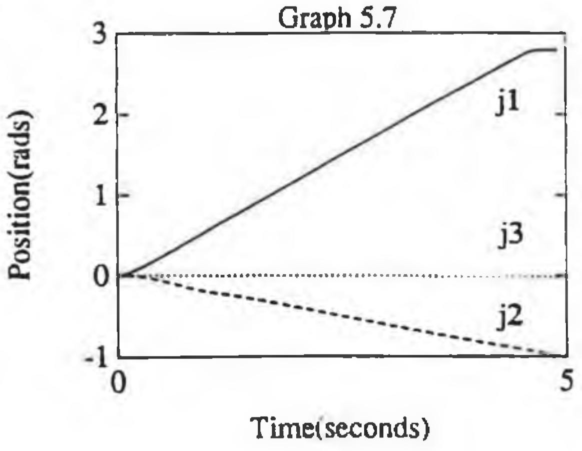

Graph 5.2

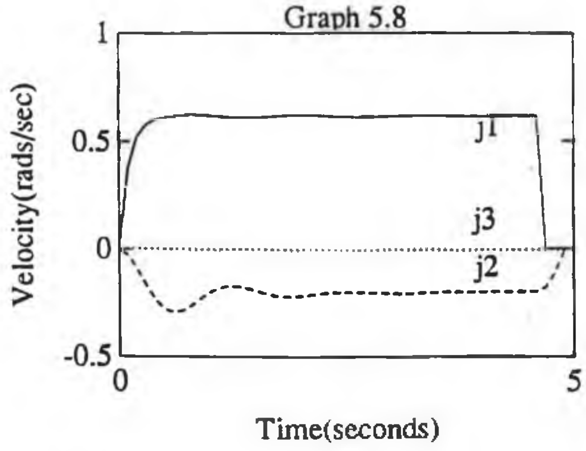

Graph 5.3


Graph 5.4

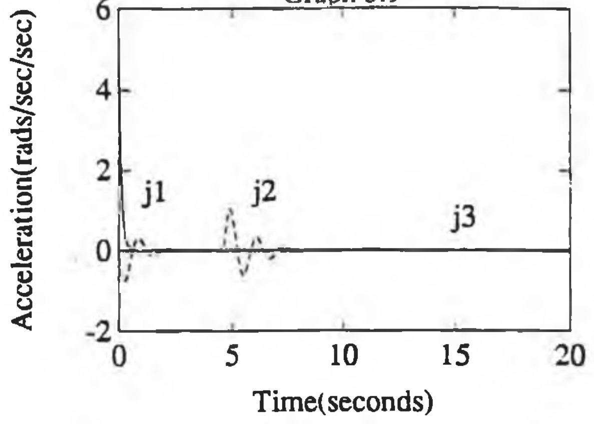

Graph 5.5


Graph 5.6

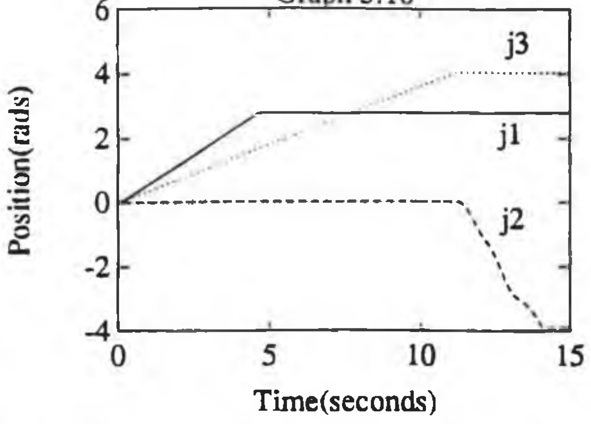

Graph 5.7


Graph 5.8

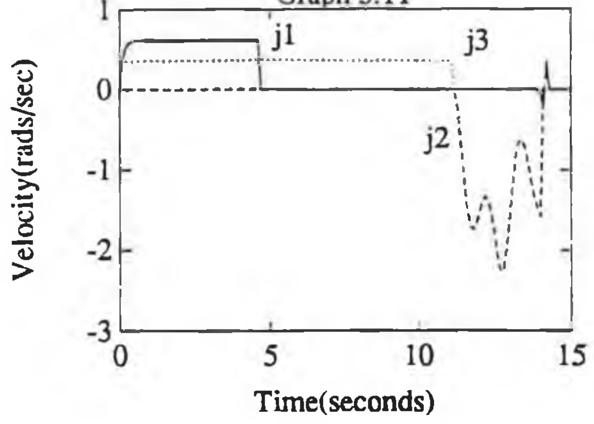
Graph 5.9



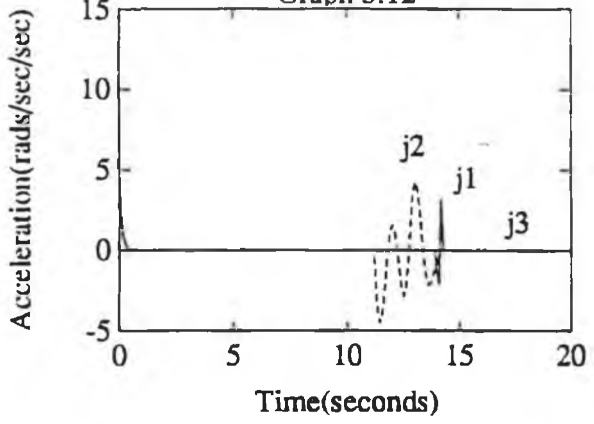
Graph 5.10



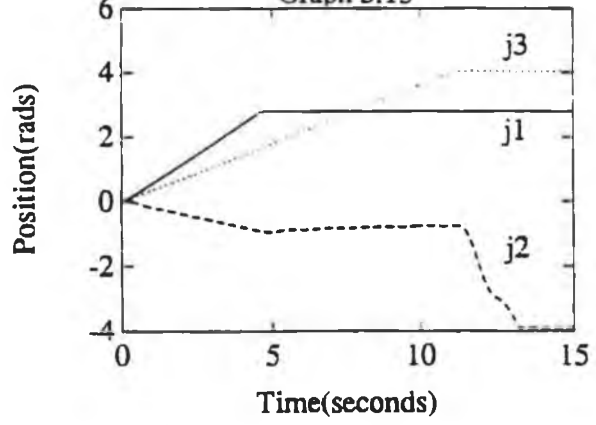
Graph 5.11



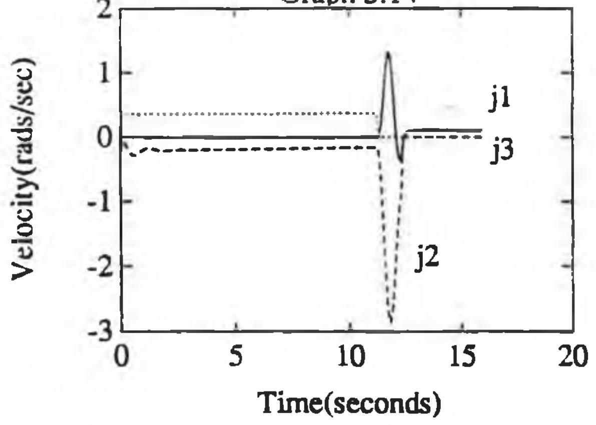
Graph 5.12



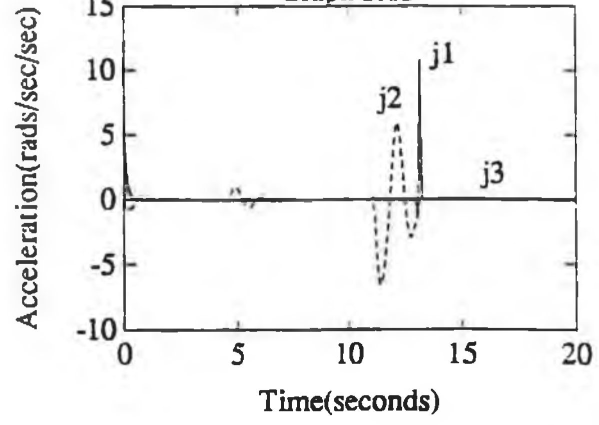
Graph 5.13



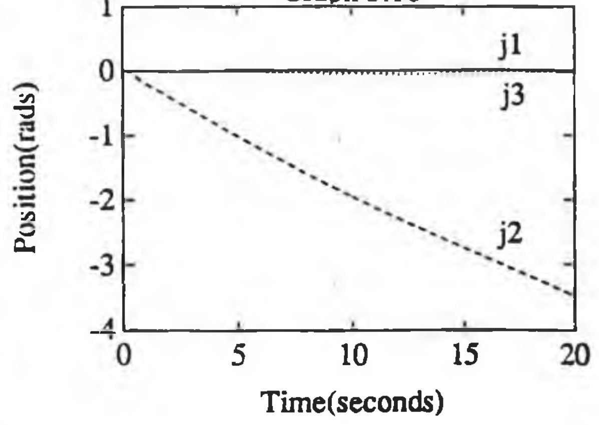
Graph 5.14

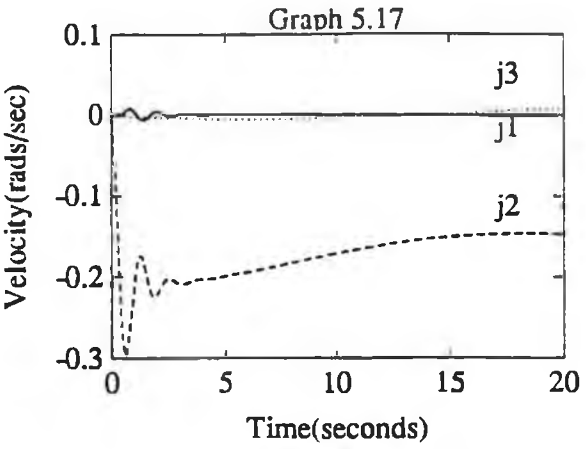


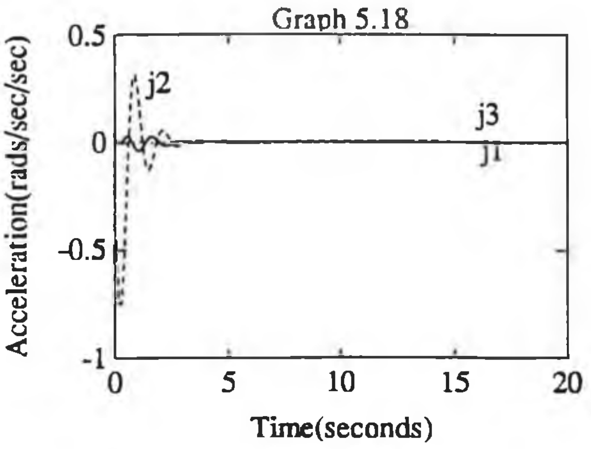
Graph 5.15

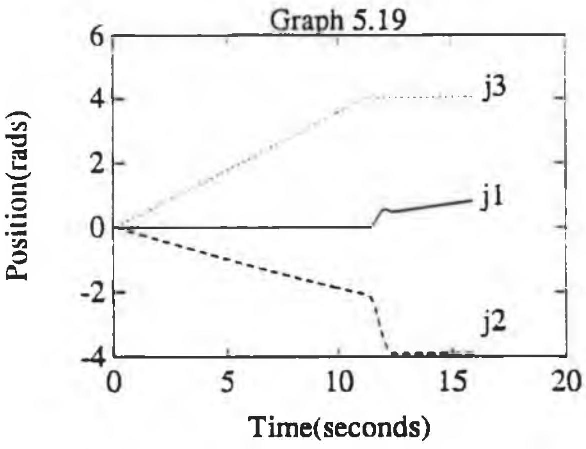


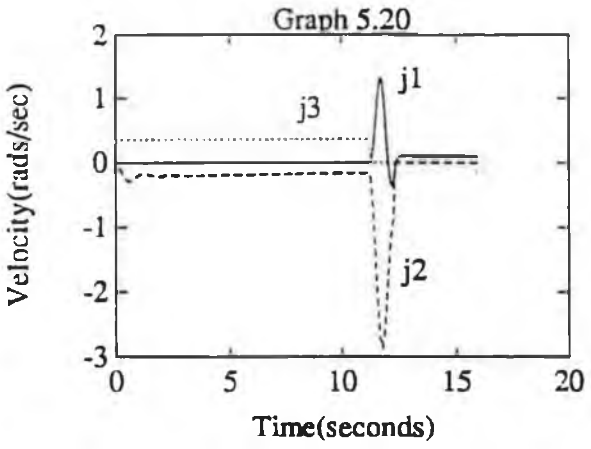
Graph 5.16

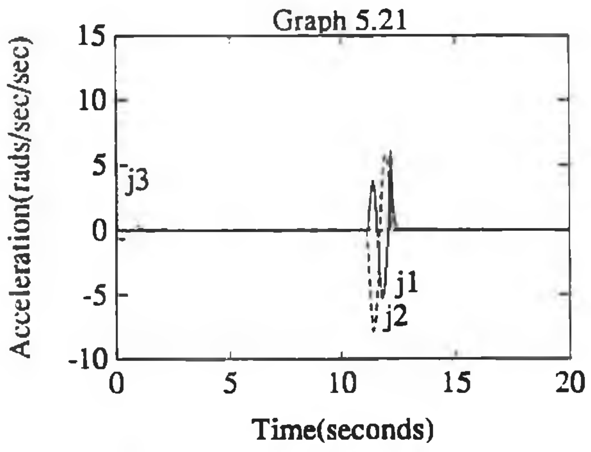


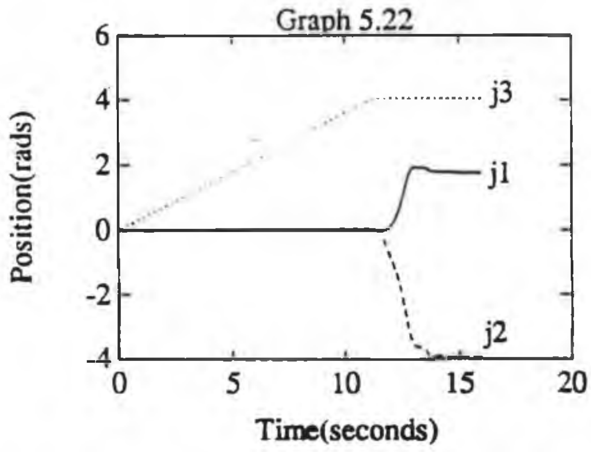

Graph 5.17
Velocity(rads/sec)
Time(seconds)
j3
j1
j2

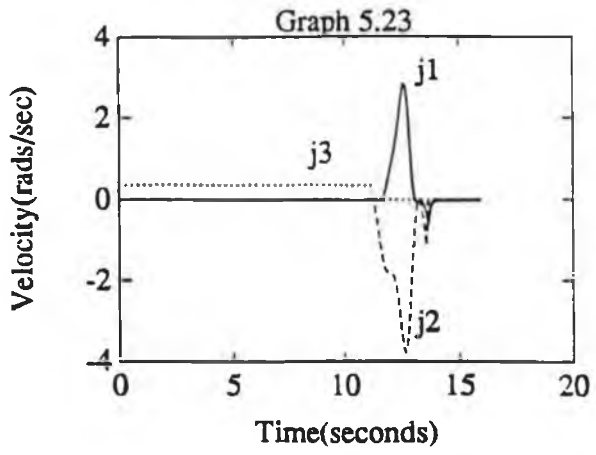

Graph 5.18
Acceleration(rads/sec/sec)
Time(seconds)
j2
j3
j1

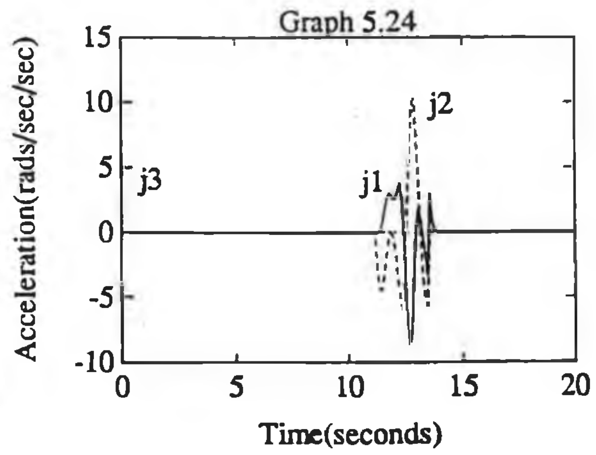

Graph 5.19
Position(rads)
Time(seconds)
j3
j1
j2

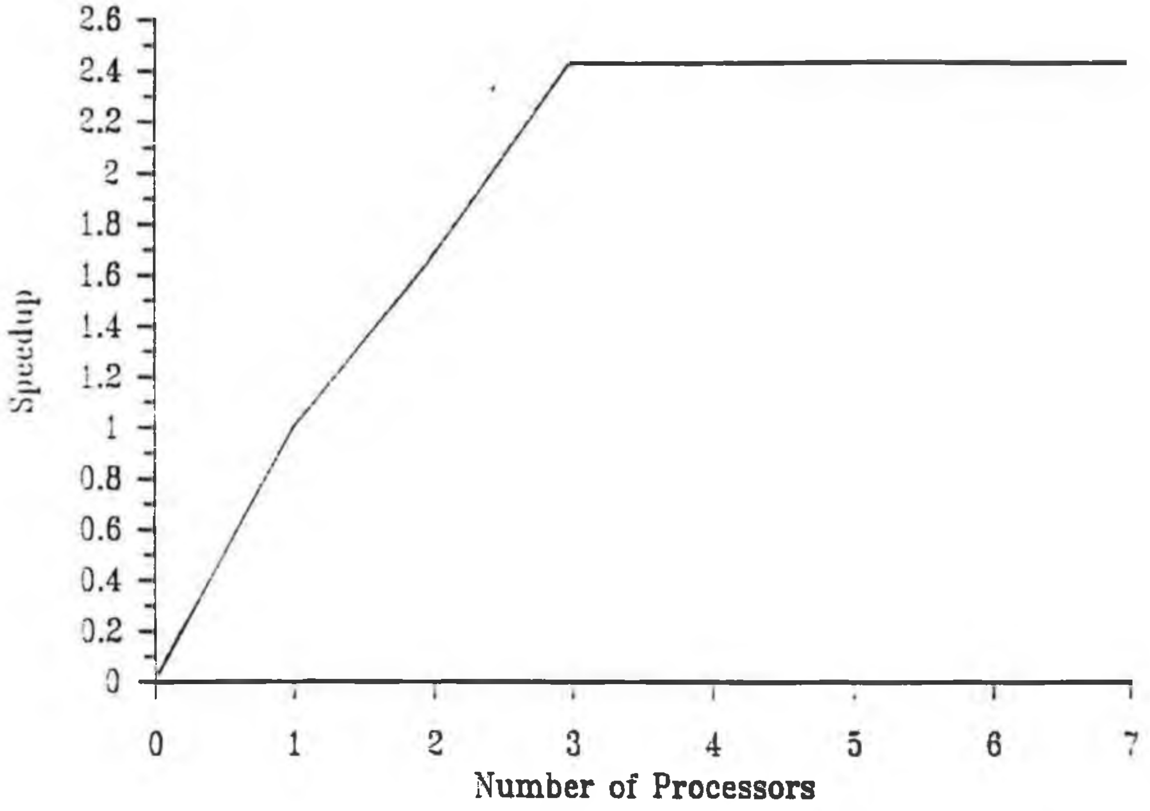

Graph 5.20
Velocity(rads/sec)
Time(seconds)
j1
j3
j2


Graph 5.21
Acceleration(rads/sec/sec)
Time(seconds)
j3
j1
j2

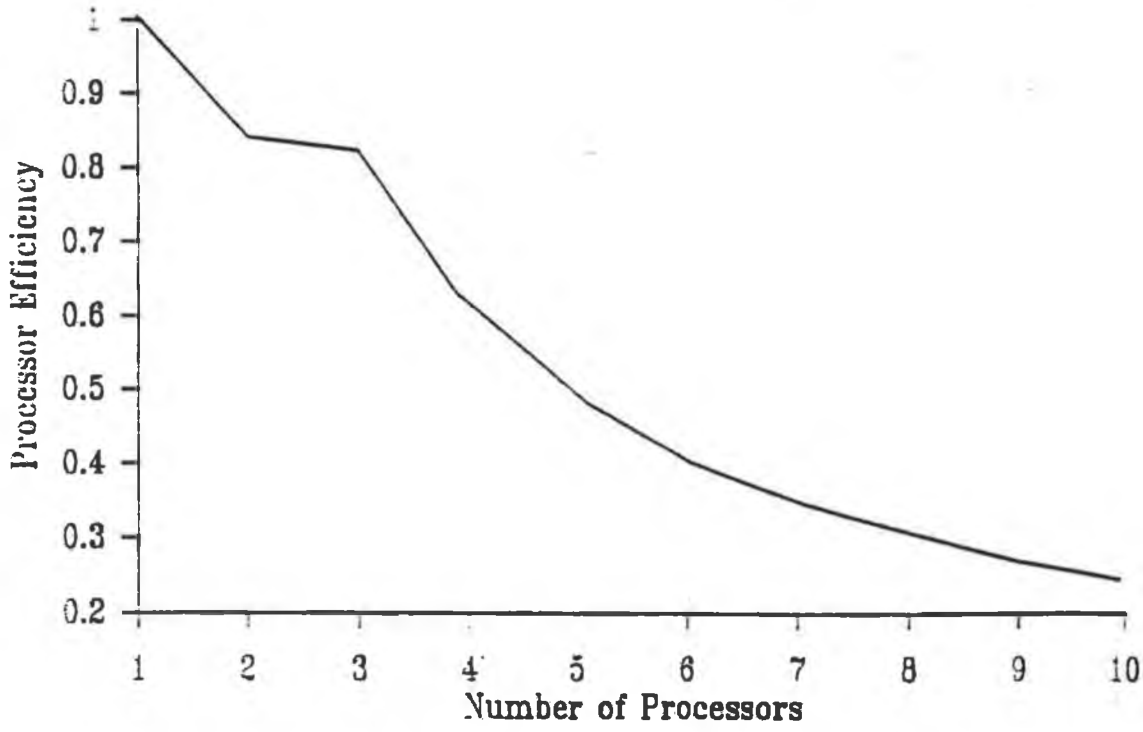

Graph 5.22
Position(rads)
Time(seconds)
j3
j1
j2


Graph 5.23
Velocity(rads/sec)
Time(seconds)
j1
j3
j2

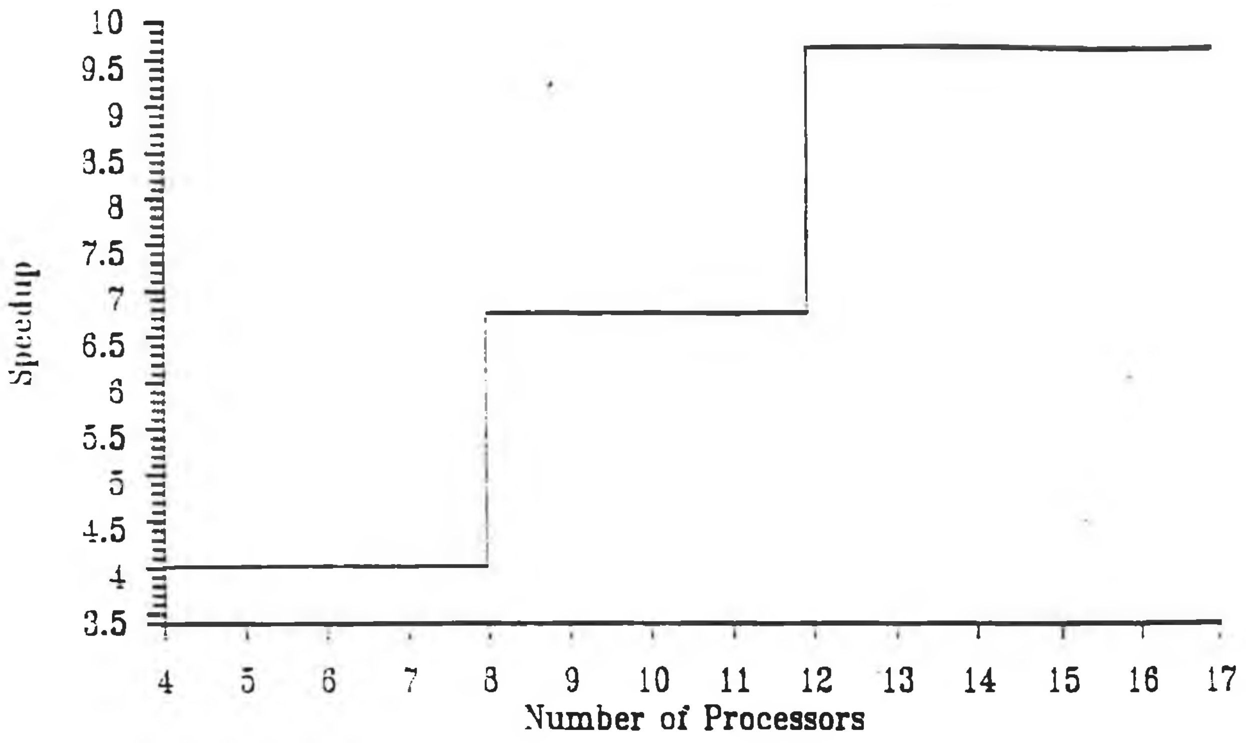

Graph 5.24
Acceleration(rads/sec/sec)
Time(seconds)
j2
j3
j1



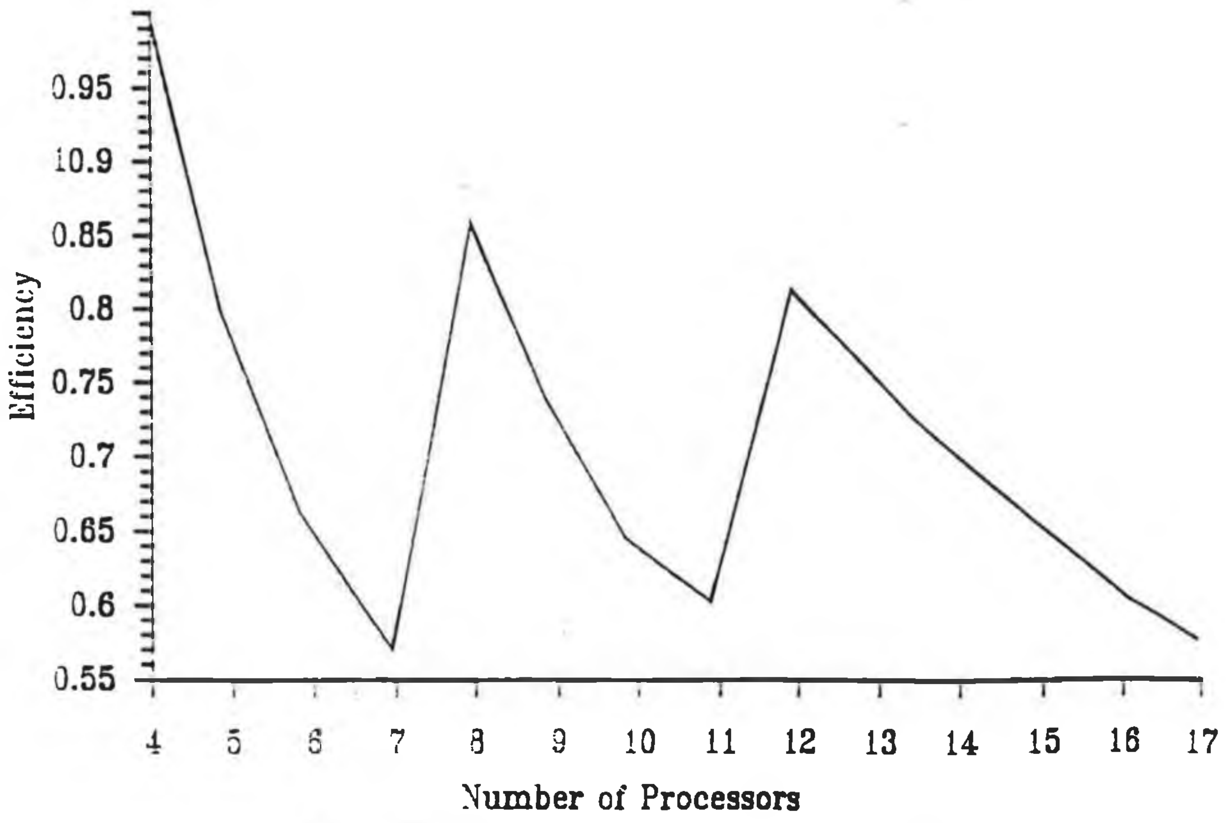
Graph 5.25: Speedup Versus Number of Processing Units



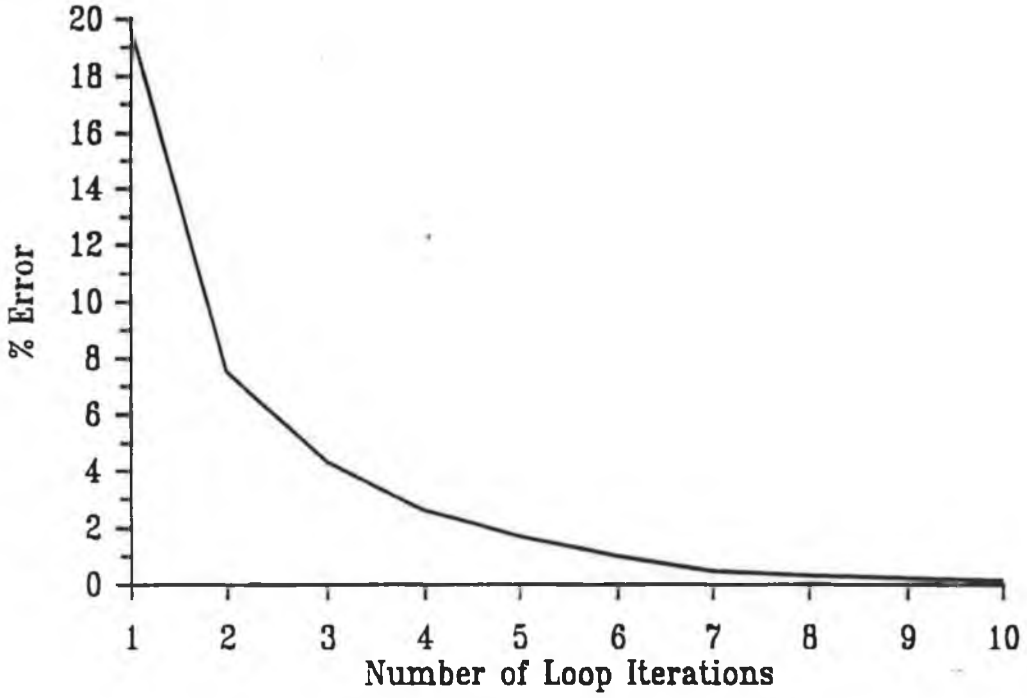
Graph 5.26: Efficiency Versus Number of Processing Units.



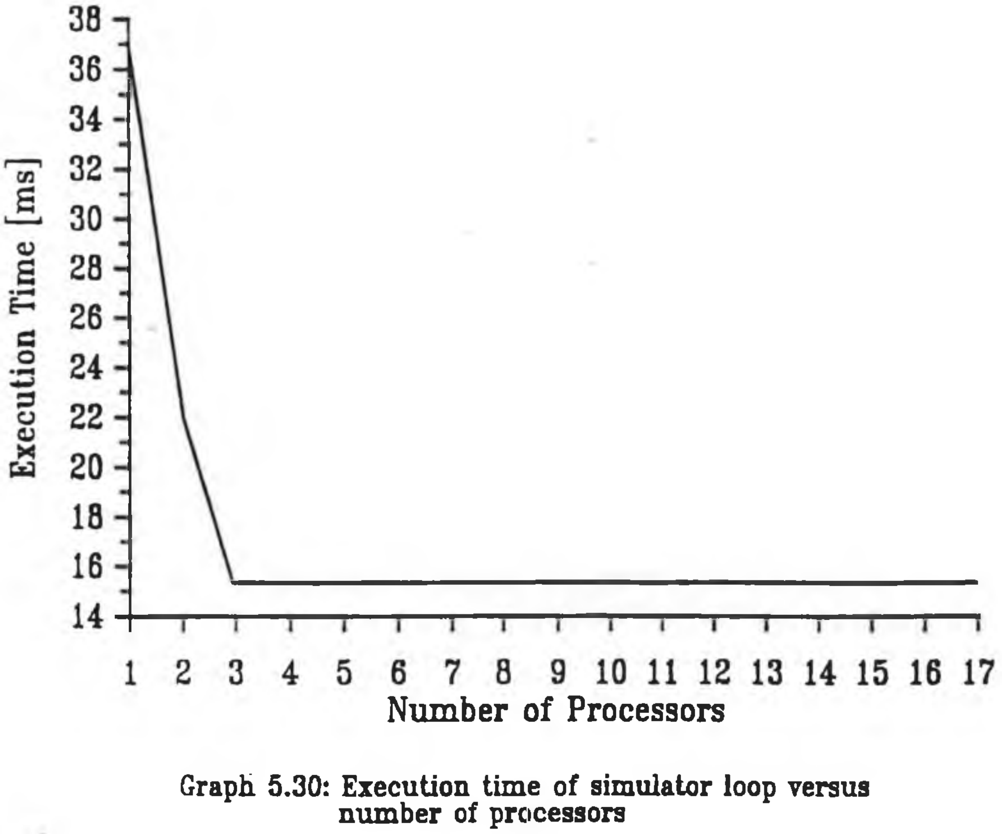
Graph 5.27: Speedup Versus Processing Numbers (With Pipeline)



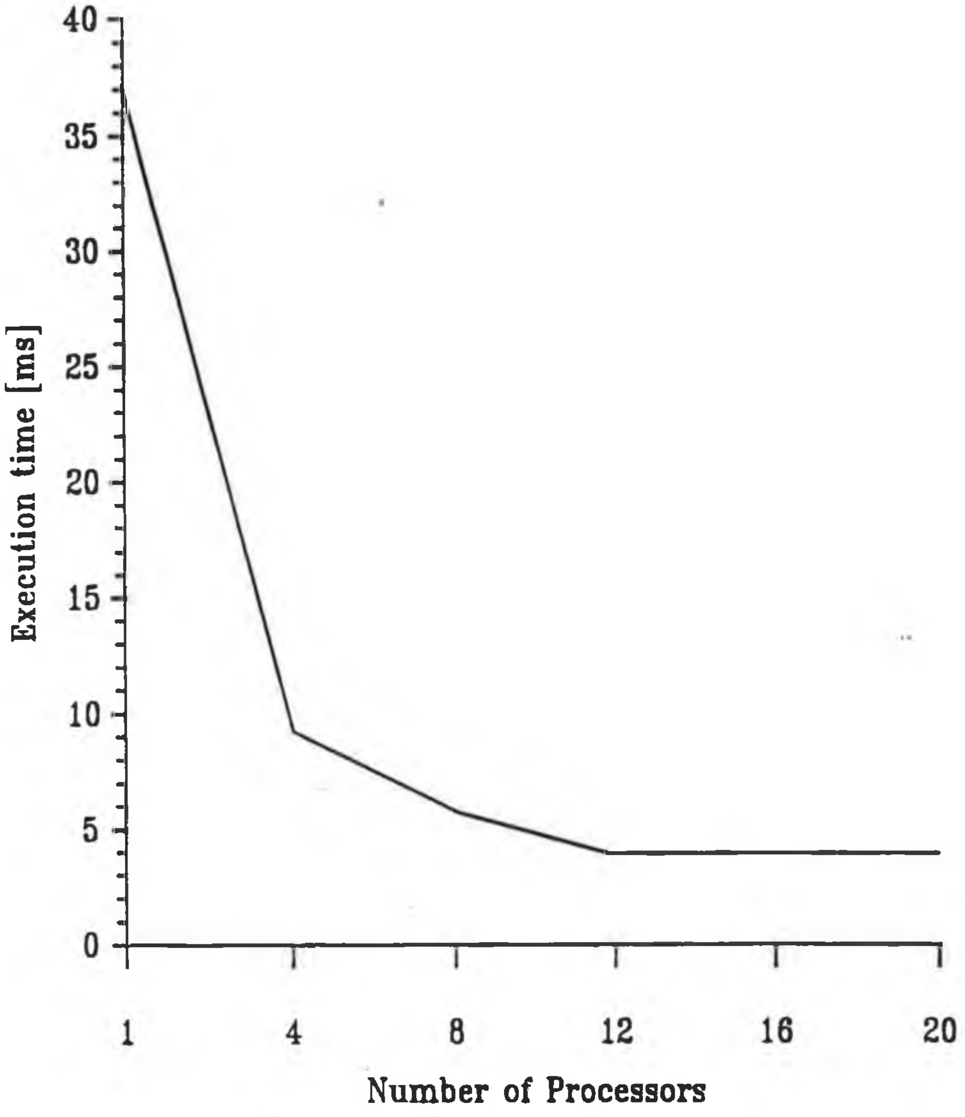
Graph 5.28: Efficiency Versus Processing Numbers (With Pipeline)



Graph 5.29: Percentage error in pipelined simulator versus the number of iterations of simulator loop.

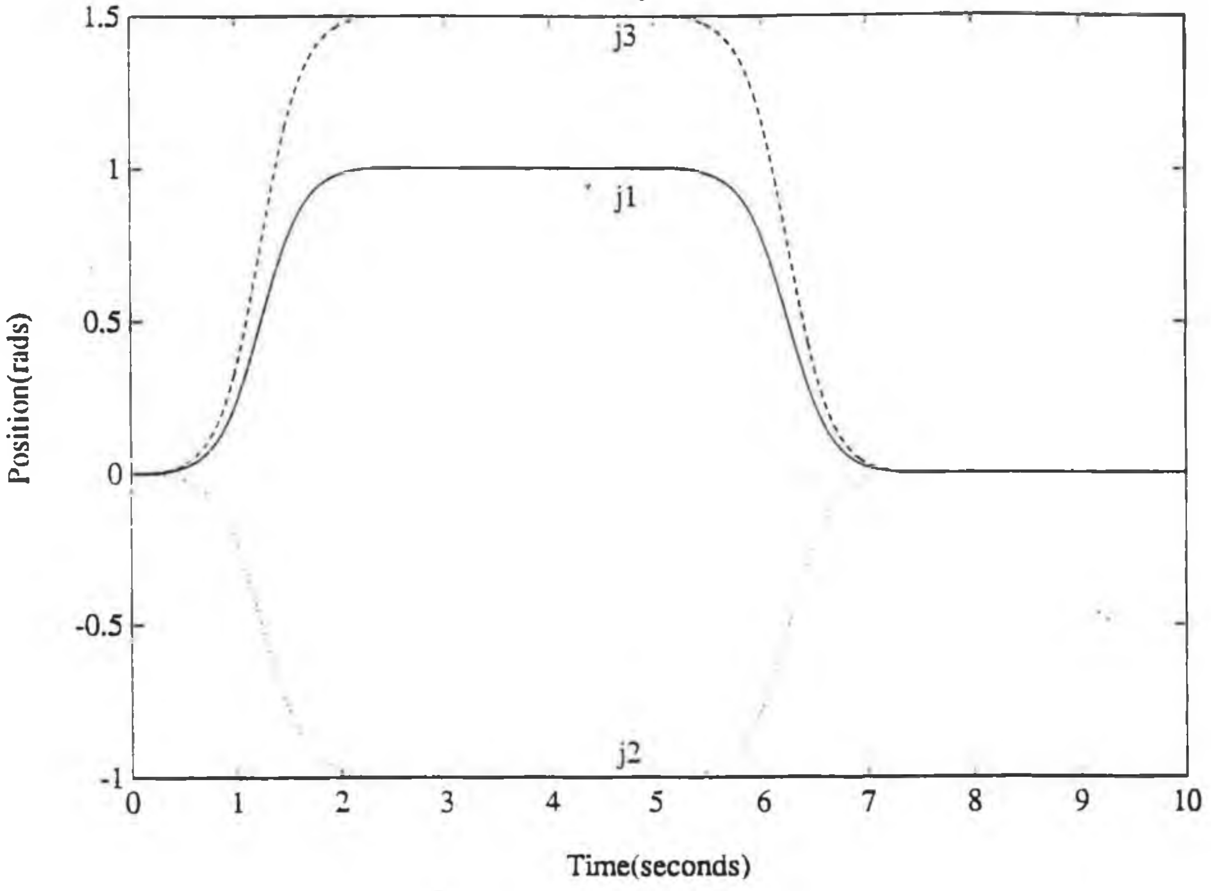


Graph 5.30: Execution time of simulator loop versus number of processors

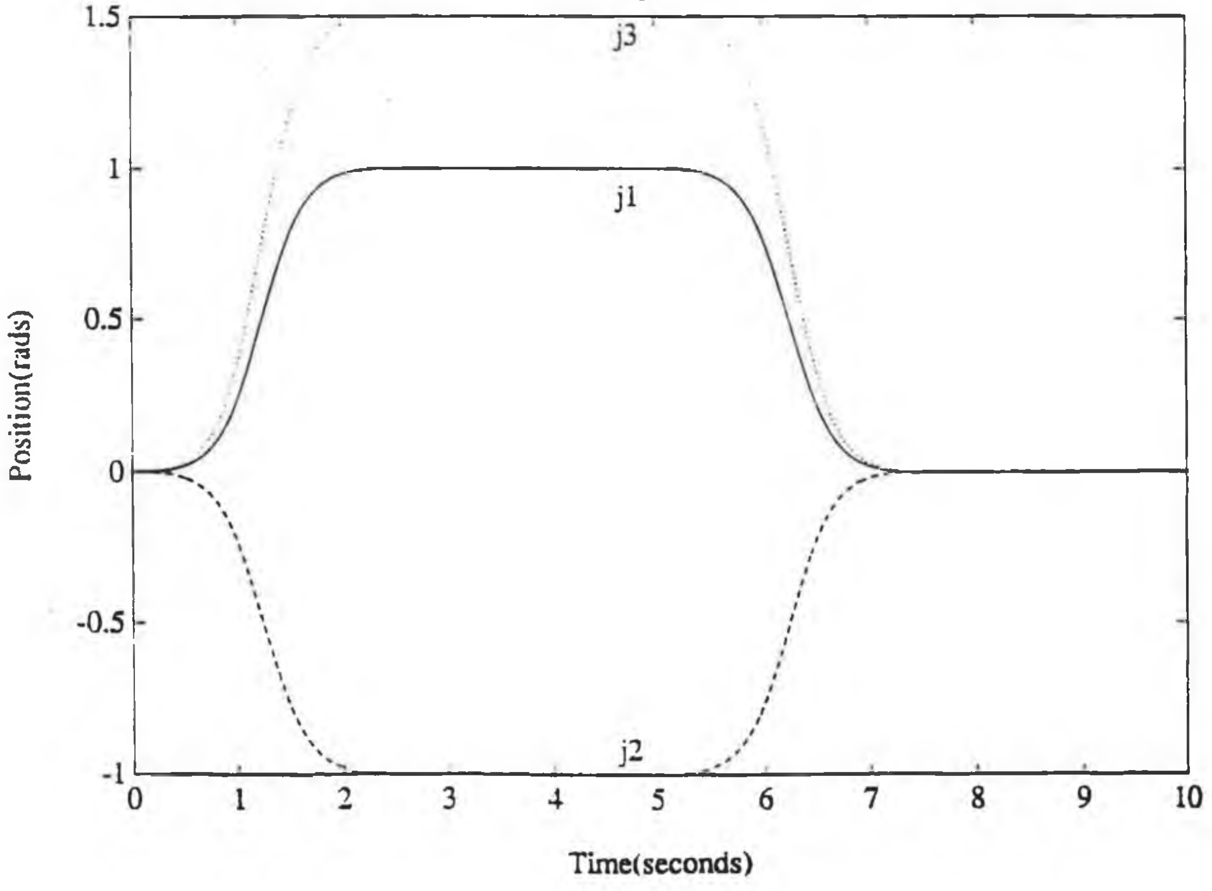


Graph 5.31: Execution time of pipelined simulator versus number of processors defined in modules of four

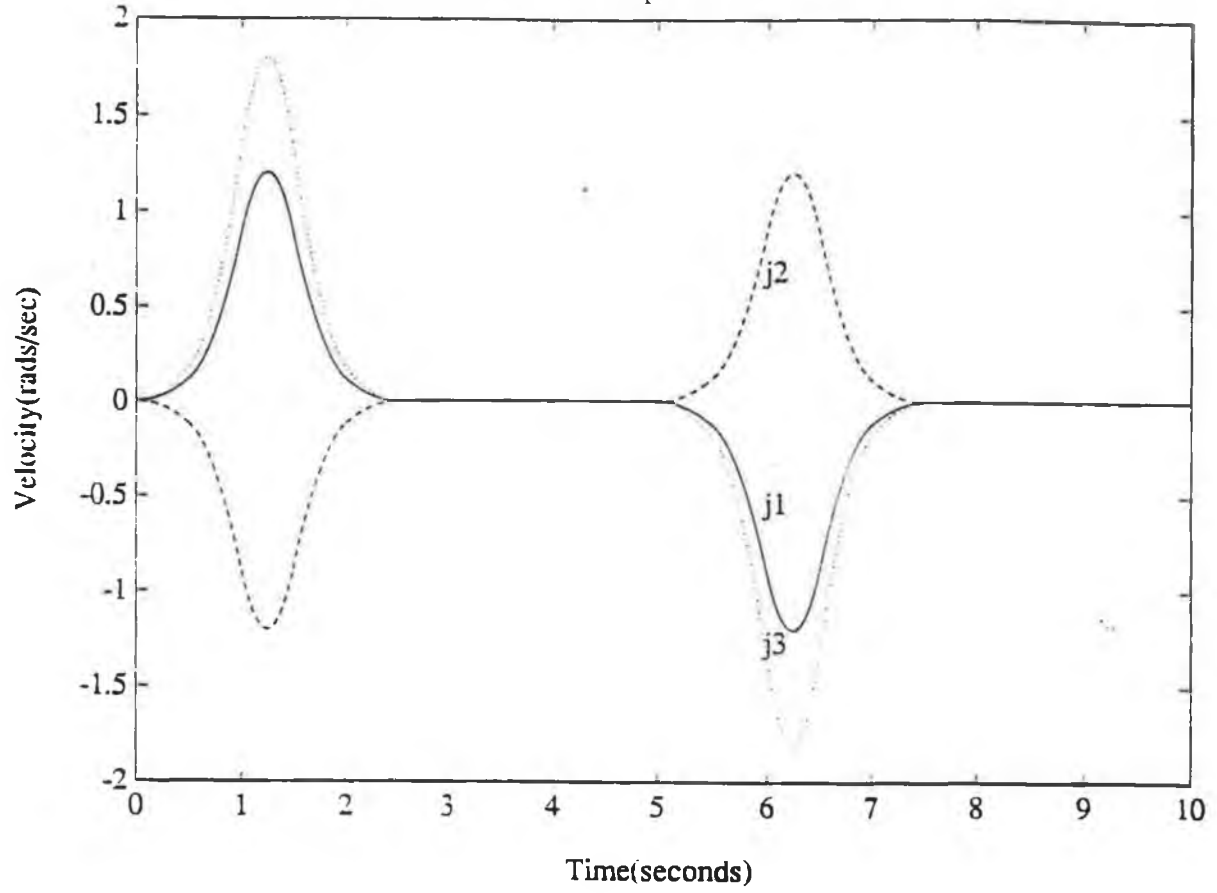
Graph 6.1



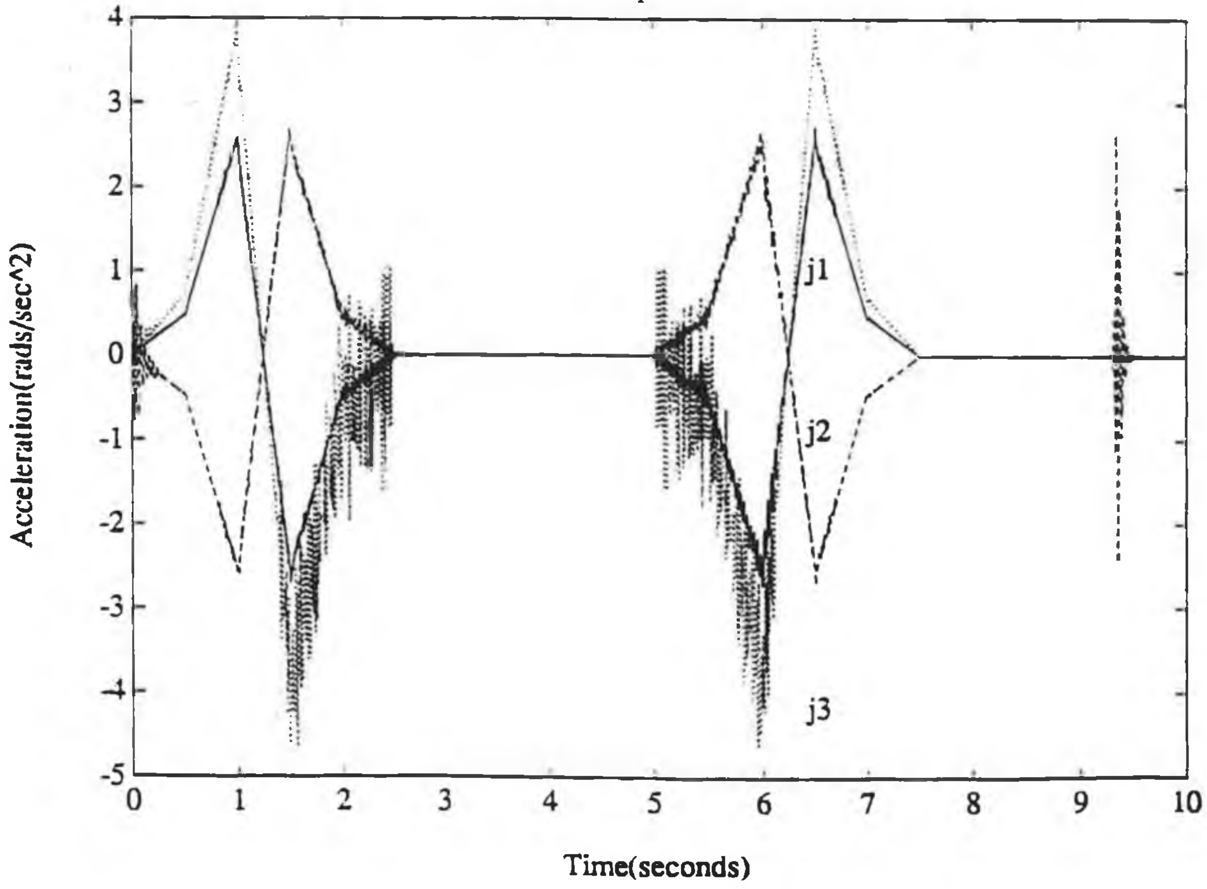
Graph 6.2



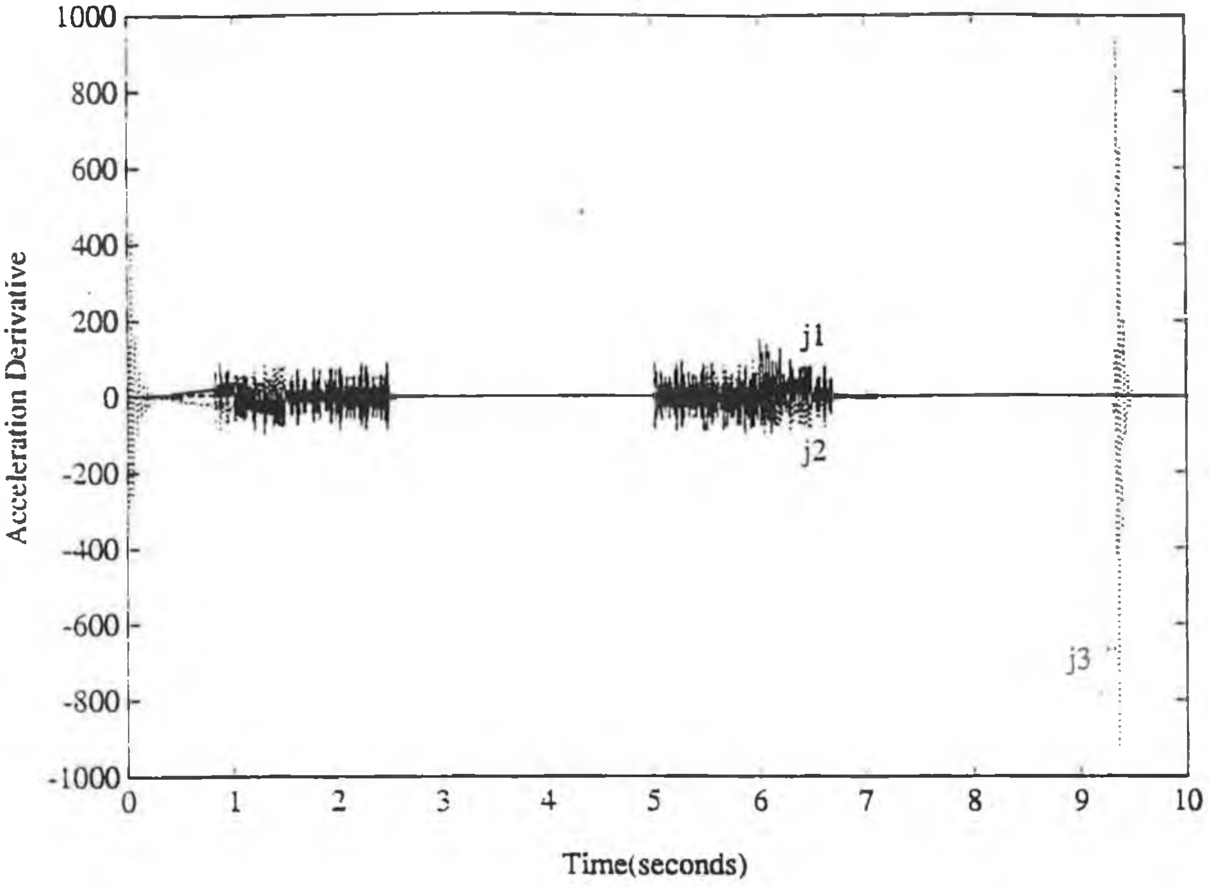
Graph 6.3



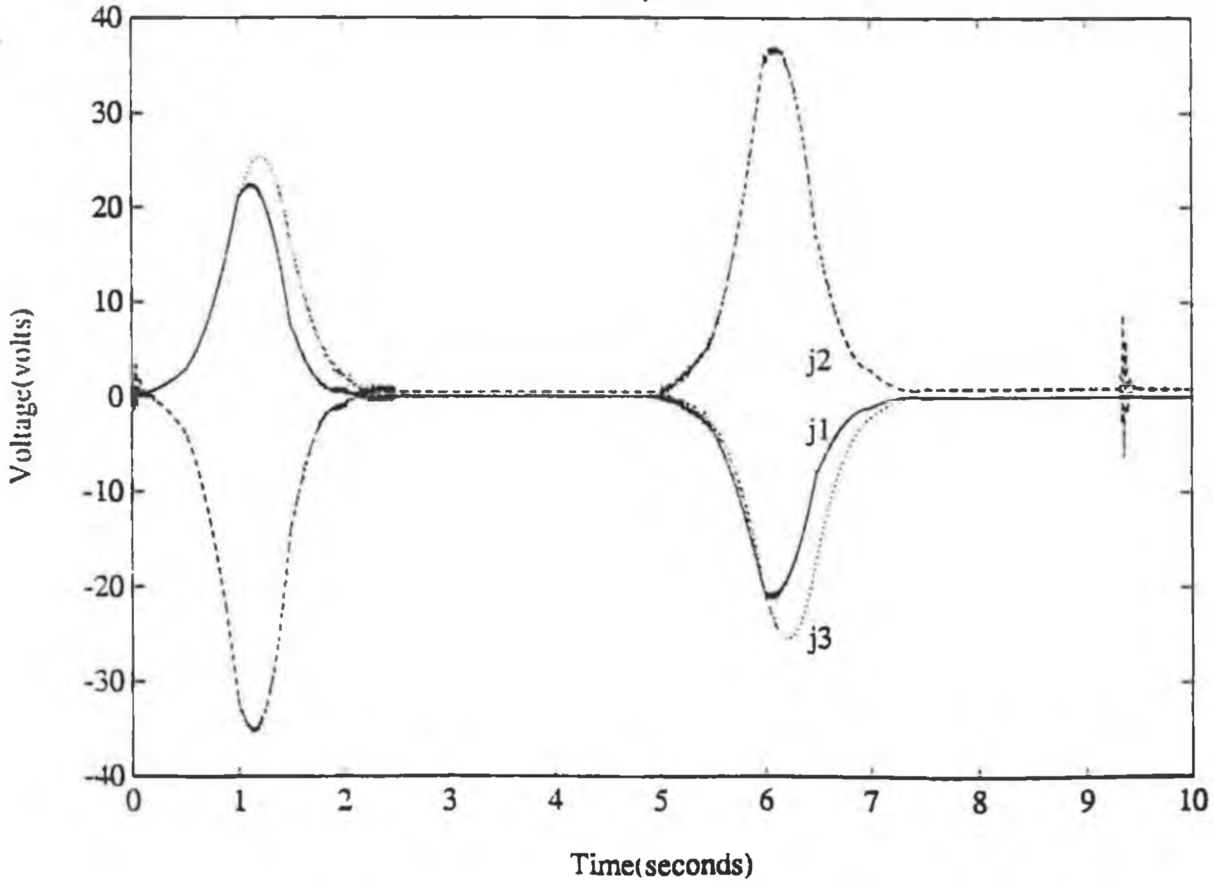
Graph 6.4



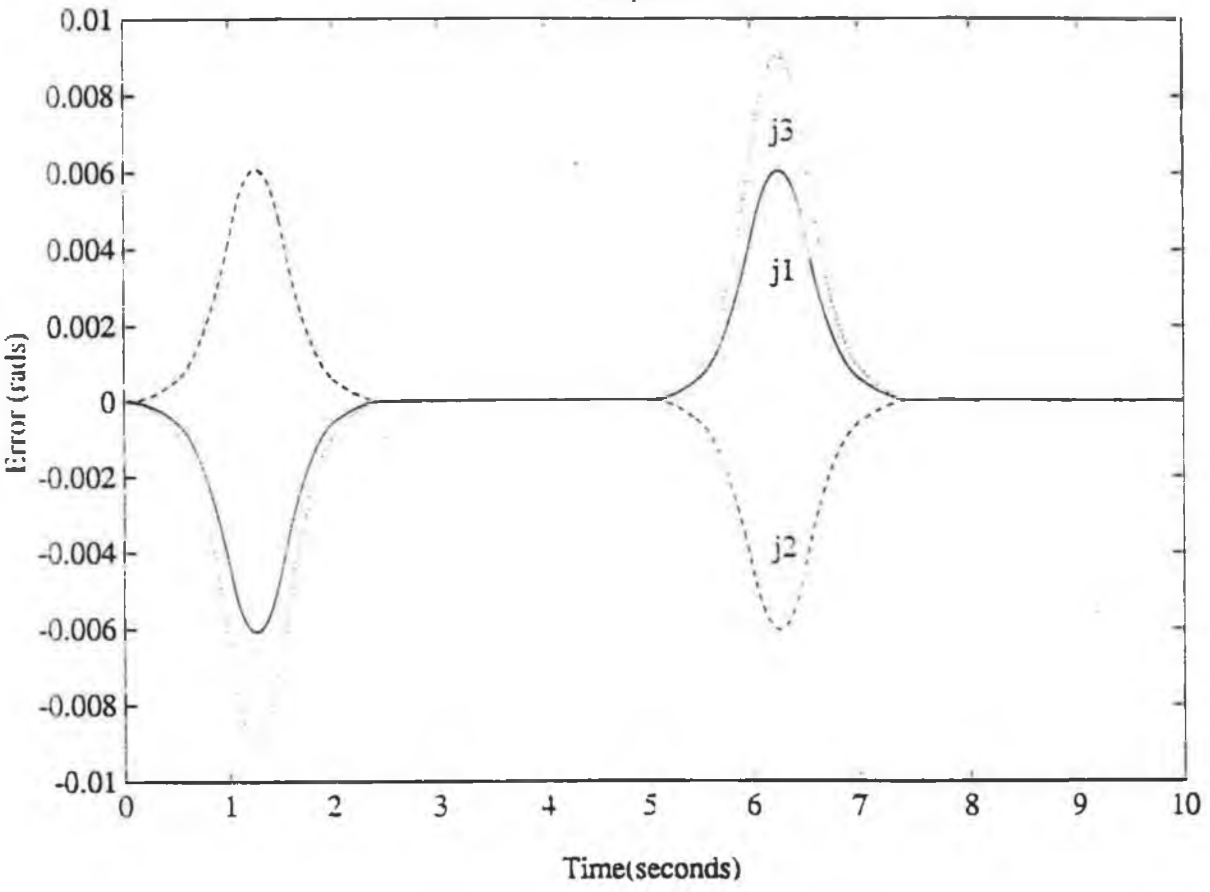
Graph 6.5

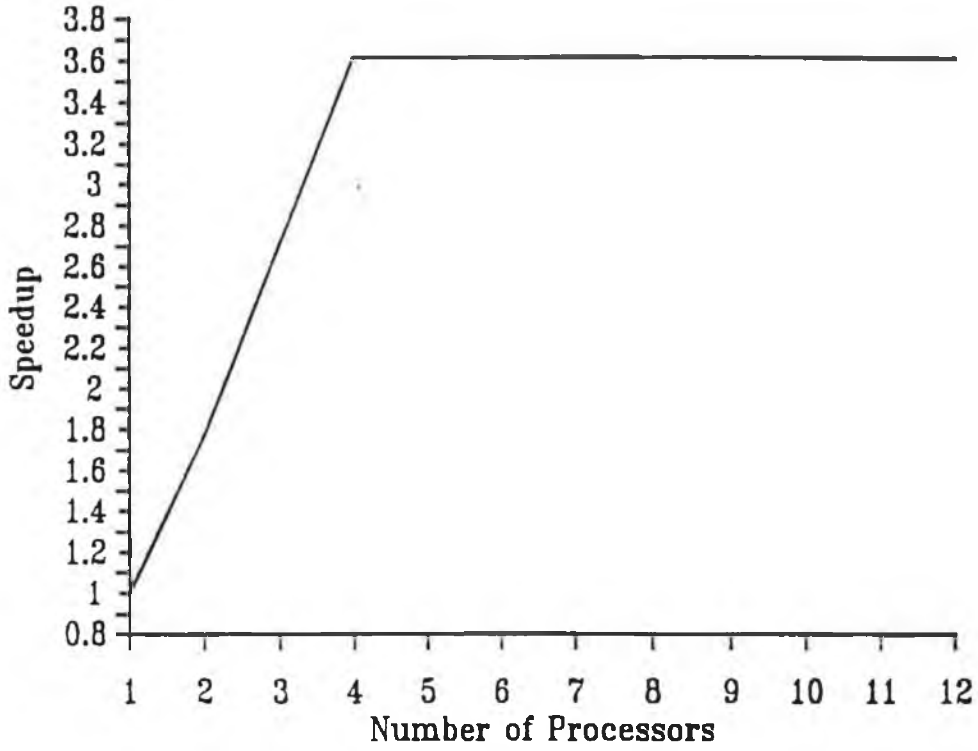


Graph 6.6

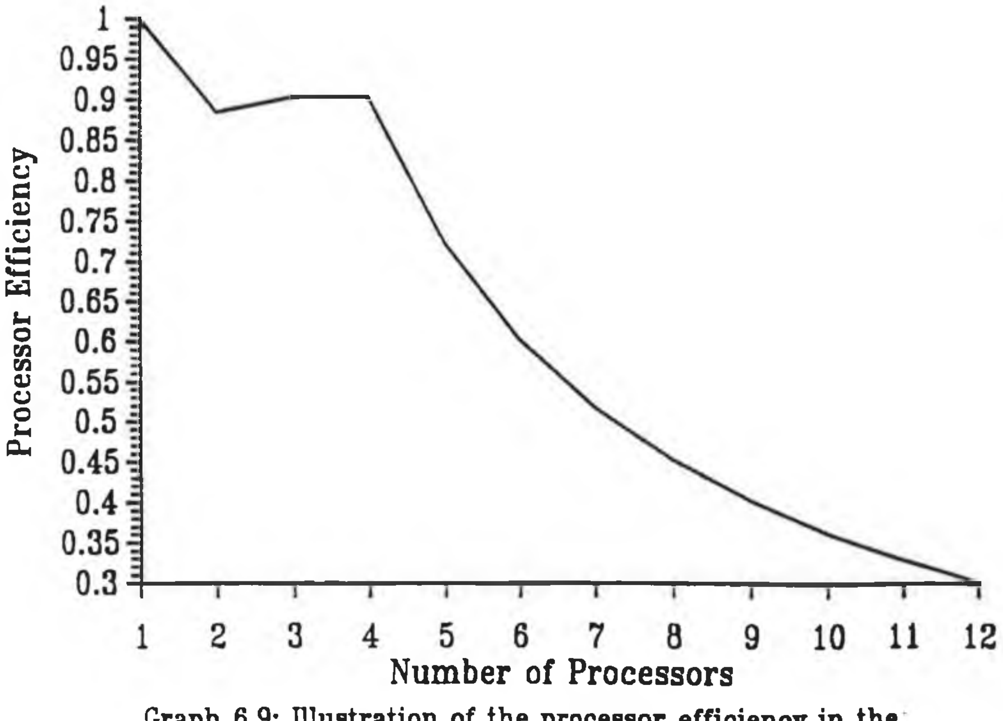


Graph 6.7

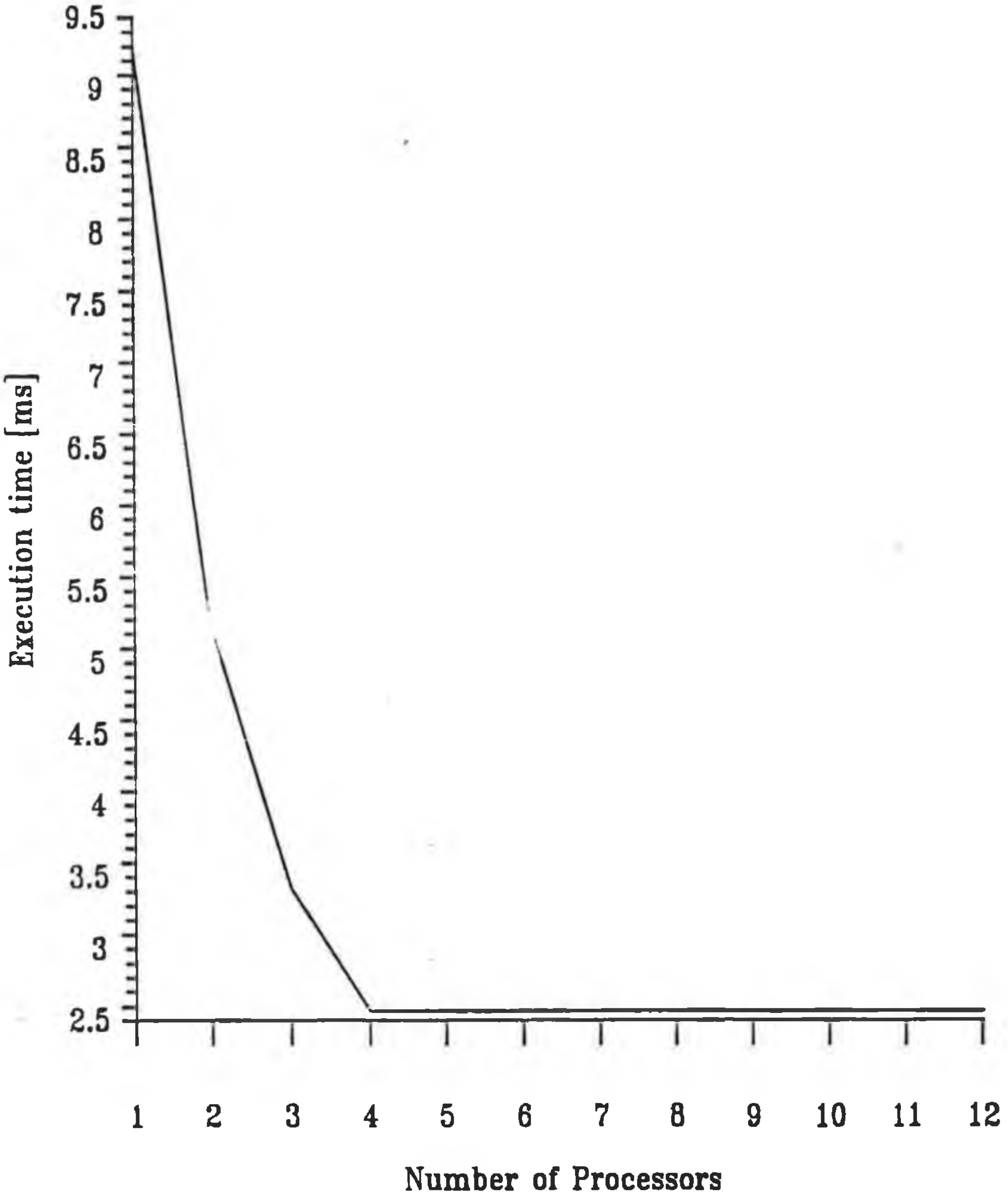

j3
j1
j2
Error (rads)
Time(seconds)
0.01
0.008
0.006
0.004
0.002
0
-0.002
-0.004
-0.006
-0.008
-0.01
0
1
2
3
4
5
6
7
8
9
10



Graph 6.8: Illustration of the speedup of the parallel controller as a function of the number of TRAMS



Graph 6.9: Illustration of the processor efficiency in the parallel implementation of the PUMA 560 controller



Graph 6.10: Execution time for Feedforward/Feedback controller for the PUMA 560 manipulator versus number of Processors