INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600
Parallel algorithms and architectures for the manipulator inertia matrix

Amin-Javaheri, Masoud, Ph.D.
The Ohio State University, 1989
Parallel Algorithms and Architectures for
the Manipulator Inertia Matrix

A Dissertation
Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of the Ohio State University

by

Masoud Amin-Jawaheri, B.S.E.E., M.S.

* * * * *

The Ohio State University
1989

Dissertation Committee:
Professor D. E. Orin
Professor H. Hemami
Professor P. Sadayappan

Approved by:

Adviser
Department of Electrical Engineering
To my Family
ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my adviser, Dr. David E. Orin, for his excellent guidance during my entire Ph.D. program. I truly admire his technical expertise and dedication to his work. I am also deeply grateful to Dr. H. C. Ko, department chairman, for his support and moral guidance throughout my entire graduate work at The Ohio State University. I am also grateful to the entire faculty for making my studies and stay at The Ohio State University for nine years such an enjoyable and invaluable experience. In particular, I wish to thank Dr. K. J. Breeding, Dr. J. G. Gottling, Dr. H. Hemami, and Dr. P. Sadayappan for their valuable comments and discussions during my entire education.

Last, but not least, I wish to thank my parents for their love and support during the entire period of my education and without whose encouragement this work would have never been accomplished.

This work was supported in part by the National Science Foundation, Computer Engineering Grant No. DMC-8312677 and Computational Engineering Grant No. EET-8718434 and by the Department of Electrical Engineering.
VITA

April 20, 1960 ......................... Born - Esfahan, Iran

January 1979 - August 1980 .......... Undergraduate Study,
The George Washington University,
Washington D.C.

April 1982 - June 1983 .............. Undergraduate Research Associate,
ElectroScience Laboratory,
The Ohio State University,
Columbus, Ohio

August 1983 ......................... B.S.E.E., The Ohio State University

Sept. 1983 - Dec. 1985 ............. Graduate Teaching Associate,
Department of Electrical Engineering,
The Ohio State University

December 1985 ....................... M.S.E.E., The Ohio State University

January 1986 - August 1987 ........ Graduate Research Associate,
Department of Electrical Engineering,
The Ohio State University

Sept. 1987 - Present ............... Graduate Teaching Associate,
Department of Electrical Engineering,
The Ohio State University
PUBLICATIONS


FIELDS OF STUDY

Major Field:

Robotics and Computer Engineering: Professor D. E. Orin
Professor K. J. Breeding
Professor H. Hemami

Minor Fields:

Control Systems: Professor H. Hemami
Professor R. E. Fenton

Computer and Information Science: Professor P. Sadayappan

Electronics and Microelectronics: Professor J. G. Gottling
Professor W. H. Cornetet
Professor P. Roblin
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

VITA iv

LIST OF TABLES ix

LIST OF FIGURES x

## I. INTRODUCTION 1

1.1 Introduction ................................................................. 1
1.2 Dynamic Decoupling and Linearization ............................. 5
1.3 Objectives of the Dissertation ........................................... 6
1.4 Organization of the Dissertation ....................................... 9

## II. SURVEY OF LITERATURE 11

2.1 Introduction ................................................................. 11
2.2 Parallel Algorithms ......................................................... 12
  2.2.1 Scheduling of Computations on a Multiprocessor .......... 14
2.3 Parallel Architectures ...................................................... 16
  2.3.1 Task-Level Parallelism ............................................. 18
  2.3.2 Process-Level Parallelism ........................................... 21
  2.3.3 Fine-Grain Parallelism ........................................... 23
2.4 Summary ........................................................................... 27
### III. DEVELOPMENT OF $O(N)$ PARALLEL ALGORITHM MAPPED ONTO SYSTOLIC ARCHITECTURES

3.1 Introduction ....................................   29  
3.2 $O(N)$ Parallel Algorithm for the Inertia Matrix .............. 31  
3.3 Implementation of the Inertia Matrix with the Robotics Processor 40  
3.4 Summary and Conclusions .......................... 55  

### IV. DEVELOPMENT OF $O(\log_2 N)$ PARALLEL ALGORITHM MAPPED ONTO A CUBE ARCHITECTURE 

4.1 Introduction ........................................... 57  
4.2 Development of an $O(\log_2 N)$ Parallel Algorithm for the Inertia Matrix .................................................................................. 59  
4.2.1 Parallel Algorithm for $M_i$ ................................................. 60  
4.2.2 Parallel Algorithm for the Diagonal Elements ............... 64  
4.2.3 Parallel Algorithm for the Off-Diagonal Elements .... 67  
4.3 Number of Processors Required .......................... 70  
4.4 Implementation of the $O(\log_2 N)$ Algorithm on a Cube Architecture ................................................................. 73  
4.5 Summary and Conclusions ................................................... 75  

### V. NON-STRICT COMPUTATIONAL MODELS

5.1 Introduction ........................................... 79  
5.2 An Overview of the $O(N)$ Parallel Algorithm .............. 83  
5.3 Non-Strict Computational Algorithm ................................. 84  
5.4 Simulation Results ............................................. 90  
5.4.1 Non-Strict Results with Zero-Order Output Prediction . 94  
5.4.2 Non-Strict Results with Multiple Rate Computation . 97
# LIST OF TABLES

1. $O(N)$ Parallel Algorithm for Computing the Inertia Matrix .......................... 39
2. Architecture Parameters as a Function of the Number of Processors Used. (Numbers correspond to $N = 8$) .............................................. 53
3. $O(\log_2 N)$ Parallel Algorithm for Computing the Inertia Matrix .......................... 65
4. Minimum Number of Processors Needed in Stage B for $N = 8$ .......................... 72
5. Number of Processors vs. Order of Computation ........................................ 72
6. Architecture Parameters as a Function of the Number of Processors Used. (Numbers correspond to $N = 8$) .............................................. 76
7. Effective Computational Delays with Zero-Order Output Prediction ($t_f = 4 \text{ sec}$). (First row of three – Strict, second row – Zero-Order Non-Strict, third row – First-Order Non-Strict) ........................................ 95
9. Modified Denavit-Hartenberg Parameters for PUMA 560 .................................. 115
10. Link Centers of Gravity for PUMA 560 ......................................................... 115
11. Dynamic Parameters for PUMA 560 ............................................................. 116
12. Joint Angle Limits and Maximum Joint Torques for PUMA 560 .................. 116
**LIST OF FIGURES**

1. Block Diagram of the Robotics Processor. .......................................................... 25
2. Two Composite Rigid Body Modeling of an Open-Chain Mechanism. .................. 34
3. Composite Center of Mass ($c_i$) and Composite Moment of Inertia ($E_i$) for links $i$ through $N$. .......................................................... 35
4. Task Graph for the $O(N)$ Algorithm. .............................................................. 42
5. $N$-Processor Systolic Architecture in Column Form. ....................................... 45
6. Timing Diagram for $N$ Processors in Column Form. ....................................... 46
7. $N$-Processor Systolic Architecture in Row Form. ........................................... 48
8. Timing Diagram for $N$ Processors in Row Form. ........................................... 49
9. $N$-Processor Systolic Architecture in Diagonal Form. ..................................... 50
10. Timing Diagram for $N$ Processors in Diagonal Form. .................................... 51
11. $N(N+1)/2$-Processor Systolic Architecture. .................................................. 52
12. Flow of Data and Computation for Determining $M_1$ ($M_{1,8}$) ....................... 61
13. Flow of Data and Computation for Determining $M_f$ ($M_{f,N}$) ...................... 63
14. Flow of Data and Computation for the Parallel Algorithm ($N = 8$). ............... 68
15. $N$-Processor Cube Configuration ($N = 8$). .................................................. 74
16. Non-Strict Computational Approach Showing the Use of Prediction. ................. 85
18. Various Computational Techniques. ................................................................. 87
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>Trajectory Profile for Joint Angle #1 ($t_f = 4$ sec.)</td>
<td>92</td>
</tr>
<tr>
<td>20</td>
<td>Exact Computation of Joint Torque 1.</td>
<td>93</td>
</tr>
<tr>
<td>21</td>
<td>Joint Torque Error for Joint Torque #1 at $f_s = 72.5$ Hz.</td>
<td>101</td>
</tr>
<tr>
<td>22</td>
<td>Link Frames and Parameter Assignment.</td>
<td>113</td>
</tr>
<tr>
<td>23</td>
<td>PUMA 560 in Zero Position.</td>
<td>113</td>
</tr>
<tr>
<td>24</td>
<td>Flow Chart of the Simulator.</td>
<td>118</td>
</tr>
</tbody>
</table>
CHAPTER I
INTRODUCTION

To you, a robot is a robot. Gears and metal.
Electricity and positrons. Mind and iron!
Human-made! If necessary, human-destroyed.
But you haven't worked with them, so you don't know them.
They're a cleaner, better breed than we are.
- Isaac Asimov

1.1 Introduction

Robots of a decade or more ago were usually relatively simple mechanisms which performed carefully structured tasks with minimal accuracy, repeatability, resolution, and speed. Such mechanisms were often based upon just a few degrees of freedom which could be controlled somewhat independently using very basic control schemes. During this same period, a number of telemanipulator systems, which use the versatility of a human operator in a master-slave mode, were developed for application in less structured environments, in nuclear power plants and space. However, even in these systems, the control system design remained relatively basic.

Over the past decade, the growing demands of robotic applications have steadily increased the sophistication of the mechanisms and tasks considered. At the same time, rapid advancement of computer technology has opened the way
to implement more complex control structures. Thus, as robot manipulators have become mechanically more complex (e.g. flexible manipulators and direct-drive arms) and tasks have become more sophisticated (e.g. increasing levels of planning and sensing in unstructured environment), more advanced control schemes have been proposed.

However, a major challenge in effectively realizing these advanced control schemes for robotic systems yet remains, and this is the difficulty of implementing the kinematic and dynamic equations required for coordination and control, in real time. While the total amount of computation appears to be somewhat less than that of many scientific computations, implementation in real-time implies that these same computations must be repeated at rates of hundreds or perhaps thousands of times per second. This, then, results in an important computational problem in robotics control, and consequently, the computational aspects of dynamic control techniques is the main thrust of this dissertation work.

While the dynamics of the mechanism were not considered in earlier control schemes, the more recent development of model-based control [1,2] includes the dynamics in an integral way. Typically, the control is based on an Inverse Dynamics computation which determines the required actuator torque for a desired system trajectory. In model-based control, accurate modeling of the dynamic parameters of the plant is very crucial for dynamic stability. Equally important is the rate at which the dynamics may be computed.

In addition to computation of Inverse Dynamics, implementation of a number of control schemes is based upon computation of the inertia matrix so as to decouple the dynamics along the several axes of the manipulator [3,4,5]. This allows either linear or nonlinear control schemes to be more effectively applied. The inertia matrix basically relates the joint accelerations to the joint torques, but a more
formal definition for it is given in the next section which also outlines its use for
decoupling the dynamics and linearizing the response.

Other applications in which the inertia matrix has been explicitly used include
surface tracking and object identification using force control [6] and computation
of the collision effects between a manipulator and its environment or between two
manipulators in a shared work space [7].

The inertia matrix is also an integral part of the computations required to
simulate robotics mechanisms [8,9,10]. While simulation has been considered in
the past to be an off-line problem so that fast computation was not critical, there
are now several applications in which real-time simulation is important. For in-
stance, because of transmission delays in supervisory control for space telerobotics
applications, real-time simulation is crucial in graphically depicting the response
before the actual command is transmitted [11]. Thus, computation of the inertia
matrix is important for real-time simulation as well.

Inverse Dynamics has generally been implemented using either a recursive
Newton-Euler or Lagrange-Euler formulation since explicit determination of the
terms of the dynamic equations of motion, given in the next section and which in-
cludes determination of the inertia matrix, is not computationally efficient. How-
ever, if multiple rates for the computation of the individual terms are considered,
it may be that the latter approach, which involves explicit computation of the in-
ertia matrix, will be the most efficient. In particular, while the computation of the
inertia matrix is intensive, the inertia matrix itself is slowly time-varying, so that
its computation may be performed at a much lower rate than the computation of
some of the other terms. In any event, computation of the inertia matrix is cen-
tral to many problems, and work is needed which is directed to its computational
aspects. Again, this is the main thrust of this dissertation work.

3
As mentioned previously, while the computational requirements of advanced control schemes for robotics continues to increase, computational power is also increasing, especially in parallel computer systems such as the NCUBE hypercube [12], BBN Butterfly, and Carnegie Mellon systolic array Warp computer [13]. However, to obtain the desired speedup on such machines, corresponding parallel algorithms need to be developed to fully and effectively utilize the computing resources. For a specific parallel system, the algorithms must be mapped onto the architecture while considering issues related to numbers of processors required, assignment of processes to processors, scheduling, communication overhead, and synchronization.

There are, however, several problems associated with present general-purpose parallel systems. In general, such systems may not be able to exploit the fine-grain parallelism inherent within the primitive robotics operations at the matrix-vector level. This results because of the significant overhead associated with interprocessor communication, especially in the case of the hypercube. An alternative is available to overcome the communications problems so that fine-grain parallelism is exploited, and that is the use of special-purpose architectures. In particular, Robotics Processor chips have been designed at The Ohio State University to be configured in systolic arrays to circumvent the problems associated with general-purpose computer systems [14,15]. In addition to developing parallel algorithms for the inertia matrix in this dissertation, parallel architectures based on the Robotics Processor will also be considered.

The major objective of this dissertation, then, is to investigate the computation of the manipulator inertia matrix and to devise parallel algorithms and architectures to achieve a real-time response. Further details of this objective are given in Section 1.3.
1.2 Dynamic Decoupling and Linearization

The general dynamic equations of motion for a single N-degree-of-freedom open-chain manipulator, in joint space with physical constraint forces and moments between the links eliminated, may be represented in the following form:

\[ \mathbf{T} = \mathbf{H}(\mathbf{q})\mathbf{\ddot{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) + \mathbf{J}^T(\mathbf{q})\mathbf{f} \]  

(1.1)

where:

- \( \mathbf{T} \) is an \( N \times 1 \) vector of joint torques (forces),
- \( \mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}} \) are \( N \times 1 \) vectors of joint positions, rates, and accelerations,
- \( \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) \) is an \( N \times N \) matrix of coriolis and centrifugal force terms,
- \( \mathbf{G}(\mathbf{q}) \) is an \( N \times 1 \) vector of gravitational forces,
- \( \mathbf{J}(\mathbf{q}) \) is a \( 6 \times N \) Jacobian matrix,
- \( \mathbf{f} \) is a \( 6 \times 1 \) vector of external forces and moments exerted by link \( N \),

and \( \mathbf{H}(\mathbf{q}) \) is the \( N \times N \) symmetric, positive-definite, inertia matrix that is of prime interest to compute.

One of the most effective schemes for manipulator control is the computed torque technique [1,2,16,17] which computes the dynamics on-line, using the sampled joint position and velocity data. The dynamic equations of motion in (1.1) may be rewritten for use in control as

\[ \mathbf{T}_c = \mathbf{H}_c(\mathbf{q})\ddot{\mathbf{q}}_c + \mathbf{C}_c(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{G}_c(\mathbf{q}) + \mathbf{J}^T_c(\mathbf{q})\mathbf{f} \]  

(1.2)

where the subscript \( c \) denotes computed and

\[ \ddot{\mathbf{q}}_c = \ddot{\mathbf{q}}_d + \mathbf{K}_v(\dot{\mathbf{q}}_d - \dot{\mathbf{q}}) + \mathbf{K}_p(q_d - q) \]  

(1.3)

The subscript \( d \) denotes the desired values for position, velocity, and acceleration, and the matrices \( \mathbf{K}_v \) and \( \mathbf{K}_p \) are \( N \times N \) derivative and position feedback gain matrices, respectively, for an \( N \) degree-of-freedom manipulator.
If the computed torque $T_c$ in Eq. (1.2) is applied to the manipulator so that it is equal to $T$ in Eq. (1.1) and assuming that the inertia, coriolis/centrifugal, and gravity terms are computed without error so that $H_c(q) = H(q)$, $C_c(q, \dot{q}) = C(q, \dot{q})$, and $G_c(q) = G(q)$, then it follows that

$$H(q) \left[ \ddot{e}(t) + K_v \dot{e}(t) + K_p e(t) \right] = 0 \quad (1.4)$$

where $e(t) \triangleq q_d(t) - q(t)$, $\dot{e}(t) \triangleq \dot{q}_d(t) - \dot{q}(t)$, and $\ddot{e}(t) \triangleq \ddot{q}_d(t) - \ddot{q}(t)$. Since $H(q)$ is positive definite and if $K_v$ and $K_p$ are chosen to be diagonal, then the response for each joint is decoupled from the others and is second order. Further, with appropriate values for $K_v$ and $K_p$, the characteristic roots (eigenvalues) of Eq. (1.4) will have negative real parts, and hence, the error vector $e(t)$ will have a linear response which approaches zero asymptotically. This then shows the importance of computation of the inertia matrix for decoupling the dynamics and linearizing the response.

1.3 Objectives of the Dissertation

Determination of the inertia matrix involves a considerable amount of computation (approximately equal to that of Inverse Dynamics for a 6 degree-of-freedom manipulator) which varies as $O(N^2)$ on a single processor, where $N$ is the number of degrees of freedom of the manipulator. The major objective of this dissertation is to develop efficient parallel algorithms and architectures to provide additional levels of speedup over the most efficient serial approaches.

To achieve a higher throughput and speedup, the order of computation needs to be reduced. To do so, $O(N)$ parallel algorithms for computing the manipulator inertia matrix will first be developed by assigning computation of individual rows, columns, etc. to separate processors. To further reduce the order of computation, a
parallel algorithm with a lower bound computation time of $O(\log_2 N)$ will be developed. The algorithm will be based on the most efficient serial algorithm which uses the composite rigid body method [8]. To develop the $O(\log_2 N)$ parallel algorithm, computation will be divided into $O(\log_2 N)$ levels where in the first level the composite rigid body parameters of sets of two links are computed, in the second level the composite rigid body parameters of sets of four links are computed, etc. This computational approach is referred to as recursive doubling [18] and will be used to reformulate the linear recurrence equations required to compute the diagonal elements of the inertia matrix. Computation of the off-diagonal elements involves $N$ linear recurrences of varying-size and a new method, which avoids redundant computation of position and orientation transforms for the manipulator, will be developed. Computation of both the diagonal elements as well as the off-diagonal elements are of $O(\log_2 N)$ resulting in an $O(\log_2 N)$ algorithm. Combinations of the $O(\log_2 N)$ algorithm for the diagonal elements and $O(N)$ algorithm for the off-diagonal elements will also be explored.

Parallel systolic architectures to implement the proposed algorithms will then be devised based upon a VLSI Robotics Processor [14] to exploit much of the inherent fine-grain parallelism that exists in the computation of the inertia matrix. In so doing, I/O time and idle time due to processor synchronization as well as CPU utilization and on-chip memory size will be fully included in the evaluation. Several configurations will be compared and the feasibility and effectiveness of the designs will be indicated.

The foregoing algorithmic and architectural approaches proposed will strictly adhere to the precedence relationships which exist within the recurrence equations needed to compute the inertia matrix. That is, the precedence relationships among the processes will dictate the flow of computation. However, a new computational
model which relaxes interprocess precedence will be considered. If relaxation of precedence is carefully applied, it may be used to break the serial flow inherent in the algorithm so that a greater degree of parallelism may be achieved. Since the computational precedence will not be adhered to strictly, the approach is referred to as Non-Strict Computation [19]. Thus, the objective of the Non-Strict Computational Approach is to increase parallelism by relaxing precedence among computational nodes (processes), and it will be applied to the inertia matrix. Implementation of such an algorithm on a parallel architecture should certainly result in a shorter critical path from input to output, and consequently, in lower effective computational delays and in higher throughput.

The major problem with the Non-Strict Computational Approach is that the results are corrupted because strict precedence is no longer followed. To ensure the integrity of the results and a close correlation between the strict results and the non-strict results, the freshest input data or a linear combination of the previous input data may be consumed by the processes to compute their intermediate or final results. That is, the values for the inputs may be predicted forward in order to initiate the computation sooner. Again, this should increase the process-level parallelism, and subsequently, it should result in a greater speedup through lower effective delays. In this context, higher orders of prediction to reduce the error introduced in the results will be explored.

If the inertia matrix is used for the computation of Inverse Dynamics using the dynamic equations of motion, as was alluded to earlier, then the computation of the inertia matrix may be performed at a different rate than the computation of some of the other terms. This concept is referred to as multiple rate computation, and it will be applied to compute the acceleration-dependent term in Eq. (1.1). Since the inertia matrix is slowly time-varying, it is expected that its computation
may be performed at a much lower rate than the computation of the joint torques. If this is so, the processors computing the inertia matrix will be idle much of the time and available for computing other tasks (such as the Jacobian matrix in Eq. (1.1)). Work will also be directed, then, to explore the effectiveness of multiple rate computation to increase the CPU availability.

1.4 Organization of the Dissertation

The following chapter presents a study of the previous work conducted on parallel algorithms and architectures for real-time manipulator dynamic control. First, parallel algorithms developed for Inverse Dynamics and the Inertia Matrix are discussed. Then, scheduling of computations on a multiprocessor is discussed addressing such issues as exploitation of parallelism, synchronization, and communication. Finally, parallel architectures that have been proposed for manipulator control are outlined under three general categories – those exploiting task-level parallelism, process-level parallelism, and fine-grain parallelism.

Chapter III presents the development of an $O(N)$ parallel algorithm for computing the manipulator inertia matrix. Various systolic architectures, one- and two-dimensional arrays, are proposed to implement the proposed algorithm in real time. All proposed architectures are compared based on several significant performance parameters. At the heart of each architecture is a VLSI-based Robotics Processor to exploit fine-grain parallelism inherent in the computation.

Chapter IV presents the development of an $O(\log_2 N)$ parallel algorithm based on the recursive doubling technique for computing the inertia matrix. The development first explains the application of recursive doubling to compute the composite mass of a multi-link manipulator, in parallel. Then, the concept is generalized to compute the diagonal and upper off-diagonal elements of the inertia matrix. A
cube architecture is employed to implement the algorithm. This architecture is compared with the systolic architectures presented in Chapter III.

Chapter V presents the development of the new computational model, the Non-Strict Computation Approach [19]. The Non-Strict Computational Algorithm, based on the $O(N)$ parallel algorithm presented in Chapter III, is developed to relax precedence and thus to achieve a high degree of parallelism. In so doing, the results are corrupted. In order to ensure the integrity of the results, prediction is used to reduce the error introduced. The effect of the degree of prediction to decrease the severity of data corruption is investigated in the chapter. Multiple rate computation of the acceleration-dependent term of the dynamic equations of motion to increase CPU availability is also studied. Simulation results based on the PUMA 560 are given.

Finally, in the last chapter, the entire work is summarized and appropriate conclusions are made. Since the investigations of this dissertation have lead to a number of other possible research problems, directions as to possible future research work are outlined.
CHAPTER II
SURVEY OF LITERATURE

2.1 Introduction

The computational aspects of real-time, dynamic control schemes has been a problem under consideration for some time. A number of justifications for dynamic control are outlined in [20] with specific examples of control of locomotion and manipulation robots given. Toward this end, various sophisticated control schemes have been proposed for robotic mechanisms [21]-[26].

The computational aspects of dynamic control schemes may be addressed through any of the following four approaches to achieve higher performance [27].

- Faster circuit technology,
- Optimizing and vectorizing compilers,
- Parallel algorithms, and
- Parallel architectures implementing new computational models.

It is believed that optimal solutions toward achieving an order of magnitude higher performance is attainable through proper employment of all or some of the above techniques combined. The optimality of the solution depends upon two time-varying parameters: system requirements and technological breakthroughs (e.g. VLSI technology). Thus, the solution must be constantly updated to maintain optimality.
In this chapter, an extensive survey of previous work completed on parallel algorithms and architectures to exploit parallelism in dynamic control schemes is presented. Specifically, the computation of the Inverse Dynamics problem and the manipulator inertia matrix used in linear and nonlinear dynamic control schemes are of particular interest. Toward this end, first, related parallel algorithms and their scheduling policy on parallel architectures are discussed. Then, parallel architectures previously proposed for implementing Inverse Dynamics and the inertia matrix for single open-chain systems are presented.

2.2 Parallel Algorithms

The general dynamic equations of motion for a single \( N \)-degree-of-freedom open-chain manipulator, in joint space with no physical constraint forces and moments between the links, was given in Eq. (1.1). For a desired set of joint position, velocity, and acceleration vectors for a specified trajectory and for given external forces and moments, the required torques to be applied by the actuators may be computed from Eq. (1.1). This is the problem of Inverse Dynamics which computes the actuating signals for a given velocity and acceleration profile. Eq. (1.1), however, is not usually explicitly used to compute Inverse Dynamics. Instead, linear recursive algorithms based on the Newton-Euler [26,28,29] and Lagrangian [30,31] formulations, which do not require computation of the individual terms of Eq. (1.1), have been developed. The best of the resulting algorithms are of \( O(N) \), where \( N \) is the number of degrees of freedom of the manipulator.

A comparative analysis of the computational requirements with multiple sets of processors for both the Newton-Euler and Lagrange-Euler formulations was presented by Khosla and Ramos [32]. They concluded that the Newton-Euler formulation is computationally more efficient than the Lagrange-Euler formulation,
as it had been believed previously. So, most of the algorithms and architectures discussed here are based on Newton-Euler dynamics.

Parallel algorithms based upon recursive doubling [18] have also been developed to reduce the order of computation to $O(\log_2 N)$ [33,34]. In [33], Lathrop achieved an $O(\log_2 N)$ parallel algorithm through a logarithmic recursion method. It was derived through a restructuring of the fundamental computational framework established for the $O(N)$ algorithm. In [34], Lee and Chang showed that the time lower bound for computing the Inverse Dynamics of an $N$-link manipulator using $p$ processors is $O(k_1[N/p] + k_2[\log_2 p])$, where $k_1$ and $k_2$ are constants.

The inertia matrix has also been employed in some advanced control schemes to attain good performance [3,4,5]. The inertia matrix is used to decouple the dynamics along the several axes of the robot manipulator. This allows either linear or nonlinear control schemes to be more effectively applied. Determination of the inertia matrix involves a considerable amount of computation (approximately equal to that of Inverse Dynamics for a 6 degree-of-freedom manipulator) which varies as $O(N^2)$ on a single processor [8]. In [8], the most efficient serial algorithm for computing the Inertia Matrix was developed and is based upon the composite rigid body method.

Parallel computation of the manipulator inertia matrix has been considered in the context of computing robot forward dynamics for real-time computer simulation [9,35]. In [9], parallel linear recurrence algorithms based on the recursive doubling technique [18] were employed to compute the diagonal elements of the inertia matrix. A modified row-sweeping algorithm was then applied to compute the off-diagonal elements. In [35], two virtual algorithms based upon the composite rigid body spatial inertia were presented. Spatial notation, developed by Featherstone [36], was used to formulate the problem. A time lower bound of
$O(\log_2 N)$ was achieved. However, the number of processors needed to implement the $O(\log_2 N)$ algorithm and the interconnection cost were prohibitive, and this resulted in unrealizable implementations. Thus, the virtual algorithms were then mapped onto a linear array of processors to obtain a more realizable architectural configuration. This resulted in an $O(N)$ algorithm.

2.2.1 Scheduling of Computations on a Multiprocessor

Task scheduling is usually the mechanism needed to bridge the gap between algorithms and architectures. To optimally utilize any inherent parallelism in parallel algorithms and architectures, a balanced partitioning and distribution of tasks among processing elements is a must. Normally, an optimal solution is obtained when the development of algorithms and design of corresponding architectures are considered together. Then, a scheduler is required to partition and distribute the tasks evenly among processing elements.

Parallelism and scheduling-and-synchronization overhead are the two most critical parameters affecting the overall system performance. A high degree of parallelism results in a higher system performance and also results in a higher scheduling-and-synchronization overhead [27]. The objective is to maximize parallelism and minimize overhead. Higher parallelism is achieved through task partitioning and load balancing among processors. In so doing, scheduling-and-synchronization overhead tends to escalate. Thus, parallelism and scheduling-and-synchronization overhead are directly related to the granularity of nodes in the control flow graph. To exploit fine granularity, a time penalty for scheduling each node and for synchronizing each arc connecting nodes must be paid. The time penalty is simply a major fraction of the overall execution time of a given application program. On the other hand, nodes may be merged and thus arcs may be
reduced to lower scheduling-and-synchronization overhead. In this case of coarse granularity, not all inherent parallelism which was available originally, would be exploited.

Scheduling may be either static or dynamic. In static scheduling, tasks are assigned to processors during the algorithm development by the user or at compile time by the compiler. In this scheduling strategy, there is always the possibility of not efficiently utilizing the resources at run time. On the other hand, dynamic scheduling allocates tasks to processors at run time. This strategy achieves a higher processor utilization at the expense of additional scheduling time. A comprehensive and comparative study of the performance of the two scheduling methods is conducted in [37].

With regard to specific applications in robotics, Luh and Lin employed a branch-and-bound strategy [38,39] to schedule the recursive Newton-Euler formulation of the Inverse Dynamics computation of an \( N \)-degree-of-freedom manipulator on \( N \) linearly connected processors [40]. Each processor was assigned to compute the necessary computations needed for each link. Parallelism was explored at the matrix operation level. The branch-and-bound strategy and thus the proposed scheduling technique is characterized by four procedures: a branching rule, a selection rule, an elimination rule, and a termination condition. These four processes together are serial in nature; however, every rule may be processed in parallel.

Kasahara and Narita [41] employed a fast heuristic depth-first search of the task graph to create a feasible schedule that could also be used as a bound for a more exhaustive branch-and-bound strategy for the optimal scheduling. Recently, in [42], a heuristic algorithm called Dynamical Highest Level First/Most Immediate Successors First to obtain a fast, but suboptimal scheduling technique was
presented. Then, the $A^*$ search algorithm [43] based upon a heuristic function derived from the Fernandez and Bessell bound [44] is used to minimize the cost of the graph search for the robot inverse dynamics computation. The latter three papers ignored data communication among processors by assuming a high level of granularity for the tasks. More recent studies on task allocation have focused on minimizing communication cost and processor load-balancing [15,45]. The limited degree of communication causes some processors to work on tasks that a better interconnection would eliminate. In [46,47], investigations on parallel branch-and-bound algorithms based on depth-first search on a general-purpose network architecture with limited memory space and slow interprocessor communication are conducted.

2.3 Parallel Architectures

In the last few years, much attention has been directed towards multiprocessor architectures to achieve higher performance. The main reasons for this technological attraction are the feasibility of implementation due to maturity of VLSI technology and an order of magnitude speedup factor that is achievable through exploiting parallelism. Depending on the application or algorithm, an appropriate parallel architecture may be designed to exploit parallelism at various levels. In [48], Patton had categorized various levels of parallelism in multiprocessor computer systems as follows,

- Job-level execution,
- Task-level execution,
- Process-level execution,
- Instruction-level execution,
• Register-transfer level, and

• Logic level.

For robotics applications, a job may be exemplified by coordination of multiple robots performing arc welding, spray painting, material handling, etc. Multiprocessor and multicomputer based systems are possible configurations for job-level execution. Robotics tasks may be exemplified by computation of Inverse Dynamics, the Inertia Matrix, and the Jacobian as blocks of computations within a linear or nonlinear control system. For robotics applications, this level is perhaps the most appropriate level for fault tolerance and distributed processing. Robotics processes may be exemplified by the forward recursion needed in the computation of Inverse Dynamics, or in a general way, the computations needed for each link. At this level, the general format of the machine is defined. For instance, SIMD (single instruction, multiple data stream) or MIMD (multiple instruction, multiple data stream) machines may be proposed to exploit process-level parallelism. Once the machine is defined, instruction-level parallelism may be explored by, for example, a pipelined execution, interconnection scheme, etc. At a much finer level, parallelism may be explored at the register-transfer level by incorporating wider and/or multiple busses, multiple registers, etc. Finally, at the logic level, parallel bit operations (additions or multiplications) may be substituted for serial bit manipulation.

Previous work accomplished on parallel architectures in robotics have attempted to exploit parallelism at three distinct levels: task level, process level, and fine-grain level. The next three sections are devoted to these three approaches taken to exploit the inherent parallelism in robotics applications.
2.3.1 Task-Level Parallelism

Various special-purpose architectures based upon special-purpose and general-purpose processing elements have been devised. Such architectures minimize scheduling-and-synchronization overhead by considering the control graph to have coarse granularity. One such architecture proposed in [49] is a multiprocessor system to control the OSU Hexapod vehicle. Five LSI-11 processors are fully connected through eight-channel parallel line units (PLU) in a loosely-coupled configuration. Various interconnections such as the tree and ring are examined and fault-tolerant features are added to automatically reconfigure the faulty unit out of the computational circle without completely halting the system.

Another multiprocessor system developed to control the OSU Hexapod in real-time is proposed in [50]. The system is based on four Intel 8086 microprocessors connected in a loosely coupled manner of a 4-bus architecture. The processors may be configured to compute different tasks or to compute the same task redundantly. The latter is a fault-tolerant configuration which is combined by a hardware-software voting scheme to detect and correct faulty units within the system.

A distributed multiprocessor architecture based on five single-board 68000 microprocessors with dual ported RAM was designed to control the four-finger Utah-MIT hand [51]. The processors reside on a single bus, the Multibus. One processor is dedicated to one finger; the fifth processor coordinates the movement of the hand. A VAX 11/750 is used as a file server and host for software development and data transfer in and out of the processors using a DMA (direct memory access) link. The hand is interfaced to the Multibus through data acquisition boards. The message passing technique is used to exchange data among tasks running on
separate processors. The latest version of the architecture incorporates a VME bus plus 68020 microprocessors and a SUN 3/280 as a host [52].

At Stanford University, a system called NYMPH was developed to exploit parallelism at the task execution level [53]. It is comprised of seven NSC32016 single-board computers connected to a 68000 based SUN workstation host computer. All the processor boards reside on a single Multibus. A hand or arm is connected through a controller to the same communication bus via analog-to-digital converters and parallel I/O adaptors. The 32016 boards run at 10 MHz and each board includes 32 K bytes of ROM, 512 K bytes of RAM, a 32081 floating-point coprocessor, an 8255 PPI (Programmable Peripheral Interface), and two serial ports. A floating-point multiply can be performed in 6 $\mu$sec. An Ethernet link (10 MB/second bandwidth) interfaces the SUN computer to a VAX computer which is used for program development. The system was designed around three major attributes: large computational power, large amounts of memory, and graphics capabilities to develop an operating system for multiprocessor systems with a user friendly interface. The system emulates a distributed system with the 68000 being the server and the 32016 processors being the clients. Thus, communication among processors (i.e. I/O requests and proper interprocessor handshaking) is coordinated by the 68000 processor in conjunction with the V kernel. Mainly, message passing is used for interprocessor communication and synchronization is performed through two V kernel primitives: signal() and wait().

This architecture had been applied to two different applications. It had been used to control the Stanford/JPL hand having 3 fingers with 3 degrees of freedom per finger. The other application was the development of an experimental programming system called COSMOS designed to facilitate experiments in manipulator position and force control.
In [54,55], the authors propose a 3-bus architecture based on a special-purpose Robotics Processor used to exploit fine grain parallelism. The processor will be discussed later. In this design, problems such as data transfer and real-time operation are dealt with. A 32-bit data bus is provided to obtain sufficient accuracy and resolution. Each Robotics Processor is assigned to distinct tasks such as Inverse Dynamics, Inverse Kinematics, etc. A host computer and data acquisition processing boards are interfaced to the Robotics Processors through a VME bus. Each Robotics Processor is interfaced to 5 dual-axis servo controllers via a private bus. The servo controllers are interfaced to 5 interface cards through another private bus. Processors are configured as master and slaves to simplify bus communication.

The host computer performs global coordination and path planning, provides interfaces to other systems, supports software development and debugging, and downloads programs to Robotics Processor(s). Sensory boards process sensory data and send sensory information to the Robotics Processor(s). The Robotics Processors input servo controller programs from the host and download them to the servo controllers, evaluate forward and inverse kinematic solutions, evaluate inverse dynamic equations, perform trajectory interpolation, broadcast data (i.e. torques, gains, and commands) to some or all servo controllers, and read feedback parameters from the servo controllers. Servo controllers evaluate the control laws, change control laws in real time, read feedback parameters from interface cards, vary update rates in real time, and output actuator torques to interface cards. Finally, interface cards convert the servo controller torques to analog form, read feedback sensors and convert the data to digital format, pass feedback information to servo controllers, and process index pulses for homing.

A Digital Signal Processor (DSP) based multiprocessor architecture called SPARTA has been developed at IBM [56]. The processors reside in an IBM PC on
the PC bus for interprocessor communication. The PC is used for runtime software
development and support. Digital Signal Processors exploit parallelism at a low
level; whereas, the overall architecture was designed to achieve real-time robot
control by distributing tasks among various DSP's. A similar architecture based
on a four-node 32-bit processor configuration has been proposed to control the
University of Minnesota direct drive arm [57]. The processors are fully connected
and reside in an IBM PC/AT computer.

2.3.2 Process-Level Parallelism

Various architectures have been proposed to compute Inverse Dynamics and
to exploit parallelism at the process execution level. In [58], parallel pipelined
multiprocessor architectures based upon an off-the-shelf processing element are
used to map the serial Newton-Euler and Lagrange-Euler formulations. The serial
algorithms are decomposed and then mapped onto the pipelined architecture. The
number of pipelined stages are a function of the number of algorithm decomposition
levels. The proposed architecture which implements the decomposed Newton-
Euler formulation offers a higher throughput than the architecture implementing
the Lagrange-Euler formulation.

Concurrent computation of Inverse Dynamics was attempted at the Oak Ridge
National Laboratory on a hypercube ensemble machine [59]. The machine is a
Multiple Instruction Multiple Data (MIMD) multiprocessor connected in a binary
d-dimensional cube topology using fully asynchronous bidirectional channels for
data communication among processors. For a d-dimensional cube topology, \( N = 2^d \)
processors make up the system. A scheduler called ROSES has been developed to
map precedence-constrained task graphs onto the machine.
In [60,61], the authors describe the development of a loosely-coupled multiprocessor system dedicated to dynamic control of multilink systems such as manipulators, legged vehicles, etc. At the heart of the system resides a PDP-11/23 as the central processing unit. Connected to the central processing unit are several satellite processors (PDP-11/03). Each satellite processor is connected to its adjacent processor via parallel I/O channels. The architecture is loosely coupled, that is, system modularity and flexibility is preserved. Various needed peripherals are served by the central processing unit. The architecture configuration is based upon static scheduling of the state space formulation of the Newton-Euler equations of motion [29]. Since the resolution of total forces and moments are not recursive (decoupled) in this formulation, the satellite processors compute the forward equations in parallel. The backward recursive equations are solved by the central processor. Besides, the central processor is used to coordinate the activities of satellite processors. Most I/O communications between the central processor and the satellite processors or between two satellite processors is through the system executive. This itself could be a bottleneck in the system. Also, due to the use of minicomputers, the speed of the processors are limited. In general, the cost-effectiveness vs. speed performance delivered by such systems is not optimal.

A more feasible architecture is proposed in [62]. At the heart of the system, there are two bit-serial array processors: one evaluates the forward recursions and the other computes the backward recursions of the Newton-Euler algorithm. Input parameters are loaded in a parallel-in-serial-out (PISO) first-in-first-out (FIFO) register bank. The computation is triggered as soon as needed operands are available in the registers. Output of the first processor, total forces and moments, are loaded into a serial-in-serial-out (SISO) last-in-first-out (LIFO) register bank. The second processor starts as soon as the \(N^{th}\) iteration of the first processor is
over. The output of the second processor, joint torques and forces, are stored in an output FIFO and may be accessed by the host computer. All the serial communication is asynchronous. A similar two-processor architecture is also proposed in [63].

2.3.3 Fine-Grain Parallelism

Various special-purpose full-custom VLSI processing elements have been designed to exploit low-level parallelism for robotics applications. One such architecture was designed by Leung and Shanblatt for the Direct Kinematics computation [64]. The MOS processor employs a two-phase nonoverlapped clocking scheme for synchronization. Two fixed-point multipliers and one adder are pipelined to compute the necessary primitive operations needed for Direct Kinematics. Input and output register files are connected to two internal busses and external communication is facilitated by two I/O ports. Seshadri [65] also mapped the same problem into a WE DSP16 Digital Signal Processor (DSP) by AT&T [66]. The Harvard architecture of DSP’s is also well suited for exploring low-level parallelism within robotics computation.

To facilitate implementation of robotics computations in general, the design of a VLSI-based Processing Element (PE) using 3 \( \mu \)m n-channel Metal Oxide Semiconductor (nMOS) technology has been completed [14]. The processor is composed of five major elements: a 32-bit floating-point multiplier (FPM), a 32-bit floating-point adder/subtractor (FPA/S), a 32-bit triple-port register file (RF), four 16-bit unidirectional I/O ports, and four 16-bit to 32-bit format converters (FC). Both the floating-point multiplier and adder have three stages of internal pipelining which provide increased throughput. The triple-port memory is used to hold the intermediate results and six different locations may be accessed in each clock cycle to

23
load/store all FPM and FPA/S operands/results during an arithmetic operation. Four 16-bit unidirectional I/O ports provide local and/or global broadcasting of data to their adjacent processors and/or to a high-level controller. The I/O ports are 16-bit, as opposed to 32-bit, in order to satisfy the I/O pin constraint on an integrated circuit chip. The north and west ports (PN and PW) are used for input while the south and east ports (PS and PE) are used for output. Four 32-bit words may be simultaneously transferred through all ports in two clock cycles through use of a single I/O operation.

The Robotics Processor is a three-bus horizontal microprogrammed machine which uses a 16 MHz external clock frequency and a 1 MHz internal clock frequency (clock period = 1 $\mu$s). In addition to the major data path elements, a microprogram sequencer, a control RAM, a microprogram register, and a boot-strap unit are incorporated into the chip for control. The microcode instruction width is 40–46 bits depending on the size of the register file (64–128 words) [14]. Three major instructions are available to perform either arithmetic, branch, or I/O operations. No means for handshaking is furnished through the port interfaces so that communication among processors must be synchronized through careful microcoding. That is, delays in program execution are necessary and result in processor idle periods during which a NO-OP instruction is executed. Further details of the Robotics Processor are given in [14]. The architecture is general enough to handle most robotics algorithms and is used in later chapters to implement the inertia matrix. A block diagram of the architecture is shown in Fig. 1.

Wang and Butner [54] have proposed a special-purpose system configuration to exploit job-level parallelism as explained before. A major block within the system is a Robotics Processor which exploits low-level parallelism. The Robotics Processor is equipped with two busses for global communication. One is designated
Figure 1: Block Diagram of the Robotics Processor.
as the bus master while the other is assigned as the bus slave to simplify bus communication. To ensure fast data transfer, the Robotics Processor is optimized for register-register operations. So, it inherents some of the RISC (Reduced Instruction Set Computer) concepts and implementations. The processor is composed of 3 floating-point multipliers operating in parallel and 3 Arithmetic Logic Units (ALU's) for other arithmetic and logical operations on three 32-bit wide busses. A multiplexing scheme is employed to route partial results among the three units. Intermediate results may be stored in a 32-bit wide vector register file. A program memory may be written into from an external bus and a data memory is connected to two external busses, one for inputing and the other for outputing the data. This processor runs at a 10MHz rate to compute a vector cross product in about 1 $\mu$sec. Unlike previous proposed architectures, much attention is given to the computation of sine and cosine functions. The CORDIC (COordinate Rotation Digital Computer) algorithm is used to implement trigonometric functions in VLSI. They estimate that a 32-bit trigonometric function may be computed in about $3 - 4 \mu$sec.

Other architectures have been proposed to take advantage of the inherent parallelism that exists within robotic computations [15]. There, the idea of a layered restructurable VLSI architecture is introduced. A four-layered architectural design is presented. At the lowest level of the architecture, a two-port register file, a floating-point multiplier (FPM), and a floating-point adder (FPA) are cascaded together through a two-bus interconnection scheme; all three blocks together are referred to as the floating-point processor (FPP). Such an architecture puts equal emphasis on both the FPM and FPA because, as stated in [15,67], the number of multiplies and additions are approximately equal for Inverse Plant Plus Jacobian control. Multiple FPP's may be connected in parallel through a dual
shift/broadcast network and a dual I/O channel to make up the second layer. So, this layer which is referred to as the Robotics Vector Processor (RVP) exploits parallelism at the matrix/vector operation level. Three FPP's along with one common controller and two 16-bit bidirectional I/O channels are used within an RVP for robotics applications since the primitive operations operate on $3 \times 3$ matrices and $3 \times 1$ vectors. The third layer called the Variable Very-Long-Instruction-Word Processor ($V^2LIWP$) is composed of several synchronized RVP's (tightly-coupled) and perhaps a scalar processor to exploit intra-block parallelism. RVP's activities are coordinated through a vector sequencer. The number of processors used depends on the algorithm being implemented. Finally, the last layer employs several $V^2LIWP$'s with a common memory as the coupling mechanism. This decentralized multiprocessor configuration exploits inter-block parallelism within, for example, multi-rate control algorithms.

2.4 Summary

This chapter has outlined some of the previous work completed in the areas of parallel algorithms and architectures for the real-time control of robotic systems. Specifically, a brief summary of parallel algorithms devised previously for the computation of the Inverse Dynamics problem and the manipulator inertia matrix have been presented. Special-purpose parallel architectures exploiting task-level parallelism, process-level parallelism, and fine-grain parallelism are briefly summarized. Previous related work on task partitioning and scheduling as a mechanism to bridge the gap between the algorithms and architectures is also presented.

The remainder of this dissertation is devoted to the development of parallel algorithms and architectures to compute the inertia matrix in real time. The next chapter will present the development of an $O(N)$ parallel algorithm plus several
systolic architectures. The development of an $O(\log_2 N)$ parallel algorithm will be given next. Then, a new computational model will be presented to further increase parallelism within the computation of the inertia matrix. The last chapter will summarize and conclude the work conducted in this dissertation, and possible related future work will be suggested.
CHAPTER III

DEVELOPMENT OF $O(N)$ PARALLEL ALGORITHM MAPPED ONTO SYSTOLIC ARCHITECTURES

3.1 Introduction

Determination of the inertia matrix involves a considerable amount of computation (approximately equal to that of Inverse Dynamics for a 6 degree-of-freedom manipulator) which varies as $O(N^2)$ on a single processor, where $N$ is the number of degrees of freedom of the manipulator [8]. To achieve a higher throughput and speedup, the order of computation needs to be reduced. Toward this end, this chapter presents the development of an $O(N)$ parallel algorithm for computing the manipulator inertia matrix.

The algorithm is based upon the determination of the mass, center of mass, and inertia of a series of composite rigid bodies for the manipulator [8]. These composite rigid bodies are made up of sets of links at the end of the manipulator which are assumed to be fixed with respect to each other (no relative joint movement). The equations for the algorithm are developed and then summarized. They are presented in a form which shows much of the parallelism inherent in the computation of the inertia matrix. The computation required is reduced from $O(N^2)$ for a single processor to $O(N)$ for $N$ processors.

Separate processors may be assigned to compute individual rows, columns, etc. Such mappings are used to partition the processes among several processors.
connected in a systolic configuration. The total computational delay is expected to be reduced significantly. At the heart of the systolic architectures is a VLSI-based Robotics Processor which is used as the basic processing element [14]. It contains a pipelined 32-bit floating-point multiplier and a 32-bit floating-point adder/subtractor functioning in parallel to achieve maximum throughput. A detailed description of this processing element which is used primarily to exploit fine-grain parallelism was given in Chapter II.

Previously, a similar approach has been taken to develop parallel architectures for other robotics computations, especially Inverse Dynamics [15,32,33,34]. Some of the issues considered include development of parallel algorithms, choice of processor, interconnection structure between processors, effective mapping of the algorithm onto the architecture, and evaluation of alternative configurations.

Parallel computation of the inertia matrix has also been considered in the context of computing robot forward dynamics for real-time computer simulation [9]. Parallel linear recurrence algorithms based on the “recursive doubling” technique [18] were used to compute the diagonal elements of the inertia matrix and a modified row-sweeping algorithm was applied to compute the off-diagonal elements [9]. The latter is similar to the systolic architecture developed in column form, given later in this chapter. In each case, a processor is assigned to compute the elements of a single column of the inertia matrix. This chapter will extend the work to evaluate a number of other forms of systolic architectures for the inertia matrix.

In the next section, the development of an $O(N)$ parallel algorithm is given. Then the development of various systolic architectures, using 1, $N$, and $N(N+1)/2$ processors, for computing the inertia matrix, will be presented. The various configurations will be evaluated in terms of the compute time delay, which is of $O(N)$
for both the \( N \) and \( N(N + 1)/2 \) processor cases. In addition to this basic delay parameter, other important criteria will be used to evaluate the various configurations and these include the initiation rate (sampling rate), CPU utilization, speedup, and on-chip memory required. In all cases, the I/O communication time and idle time, required to synchronize the processors, will be fully considered in the evaluation.

3.2 \( O(N) \) Parallel Algorithm for the Inertia Matrix

In this section, the development of a linear recursive algorithm for computing the manipulator inertia matrix is fully explained. The algorithm is derived from [8] but expressed in a form which explicitly shows the reduction of the computation from \( O(N^2) \) to \( O(N) \) when \( N \) processors are used. Details of the development are given to provide additional insight into the algorithm so that subsequent selection of appropriate systolic architectures may be facilitated.

The development of the algorithm may be formalized by considering the general dynamic equations of motion for a single open-chain manipulator, in joint space with physical constraint forces and moments between the links eliminated. The general dynamic equations of motion with the foregoing conditions taken into account are given in Eq. (1.1), and are repeated here for ease of reference:

\[
\mathbf{T} = \mathbf{H}(\mathbf{q})\mathbf{\ddot{q}} + \mathbf{C}(\mathbf{q}, \mathbf{\dot{q}})\mathbf{\dot{q}} + \mathbf{G}(\mathbf{q}) + \mathbf{J}^T(\mathbf{q})\mathbf{f} \tag{3.1}
\]

where

- \( \mathbf{T} \) \( N \times 1 \) vector of joint torques (forces),
- \( \mathbf{q}, \mathbf{\dot{q}}, \mathbf{\ddot{q}} \) \( N \times 1 \) vectors of joint positions, rates, and accelerations,
- \( \mathbf{C}(\mathbf{q}, \mathbf{\dot{q}}) \) \( N \times N \) matrix of coriolis and centrifugal force terms,
- \( \mathbf{G}(\mathbf{q}) \) \( N \times 1 \) vector of gravitational forces,
- \( \mathbf{J}(\mathbf{q}) \) \( 6 \times N \) Jacobian matrix,
- \( \mathbf{f} \) \( 6 \times 1 \) vector of external forces and moments exerted by link \( N \),
and $H(q)$ is the $N \times N$ symmetric, positive-definite, inertia matrix that is of interest to compute.

For a given set of joint position, velocity, and acceleration vectors for a specified trajectory and for given external forces and moments, the required torques to be applied by the actuators may be computed from Eq. (3.1). This is the problem of Inverse Dynamics which computes the actuating signals for a given velocity and acceleration profile. The most efficient algorithm for Inverse Dynamics is generally considered to be based on the Newton-Euler formulation [28,30] which does not require computation of the individual terms of Eq. (3.1). Briefly, the technique employs a two-step process, a forward recursion and a backward recursion, to give a computation which is linearly related to $N$, the number of degrees of freedom. The forward recursion ($i = 1$ to $N$) begins at the base of the manipulator, and the velocity and acceleration of a link are computed based on the values of these variables for the previous link and the relative values at the connecting joint. The resultant forces and moments exerted on each link are first computed in the backward recursion ($i = N$ to 1) by applying Newton's and Euler's equations. Then, beginning with the external forces and moments applied to the end-effector, the forces and moments applied to a link (at the near end) are computed based on the resultant values for the link and those applied by it to the previous link considered.

A straightforward approach in solving for $H(q)$ is to apply Inverse Dynamics to Eq. (3.1) $N + 1$ times. To do so, Eq. (3.1) may be rewritten in the following form:

$$\tau = H(q)\dot{q} + b$$

(3.2)

where

$$b = C(q, \dot{q})\dot{q} + G(q) + J^T(q)f.$$  

(3.3)
The vector \( \mathbf{b} \) may be solved for by simply setting \( \ddot{\mathbf{q}} \) equal to 0 when applying Inverse Dynamics. That is,

\[
\mathbf{b} = \mathbf{T} \bigg|_{\ddot{\mathbf{q}}=[00...0]^T} \cdot
\]

(3.4)

Now, the columns of \( \mathbf{H}(\mathbf{q}) \) may be solved for in Eq. (3.2) by applying a unit vector of acceleration to the joints. That is,

\[
\mathbf{H}_1 = (\mathbf{T} - \mathbf{b}) \bigg|_{\dot{\mathbf{q}}=[100...]^T} \cdot
\]

(3.5)

where \( \mathbf{H}_1 \) is the first column of the \( \mathbf{H}(\mathbf{q}) \) matrix and in general,

\[
\mathbf{H}_i = (\mathbf{T} - \mathbf{b}) \bigg|_{\dot{\mathbf{q}}=[00...1...0]^T} \cdot
\]

(3.6)

By repeating the above process \( N \) times (not necessarily recursively), all of the elements of \( \mathbf{H}(\mathbf{q}) \) may be extracted.

Further simplifications may be made to decrease the amount of computation needed. One obvious simplification is to set \( \dot{\mathbf{q}} \), \( \mathbf{G}(\mathbf{q}) \), and \( \mathbf{f} \) to zero when applying Inverse Dynamics. This itself eliminates the need to compute \( \mathbf{b} \) and all other subsequent subtractions of \( \mathbf{b} \) from \( \mathbf{T} \). Also, it should be noted that the inertia matrix is a symmetric matrix. That is, the diagonal and the upper off-diagonal elements of \( \mathbf{H}(\mathbf{q}) \) are all that are needed to fully specify it. This indicates that all of the elements of the \( \mathbf{T} \) vector need not be obtained in each step and this further simplifies the computation.

One additional concept which is of computational significance is to note that the application of a unit acceleration to a joint (for instance \( \ddot{q}_i = 1 \) at joint \( i \)) with all joint velocities and other joint accelerations equal to zero, divides the manipulator chain into two sets of composite rigid bodies with one degree of freedom between them as shown in Fig. 2. Links \( i \) through \( N \) constitute one of these com-
Figure 2: Two Composite Rigid Body Modeling of an Open-Chain Mechanism.

Positc rigid bodies and may be modeled as a single rigid body with a composite mass ($M_i$), composite center of mass ($c_i$), and composite moment of inertia ($E_i$) as shown in Fig. 3. For this single composite rigid body, the forces and moments at joint $i$ due to a unit acceleration there may be rather simply computed by applying the Newton-Euler equations to the composite body. While the torques within the composite body are not computed, this is not a problem since they are related to the off-diagonal elements of the lower-half of the inertia matrix.

Since the computation of $M_i$, $c_i$, and $E_i$ is an important part of the determination of the inertia matrix, a linear recursive technique has been derived for each and is given as follows. From basic physics principles,
Figure 3: Composite Center of Mass ($c_i$) and Composite Moment of Inertia ($E_i$) for links $i$ through $N$.

\[ M_i = M_{i+1} + m_i \]  
\[ \text{where } m_i \text{ is the mass of link } i. \]  
\[ \text{Also, } c_i, \text{ the composite center of mass with respect to the origin of link } i \text{ coordinates is given as (Fig. 3)} \]
\[ c_i = \frac{1}{M_i} \left\{ m_i (s_i^*) + M_{i+1} \left( c_{i+1} + p_{i+1}^* \right) \right\} \]  
\[ \text{where } p_{i+1}^* \text{ denotes the position of the origin of link } i+1 \text{ coordinates with respect to the origin of link } i \text{ coordinates, and } s_i^* \text{ denotes the position of the center of mass of link } i \text{ with respect to the origin of link } i \text{ coordinates. Finally, } E_i \text{ may} \]
be determined by using the three-dimensional version of the parallel axis theorem which states

$$I_O = I_G + M \left[ \left( r^T r \right) 1 - (rr^T) \right]$$

(3.9)

where $1$ is the $3 \times 3$ identity matrix, $I_O$ is the inertia matrix of a rigid body about an arbitrary origin $O$, $I_G$ is the inertia matrix relative to the center of mass, and $r$ is the position vector between the two points. Thus, noting Fig. 3:

$$E_i = E_{i+1} + M_{i+1} \left[ \left( r_{i+1}^T r_i \right) 1 - (r_ir_i^T) \right] + I_{i+1} + m_i \left[ \left( r_i^* r_i^* \right) 1 - (r_i^* r_i^*) \right]$$  (3.10)

where $I_i$ is the inertia matrix of link $i$ at its center of mass.

Links $i - 1$ through 1 constitute the second composite rigid body. For these links ($j \leq i - 1$), particular components of the corresponding forces and moments at joint $j$ due to a unit acceleration at joint $i$ ($f_{j,i}$ and $n_{j,i}$ respectively), that are needed to keep the links fixed with respect to each other are just the elements of the inertia matrix. The Newton-Euler approach may be used to determine the necessary force and moment components (along or about the joint axes). The appropriate equations are developed below for revolute joints while the final equations for both revolute and prismatic joints are given in Table 1.

For the upper composite rigid body represented by the three quantities $M_i$, $e_i$, and $E_i$, the total resultant force ($F_i$) and moment ($N_i$) exerted on link $i$ may be computed by applying the Newton-Euler equations:

$$F_i = M_i \ddot{e}_i$$

(3.11)

and

$$N_i = E_i \dot{\omega}_i + \omega_i \times (E_i \omega_i).$$

(3.12)
The vector $\omega_i$ is the angular velocity of the $i^{th}$ link (and composite body) and is zero since all velocities are zero in the analysis. Also, for a unit acceleration at joint $i$, 

$$\dot{z}_i = \bar{q}_i z_i = z_i.$$  \hspace{1cm} (3.13)

Thus, 

$$\dot{c}_i = \dot{z}_i \times c_i = z_i \times c_i.$$  \hspace{1cm} (3.14)

Substituting Eqs. (3.13) and (3.14) into Eqs. (3.11) and (3.12) gives:

$$F_i = M_i (z_i \times c_i)$$  \hspace{1cm} (3.15)

and

$$N_i = E_i z_i.$$  \hspace{1cm} (3.16)

Consequently, $f_{i,i}$ and $n_{i,i}$ may be determined by simply resolving these forces and moments to the origin of link $i$ coordinates with the following results:

$$f_{i,i} = F_i$$  \hspace{1cm} (3.17)

and

$$n_{i,i} = N_i + c_i \times F_i.$$  \hspace{1cm} (3.18)

Having obtained $f_{i,i}$ and $n_{i,i}$, Newton-Euler backward recursion may be applied to links $i - 1$ through 1 so as to compute the necessary inter-link constraint forces and moments to keep the joints stationary. Since the composite rigid body nearest the base is completely stationary, the resulting equations just involve resolution of the same force and moment to the various origins of the associated link coordinate systems. This gives:
\[ f_{j,i} = f_{j+1,i} \quad (3.19) \]

and

\[ n_{j,i} = n_{j+1,i} + p_{j+1}^* \times f_{j+1,i} \quad (3.20) \]

The joint torque is just the \( z \) component of the moment vector for rotational joints. That is,

\[ \tau_{j,i} = n_{j,i} \times \mathbf{z} \quad j \leq i. \quad (3.21) \]

Finally, the elements of the inertia matrix may be determined by using the following equality:

\[ H_{j,i} = \tau_{j,i} \quad j \leq i. \quad (3.22) \]

The equations in the foregoing are completely summarized in Table 1 and are generalized to cover the case of prismatic joints. Note that \( ^iU_{i+1} \) is the \( 3 \times 3 \) transformation between link \( i \) coordinates and link \( i+1 \) coordinates and is used to keep most quantities in local coordinates so as to reduce the overall computation in a way similar to the most efficient Newton-Euler method for Inverse Dynamics [28]. Quantities are expressed in the local coordinate system indicated by their leading superscript (in the case of \( p^* \)) or by their first trailing subscript (all other cases).

It should be emphasized that the algorithm listed in Table 1 is a task which is composed of two processes. The two processes are signified by BEGIN1 and BEGIN2 blocks. The BEGIN1 process generates the composite rigid body parameters and also the diagonal elements of the inertia matrix; whereas, the BEGIN2 process generates the off-diagonal elements of the inertia matrix. Once the BEGIN1 process is completed, the BEGIN2 process may begin and involves backward recursions which may all be performed in parallel. This fact is indicated in
Table 1: $O(N)$ Parallel Algorithm for Computing the Inertia Matrix.

\begin{verbatim}
CONST
\begin{align*}
M_{N+1} &= 0 \\
c_{N+1} &= 0 \\
E_{N+1} &= 0 \\
\varepsilon_0 &= [0 0 1]^T \\
\sigma_i &= \begin{cases} 
1 & \text{revolute joint} \\
0 & \text{prismatic joint}
\end{cases}
\end{align*}
BEGIN
\{\* Computation of composite rigid body parameters and diagonal elements \*\}
\{\* of the inertia matrix \*\}
FOR $i := N$ TO $1$ DO
BEGIN1
\begin{align*}
M_i &:= M_{i+1} + m_i \\
c_i &:= \left\{ \frac{1}{M_i} [m_i(s_i^T) + M_{i+1} (t_i U_{i+1} c_{i+1} + t_i p_{i+1}^e)] \right\} \\
E_i &:= t_i U_{i+1} E_{i+1} + M_{i+1} \left[ (t_i U_{i+1} c_{i+1} + t_i p_{i+1}^e) (t_i U_{i+1} c_{i+1} + t_i p_{i+1}^e)^T \\
&\quad - (t_i U_{i+1} c_{i+1} + t_i p_{i+1}^e) (t_i U_{i+1} c_{i+1} + t_i p_{i+1}^e)^T \right] \\
&\quad + \varepsilon_i + m_i \left( (s_i^T - c_i) (s_i^T - c_i) - (s_i^T - c_i) (s_i^T - c_i)^T \right) \\
F_i &:= \sigma_i (\varepsilon_0 \times M_i c_i) + \varepsilon_i (M_i \varepsilon_0) \\
N_i &:= \sigma_i (E_i \varepsilon_0) \\
f_{i,i} &:= F_i \\
n_{i,i} &:= N_i + c_i \times F_i \\
H_{i,i} &:= \sigma_i (n_{i,i} \cdot \varepsilon_0) + \varepsilon_i (f_{i,i} \cdot \varepsilon_0)
\end{align*}
END1
\{\* Computation of off-diagonal elements of the inertia matrix \*\}
FOR ALL $i := N$ TO $1$ DO
BEGIN2
\begin{align*}
f_{j,i} &:= t_j U_{j+1} f_{j+1,i} \\
n_{j,i} &:= t_j U_{j+1} (n_{j+1,i} + t_j p_{j+1}^e \times f_{j+1,i}) \\
H_{j,i} &:= \sigma_j (n_{j,i} \cdot \varepsilon_0) + \varepsilon_j (f_{j,i} \cdot \varepsilon_0)
\end{align*}
END2
\end{verbatim}
the algorithm listing by a FOR ALL statement. Thus, it can be seen that the total computation is simply of $O(N)$. The next section presents several systolic architectures to implement the above $O(N)$ algorithm using a VLSI-based Robotics Processor [14].

3.3 Implementation of the Inertia Matrix with the Robotics Processor

This section presents several systolic configurations to implement the inertia matrix using the Robotics Processor described in Chapter II (Fig. 1). The use of a systolic array of Robotics Processors, as opposed to a more traditional attached processor, not only eliminates a potential I/O bottleneck to a great extent, but also makes a VLSI-based implementation feasible. It should be noted that simplicity and regularity (modularity) are two key issues behind the VLSI design. Also, the architectures must facilitate locality of communication so that the computation is compute-bound instead of I/O-bound.

Toward this end, the $O(N)$ parallel algorithm given in Table 1 will be used to develop the systolic configurations. In each case, the I/O (communications overhead) and idle times are fully accounted for. In addition, in writing the microcode, all features of the processor including its inherent pipelining and parallelism are fully explored. Various hardware configurations have been considered with the objectives to minimize the idle time and communications overhead and the resulting execution time delay, and to maximize throughput and CPU utilization. However, unlike many of the previously proposed architectures for robotics computations, attention has also been focused on feasibility and cost effectiveness based on current or near-term technology.

From Table 1, it may be observed that most of the operations are either $3 \times 1$ vector operations (vector addition ($V+V$), dot product ($V\cdot V$), cross prod-
uct (V×V), and scalar-vector product (SV)) or 3 × 3 matrix operations (matrix addition (M+M); matrix-matrix (MM), matrix-vector (MV), and scalar-matrix (SM) multiplication). Reservation tables have been developed to schedule these operations on the Robotics Processor. Then, a task graph has been developed for the O(N) algorithm using matrix-vector operations as its primitives. It incorporates information as to the precedence of operations, degree of parallelism, and relative complexity of the different operations (Fig. 4). The complexity is explicitly indicated by the height of a computation node. All primitives are normalized with respect to the V+V operation. For example, with one unit of computation for V+V, MV would be 4 units of computation, MM would be 6 units of computation, etc.

Based on the task graph, various systolic configurations are proposed and will be evaluated. Among them are N and N(N + 1)/2-processor architectures. With the aid of the task graph and with full knowledge of the architecture, efficient code has been written to take complete advantage of parallelism and pipelining. An evaluation of the microcode steps needed to compute the inertia matrix has been used to calculate a number of parameters including the compute time, I/O time, initiation rate, CPU utilization, speedup, and total memory required.

It should be pointed out that an I/O operation on the Robotics Processor takes 2 clock cycles (since the I/O ports are only 16 bits wide); whereas, any other operation takes only one clock cycle (1 µs). Again, it should be noted that NO-OP operations are used as required to synchronize input/output operations among processors. Another important point to note is that since the Robotics Processor in its present configuration is incapable of performing sine, cosine, and inverse operations, sin θ_i, cos θ_i, and 1/π_i must be furnished by the host processor (involves relatively small amounts of computation). Also, s_i, p_i, m_i, and I_i, for a
Figure 4: Task Graph for the $O(N)$ Algorithm.
Figure 4 (continued)
particular manipulator, should be initially downloaded to the Robotics Processors before real-time execution begins.

In addition, in writing the microcode for all the proposed architectures, the following important observations have been made in order to make the code more efficient.

- The inertia matrix is a symmetric matrix. Thus, the resultant matrix produced by the transformation $jU_{j+1} E_{j+1} jU_{j+1}^T$ is also symmetric.
- For a given vector $v$, $(v \cdot v)I$ is a diagonal matrix.
- For a given vector $v$, $(vv^T)$ is a symmetric matrix.

A description of the various architectures which have been explored to execute the microcode required to compute the inertia matrix follows.

One-processor, several $N$-processor (computation of the inertia matrix in column, row, and diagonal form) and $N(N+1)/2$-processor configurations have been considered and will be compared and contrasted. For an $N$-processor configuration, for instance, in column form, it is meant that the computation is partitioned in such a way that a particular processing element generates the corresponding column elements of the inertia matrix. That is, processor $N$ generates all the elements in the $N^{th}$ column, processor $N-1$ generates all the elements in the $N-1^{st}$ column, and in general, processor $i$ generates all the elements in the $i^{th}$ column of the inertia matrix as shown in Fig. 5. However, since the inertia matrix is symmetric, only the diagonal and upper off-diagonal elements are computed. Its associated timing diagram is shown in Fig. 6. The computation of various sub-tasks is indicated, to scale, by letters on the timing diagram. In particular, note that time slot "a" corresponds to the time period during which the host computer
Figure 5: N-Processor Systolic Architecture in Column Form.
Figure 6: Timing Diagram for $N$ Processors in Column Form.
outputs the required data to the corresponding PE's, “b” corresponds to computation of \( c_i \), etc. Also, the timing diagram clearly shows the critical path of the computation (determination of \( E_i \)) from which values for compute time, I/O time, and execution time are derived.

The same approach is used to configure \( N \) processors in row or diagonal form as shown in Figs. 7 and 9, respectively. Their corresponding timing diagrams are shown in Figs. 8 and 10. Finally, the interconnections for the \( N(N + 1)/2 \)-processor architecture is shown in Fig. 11. The figure indicates that the critical computational path is dictated by the processing elements located on the diagonal. It may be noticed that the diagonal processing elements by themselves have a very close resemblance to the column configuration. The timing diagram for this critical path is very similar to the column form and it is not explicitly shown here.

The performance characteristics of the \( N \)-processor configurations are summarized in Table 2 along with 1-processor and \( N(N + 1)/2 \)-processor configurations. A number of parameters are used in Table 2 to evaluate the various configurations of Robotics Processors. Compute time is the processing delay time needed for the entire array to compute the inertia matrix. I/O time is the delay time spent doing local and global communications during a single computation of the inertia matrix. Execution time is the summation of compute time and I/O time and is the total delay in producing the inertia matrix. Initiation rate is the rate at which successive computations of the inertia matrix may be initiated. For the 1-processor configuration, it is just the inverse of the execution time. The initiation rates for all other systolic configurations are determined so as to ensure all PE's, in a given configuration, are tightly synchronized under lock-step control. This simply implies the existence of some idle time which does not effect the execution time.
Figure 7: N-Processor Systolic Architecture in Row Form.
Figure 8: Timing Diagram for $N$ Processors in Row Form.
Figure 9: N-Processor Systolic Architecture in Diagonal Form.
Figure 10: Timing Diagram for $N$ Processors in Diagonal Form.
Figure 11: $N(N + 1)/2$-Processor Systolic Architecture.
Table 2: Architecture Parameters as a Function of the Number of Processors Used. (Numbers correspond to $N = 8$)

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>1</th>
<th>$N$</th>
<th>$N$</th>
<th>$N$</th>
<th>$N(N + 1)/2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Time $T_c (\mu s)$</td>
<td>1728</td>
<td>560</td>
<td>986</td>
<td>776</td>
<td>500</td>
</tr>
<tr>
<td>I/O Time $T_{io} (\mu s)$</td>
<td>84</td>
<td>120</td>
<td>302</td>
<td>128</td>
<td>92</td>
</tr>
<tr>
<td>Execution Time $T_e (ps)$</td>
<td>1812</td>
<td>680</td>
<td>1288</td>
<td>904</td>
<td>652</td>
</tr>
<tr>
<td>Initiation Rate $IR (us^{-1})$</td>
<td>1/1812</td>
<td>1/444</td>
<td>1/1006</td>
<td>1/904</td>
<td>1/408</td>
</tr>
<tr>
<td>Average CPU Utilization %</td>
<td>100.0</td>
<td>65.0</td>
<td>35.6</td>
<td>37.0</td>
<td>21.4</td>
</tr>
<tr>
<td>Speedup ($\eta$)</td>
<td>1.0</td>
<td>2.66</td>
<td>1.41</td>
<td>2.0</td>
<td>2.76</td>
</tr>
<tr>
<td>Number of 32-bit Registers</td>
<td>245</td>
<td>119</td>
<td>63</td>
<td>119</td>
<td>63</td>
</tr>
<tr>
<td>Control Memory Size (K bits)</td>
<td>11.55</td>
<td>18.8</td>
<td>39.0</td>
<td>38.6</td>
<td>15.6</td>
</tr>
<tr>
<td>Total Memory (K bits)</td>
<td>19.3</td>
<td>22.0</td>
<td>41.0</td>
<td>42.45</td>
<td>17.6</td>
</tr>
</tbody>
</table>
The speedup, \( \eta \), is defined as the ratio of the execution time for the 1-processor case to the execution time for a particular case considered. RN is the number of 32-bit registers required in the register file to store all of the known parameters and computed variables needed in the computation. The Control Memory Size (CMS) is defined as follows:

\[
CMS = MC \left( 4 + 6 \lceil \log_2 RN \rceil \right)
\]  

(3.23)

where MC is the number of microinstruction steps required for the entire operation. The value of 6 is associated with the number of register file address fields and 4 is the number of additional control/opcode bits within a microinstruction. Finally, the total memory is just the sum of RN (times 32) and CMS.

The results tabulated in Table 2 convey several important facts. First of all as expected, as the number of processors increases the compute time and execution time decrease. These do not decrease substantially with more than \( N \) processors though, because very little additional overlap can be obtained along the critical path (computing the composite rigid body parameters). Correspondingly, the initiation rate increases since a specific computation is more widely shared among the processors. The speedup is about 2.7 for the \( N \)-column and \( N(N + 1)/2 \) cases. Among the \( N \)-processor configurations, the column form gives the best results with \( \eta = 2.66 \), \( T_e = 680 \mu s \), and approximately 22.6K bits of memory for \( N = 8 \). Also, on the average, the processors are more fully utilized for this configuration. Another very important figure in Table 2 is the amount of memory, which is a major factor in determining the required silicon area for the chip. The \( N(N + 1)/2 \)-processor configuration has by far the best results in this case. This configuration not only requires the smallest memory, but also has the smallest
execution time. The major drawback for this case is its cost which is related to the large number of processors required ($O(N^2)$).

Overall, the implementation of the $O(N)$ algorithm using $N$ processors in column form appears to give the best results for computing the manipulator inertia matrix. The $N(N + 1)/2$-processor case also had a better performance in certain aspects, but the cost of the large number of processors ($O(N^2)$) may be prohibitive.

### 3.4 Summary and Conclusions

This chapter has presented the development of an $O(N)$ parallel algorithm plus several systolic architecture configurations for computing the $N \times N$ inertia matrix for a robot manipulator. The algorithm used is generally based on employment of the computationally efficient Newton-Euler method for Inverse Dynamics (a total of $N$ times). However, considerable simplification is possible since in the use of Inverse Dynamics, the joint velocities and joint accelerations (all but one) are equal to zero. This results in sets of composite rigid bodies for which their composite mass, center of mass, and inertia are computed. The equations are summarized in Table 1 which explicitly shows much of the parallelism which is inherent in the algorithm.

The VLSI Robotics Processor given in Chapter II has been used to implement the algorithm. It has a 32-bit floating-point multiplier and 32-bit floating-point adder/subtractor in parallel, with three levels of internal pipelining each. A triple-port register file has been configured around the floating-point units so as to facilitate execution of matrix-vector operations.

Several systolic architectures have been proposed to compute the inertia matrix. Among them were an $N$-processor configuration in column form and $N(N + 1)/2$-processor configuration, both of which offered acceptable performance.
at reasonable complexity. The total execution time delay (includes compute time and I/O time) was of $O(N)$ for each of these cases. While, in general, increasing the number of processors will decrease the total execution time delay, the results of this chapter indicate that increased communications overhead and idle time (due to increasing problems of synchronization) negate many of the advantages of going to more processors. Also, the average CPU utilization decreases and therefore decreases the overall cost-effectiveness. However, the initiation rate does increase significantly with increasing numbers of processors, and to a degree, this may have a positive impact on control system stability.

Since the computation of the diagonal and upper off-diagonal elements of the inertia matrix are recursive, the concept of recursive doubling [18] may be used to develop an $O(\log_2 N)$ parallel algorithm. A complete treatment of the parallel algorithm will be the main focus of the next chapter. In addition, a cube architecture is employed to implement the $O(\log_2 N)$ algorithm using a mapping which is similar to that used in [14] for the Jacobian.
CHAPTER IV

DEVELOPMENT OF $O(\log_2 N)$ PARALLEL ALGORITHM MAPPED ONTO A CUBE ARCHITECTURE

4.1 Introduction

As mentioned previously, the determination of the inertia matrix involves a considerable amount of computation. The most efficient serial algorithm for computing the inertia matrix was first developed in [8] and requires $O(N^2)$ time on a single processor system. An $O(N)$ parallel algorithm plus several systolic architectures employing $N$ and $N(N + 1)/2$ processors were presented in the previous chapter. There, the composite rigid body method developed in [8] was used to formulate the $O(N)$ parallel algorithm [68,69].

The composite rigid body method is also used in this chapter to develop an $O(\log_2 N)$ parallel algorithm to compute the manipulator inertia matrix in real time [70,71]. The computation is divided into $O(\log_2 N)$ levels where in the first level the composite rigid body parameters of sets of two links are computed, in the second level the composite rigid body parameters of sets of four links are computed, etc. This computational approach is referred to as recursive doubling [18], and is applied to reformulate the linear recurrence equations required to compute the diagonal elements of the inertia matrix.

Computation of the off-diagonal elements involves solution of $N$ sets of linear recurrence equations of size $N$, $N - 1$, etc. for which recursive doubling is not
easily applied. In particular, the calculation of position and orientation transforms across the links of a varying-size composite rigid body at the base end of the manipulator is required. A new method is developed to compute the off-diagonal elements in $O(\log_2 N)$ time, and it avoids redundant computation of the position and orientation transforms. Hence, the computation of both the diagonal and off-diagonal elements are of $O(\log_2 N)$, resulting in an $O(\log_2 N)$ algorithm.

A combination of the $O(\log_2 N)$ algorithm for the diagonal elements and $O(N)$ algorithm for the off-diagonal elements is also considered. The resulting $O(\log_2 N) + O(N)$ algorithm is computationally the most efficient one for a cube architecture implementation, and it is discussed in a later section. As in the previous chapter, the Robotics Processor is used in the architecture. And for performance evaluations, the I/O and idle times are fully considered as well as the compute time, initiation rate, CPU utilization, speedup, and total memory required. Also, in writing the microcode for the Robotics Processor, all features of the processor including its inherent pipelining and parallelism are fully utilized.

Set notation is used to develop the equations for the $O(\log_2 N)$ algorithm in a form which explicitly shows the parallelism available. The notation used was first developed in part in [14] where recursive doubling was used to compute the Jacobian. Finally, the chapter evaluates the number of processors required for various parallel formulations of the manipulator inertia matrix.

In the next section, the $O(\log_2 N)$ parallel algorithm is developed for the inertia matrix. The entire parallel algorithm is then summarized in a table in a form which shows much of the parallelism inherent in the algorithm. The relationship between the number of processors required and the order of the computation is also given. Then a cube architecture is employed as a way to implement the $O(\log_2 N)$ algorithm for computing the diagonal elements, using a mapping which is simi-
lar to that used in [14] for the Jacobian, and the $O(N)$ algorithm for computing the upper off-diagonal elements. The performance indices of the cube configuration are then compared with the column configuration which was presented in the previous chapter and resulted in the best overall performance over other systolic configurations. Finally, the work is summarized, and conclusions are made.

4.2 Development of an $O(\log_2 N)$ Parallel Algorithm for the Inertia Matrix

The concept of recursive doubling [18] may be used to develop an $O(\log_2 N)$ parallel algorithm to compute the composite rigid body parameters and diagonal elements of the inertia matrix. It has been previously applied to achieve $O(\log_2 N)$ parallel algorithms for Inverse Dynamics [33,34]. In order to understand the basic concept and develop the notation used, computation of the composite rigid body masses for a manipulator of eight degrees of freedom will first be considered in this section. A parallel algorithm of $O(\log_2 N)$ will be developed for computing these masses, $M_i$. Then, the approach will be extended to computing all of the composite rigid body parameters, resulting in an $O(\log_2 N)$ parallel algorithm for computing the diagonal elements of the inertia matrix.

Computation of the off-diagonal elements of the inertia matrix involves $N$ independent recursions of varying size for which recursive doubling is not easily applied. The last part of this section gives an $O(\log_2 N)$ algorithm to compute these elements. It effectively uses the position and orientation transforms for the full $N$-link manipulator, which are computed while obtaining the diagonal elements, to transform forces and moments over a reduced, varying-size composite rigid body at the base end.
4.2.1 Parallel Algorithm for $M_i$

As shown in Table 1, a recursive algorithm for computing $M_i$ is given by

$$M_{N+1} = 0$$  \hspace{1cm} (4.1)

$$M_i = M_{i+1} + m_i \quad \text{for } i = N, \cdots, 1.$$  \hspace{1cm} (4.2)

Eq. (4.2) is an example of a linear recurrence relationship for which recursive doubling [18] may be applied. Consider computation of $M_1$, the composite rigid body mass for the entire manipulator. The mass across sets of two links may first be computed, all in parallel. For eight links, the links are simply combined as follows:

$$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\} \rightarrow \{1,2\}, \{3,4\}, \{5,6\}, \{7,8\} \quad (4.3)$$

which indicates that the size of the set, for which the mass has been computed, has doubled. Again, doubling the number of links in a set gives

$$\{1,2\}, \{3,4\}, \{5,6\}, \{7,8\} \rightarrow \{1,2,3,4\}, \{5,6,7,8\} \quad (4.4)$$

indicating that the mass is now computed for sets of four links. The doubling effect may be recursively applied (recursive doubling):

$$\{1,2,3,4\}, \{5,6,7,8\} \rightarrow \{1,2,3,4,5,6,7,8\} \quad (4.5)$$

so that the total mass of all eight links, $M_1$, is available after 3 steps ($\log_2 8$).

Fig. 12 shows the flow of the data and computation involved in the three steps for computing $M_1$ ($M_{1,N}$). In the figure, $M_{j,k}$ represents the mass of links $j$ through $k$. This implies the following mapping relationship:

$$M_i \rightarrow M_{i,N} \quad (4.6)$$
and

\[ m_i \rightarrow M_{i,i} \]  \hspace{1cm} (4.7)

to extend the previous notation used.

Equations may be developed for computing \( M_1 \). The total number of levels of computation is given by

\[ l_T = \log_2 N. \]  \hspace{1cm} (4.8)

The total number of sets of links at each level whose composite mass is to be computed in parallel is given by

\[ u_T = 2^{l_T - l} \]  \hspace{1cm} (4.9)
where \( l \) is the level number. Also, the variable \( u \) is defined and used in Fig. 12 as the number of a set on a given level.

At the first level \((l = 1)\), links are combined in groups of two; at the second level \((l = 2)\), links are combined in groups of four, etc. Thus, the number of links in a set at each level, the width \( w \), is a function of \( l \) and is given by

\[
w = 2^l. \tag{4.10}
\]

Each computational step in Fig. 12 then combines two sets of links into one:

\[
\{i + 1, \cdots, j\}, \{j + 1, \cdots, k\} \longrightarrow \{i + 1, \cdots, j, j + 1, \cdots, k\} \tag{4.11}
\]

where

\[
i = w(u - 1), \tag{4.12}
\]

\[
j = w(u - 0.5), \text{ and} \tag{4.13}
\]

\[
k = w u. \tag{4.14}
\]

Using the above notation for indexing, the following set of equations formalizes the parallel computation of \( M_1 (M_{1,8}) \).

\[
\begin{align*}
\text{FOR} \ l := 1 \ TO \ l_T \ DO \\
\{ \begin{array}{l}
\text{w} := 2^l \\
\text{u_T} := 2^{l_T - l}
\end{array} \} \\
\text{FOR} \ ALL \ u := 1 \ TO \ u_T \ DO \\
\{ \begin{array}{l}
i := w(u - 1) \\
j := w(u - 0.5) \\
k := w u
\end{array} \} \\
\text{BEGIN} \\
M_{i+1,k} := M_{i+1,j} + M_{j+1,k} \\
\text{END}
\end{align*} \tag{4.15}
\]

The above equations only determine the following set of composite rigid body masses:

\[
\{M_{1,8}, M_{5,8}, M_{7,8}, M_{8,8}\} \equiv \{M_1, M_5, M_7, M_8\}. \tag{4.16}
\]
To obtain the other composite rigid body masses needed, in general, multiple computations must be performed on each set of links at each level. All necessary computational steps are shown in Fig. 13. Note that the masses for subsets of links, from a link in the first half of a set to the last link in the set (Eq. (4.11)), must be computed. In so doing, another parameter which gives the total number of computations for each set of links on level \( l \) may be defined:

\[
v_T = 2^{l-1}.
\]  

(4.17)

Here, \( u \) is also defined to be the computation number for set \( u \) for a given level \( l \). Note that

\[
u_T * v_T = 2^{T-l} * 2^{l-1} = 2^{T-1} = \frac{N}{2}
\]  

(4.18)
which is constant over all the levels.

4.2.2 Parallel Algorithm for the Diagonal Elements

The approach taken to compute the composite rigid body masses, $M_i$, may be extended to develop the $O(\log_2 N)$ parallel algorithm for all of the composite rigid body parameters and to further compute the diagonal elements of the inertia matrix. The algorithm is summarized in the first part of Table 3 and will be discussed in the following paragraphs.

First consider the computation of the composite rigid body parameters: mass ($M$), center of mass ($c$), and moment of inertia ($E$). At a given level $l$, the main body of the computation is to calculate these in parallel, for any combination of $u$ and $v$, across the sets of links from $i + v$ to $k$. Note that the components of $c$ and $E$ are determined with respect to the coordinate system of the base link of the set $(i + v)$ so that the orientation and position transforms from links $i + v$ to $j + 1$, $i + v U_{j+1}$ and $p_{i+v,j+1}$, are required in the computations. Note also that transform of inertial quantities associated with the links, back to the base of the manipulator, is not required here as has been the case in other work [9]. As in the general case for recursive doubling [18], the transforms needed on level $l$ are computed on level $l - 1$. Also, in the equations, note that $P_{i,i+1} = P_{i+1}, M_{i,i} = m_i, c_{i,i} = s_i$, and $E_{i,i} = I_i$ which are all initially given for each link $i$.

The ceiling function $\lceil \cdot \rceil$ and conditions on the range of the indices $j$ and $k$ have been used so that the equations are appropriate for any number of degrees of freedom, $N$. Note that the computation is required only if the number of degrees of freedom $N$ is greater than or equal to the number of the first link ($j + 1$) of the second of the sets of links to be combined (Eq. (4.11)).
Table 3: $O(\log_2 N)$ Parallel Algorithm for Computing the Inertia Matrix.

BEGIN11

\[ l_T := \lceil \log_2 N \rceil \]

FOR \( l := 1 \) TO \( l_T \) DO

BEGIN12

\[
\begin{cases}
  w & := 2^l \\
  w_T & := 2^{l_T - 1} \\
  w_{v_T} & := 2^{l-1}
\end{cases}
\]

FOR ALL \( u := 1 \) TO \( w_T \) AND
FOR ALL \( v := 1 \) TO \( w_{v_T} \) WITH

\[
\begin{cases}
  i & := w(u - 1) \\
  j & := w(u - 0.5) \\
  k & := w u
\end{cases}
\]

IF \((k > N)\) THEN \((k := N)\)

IF \((j + 1 \leq N)\) THEN DO

BEGIN13

\[
i^{+v}U_{k+1} := i^{+v}U_{j+1} + i^{+1}U_{k+1}
\]

\[
pi^{+v}_k := pi^{+v}_{j+1} + i^{+v}U_{j+1} \cdot pi^{+v}_{j+1}
\]

\[
m_i^{+v}_k := m_i^{+v} + m_{j+1}
\]

\[
c_i^{+v}_k := \frac{1}{m_i^{+v} + m_{j+1}} [m_{j+1} c_i^{+v,j} + m_{j+1,k} \left( i^{+v}U_{j+1} \cdot c_{j+1,k} + pi^{+v}_{j+1} \right)]
\]

\[
e_i^{+v}_k := i^{+v}U_{j+1} e_{j+1,k} + i^{+1}U_{k+1} + e_i^{+v} + e_{i^{+v}} +
\]

\[
m_{j+1} \left( \left( i^{+v}U_{j+1} \cdot c_{j+1,k} + pi^{+v}_{j+1} - c_i^{+v,k} \right) 1 -
\]

\[
\left( i^{+v}U_{j+1} \cdot c_{j+1,k} + pi^{+v}_{j+1} - c_i^{+v,k} \right) (i^{+v}U_{j+1} \cdot c_{j+1,k} + pi^{+v}_{j+1} - c_i^{+v,k})^T
\]

\[
m_{j+1} \left( (c_i^{+v,j} - c_i^{+v,k})^T (c_i^{+v,j} - c_i^{+v,k}) 1 -
\]

\[
(c_i^{+v,j} - c_i^{+v,k})(c_i^{+v,j} - c_i^{+v,k})^T
\]

END13

END12

FOR ALL \( i := 1 \) TO \( N \) DO

BEGIN14

\[
f_{i,i} := \sigma_i (z_0 \times M_i,N c_{i,N}) + \sigma_i (M_i,N z_0)
\]

\[
m_{i,i} := \sigma_i (E_{i,N} z_0) + c_{i,N} \times f_{i,i}
\]

\[
H_{i,i} := \sigma_i (n_{i,i} \cdot z_0) + \sigma_i (f_{i,i} \cdot z_0)
\]

END14

END11
BEGIN21
FOR ALL \( x := 1 \) TO \( N \) DO
BEGIN22
\( l_x := \lfloor \log_2 x \rfloor \)

FOR \( l := 1 \) TO \( l_x \) DO
BEGIN23
\[
\begin{align*}
  w &= 2^l \\
  u_T &= 2^{l_x - 1} \\
  v_T &= 2^{l - 1}
\end{align*}
\]

FOR ALL \( u := 1 \) TO \( u_T \) AND
FOR ALL \( v := 1 \) TO \( v_T \) WITH

\[
\begin{align*}
  i &= w(u - 1) \\
  j &= w(u - 0.5) \\
  k &= w u
\end{align*}
\]

IF \( (j + 1 \leq x \leq k) \) THEN DO
BEGIN24
\[
\begin{align*}
  f_{i+v,x} &= f_{j+1,x} \\
  n_{i+v,x} &= f_{j+1,x} (n_{j+1,x} + P_{i+v,j+1} \times f_{j+1,x}) \\
  H_{i+v,x} &= \sigma_{i+v}(n_{i+v,x} \cdot z_0) + \delta_{i+v}(f_{i+v,x} \cdot z_0)
\end{align*}
\]

END24
END23
END22
END21
The composite rigid body parameters as well as the diagonal components of the inertia matrix may be computed in \(O(\log_2 N)\) time and this is graphically depicted as Stage A of the computation in Fig. 14. In the figure, the numbers in the parentheses within a box give the associated links for which the computation is made. A "zeroth" level of computation is also shown in which the position and orientation transforms across each individual link are computed. Note that the angle for the first degree of freedom is not needed and that \(iU_{i+1}\) and \(iP_{i+1}^i\) are computed for link \(i\).

4.2.3 Parallel Algorithm for the Off-Diagonal Elements

To obtain the off-diagonal elements of the inertia matrix, the force and moment at joint \(i\) due to a unit acceleration there \((f_{i,i}^t\) and \(n_{i,i}^t)\) should be resolved to the previous joints \((f_{j,i}^t\) and \(n_{j,i}^t)\) for \(j \leq i\). This involves a total of \(N\) linear recurrences of size \(N, N-1, \text{etc.}\). Several approaches to parallel computation of the off-diagonal elements are first discussed in this section. Then an efficient approach, which uses the transforms computed in Stage A for the diagonal elements and which achieves an \(O(\log_2 N)\) time, is detailed.

First of all, computation of the off-diagonal terms may be computed in \(O(N)\) time if the parallel algorithm given in Table 1 is employed. This results in a computation time of \(K_1O(N) + K_2O(\log_2 N)\) for the entire inertia matrix where the \(K_1\) coefficient is relatively small. This is similar to the modified row-sweeping algorithm applied by Lee and Chang [9] when computing the inertia matrix for parallel forward dynamics computation.

The order of the computation may be further reduced if recursive doubling is applied to each of the recurrence equations needed to compute the off-diagonal elements of the inertia matrix. Computation of the largest recurrence, which gives
Figure 14: Flow of Data and Computation for the Parallel Algorithm ($N = 8$).
the elements of the last column of the inertia matrix, can then be achieved in $O(\log_2 N)$ time. Since each of the recurrences may be computed in parallel, the overall computation time for the off-diagonal elements is $O(\log_2 N)$. The major problem with this approach, however, is that the position and orientation transforms ($U$'s and $p$'s) across the links of a varying-size composite rigid body at the base end of the manipulator are computed independently. This results in redundant computation both within this stage of computation and with Stage $A$ for the diagonal elements.

The most efficient method for computing the off-diagonal elements is to use the transforms computed in Stage $A$ while yet completing the computation in $O(\log_2 N)$ time. But it should be understood that these transforms were basically computed for a manipulator of size $N$ only. However if the computational structure of Stage $A$ is considered in Stage $B$ for the off-diagonal elements, then the more efficient algorithm is achieved. Noting Fig. 14, computation of the off-diagonal elements of column 6 is shown. Judicious elimination of many of the computational blocks (shown with dashed boxes) results in effective use of the transforms available from Stage $A$ while still achieving $O(\log_2 N)$ time.

Note that the computation in the left part of Stage $B$ is generally not needed. This is shown implemented in the equations, in the last part of Table 3, through appropriate conditions on the indices $j$ and $k$. Essentially, the computation is not required unless the column number $x$ falls within the range of the second of the sets of links to be combined (Eq. (4.11)). In Fig. 14, the numbers in parentheses within a box give the indices for the $f$'s and $n$'s which are calculated. Not explicitly shown in the figure is the flow of the position and orientation transforms from Stage $A$ to Stage $B$ which are required for the computation of the off-diagonal terms.
The $O(\log_2 N)$ parallel algorithm is shown in its entirety in Table 3. When compared with Table 1, it may be noted that the total number of primitive operations has increased with the parallel algorithm. In general, the number of primitive operations increases as one attempts to decrease the order of the computation. In fact if the total number of operations decreased when going from a serial algorithm to a parallel algorithm, then the parallel algorithm should also be used on a serial processor since it becomes the most efficient serial algorithm as well.

4.3 Number of Processors Required

With parallel formulations to compute the inertia matrix in reduced order time, the total number of processors required generally increases. In this section, the relationship between the two will be explored. An idealized shared memory model will be assumed so that effects of I/O will not be considered. Detailed evaluation of these effects in the context of systolic architectures for the inertia matrix was presented in the previous chapter when considering the $O(N)$ algorithm.

The most efficient serial algorithm computes the $N^2$ components of the inertia matrix in $O(N^2)$ time [8]. In the previous chapter, a number of configurations of $N$ processors were used to implement the $O(N)$ algorithm developed there. For the $O(\log_2 N)$ algorithm developed in this chapter, the number of processors required may be determined in the following way. First of all, consider use of the $N/2$ processors needed in Stage A (Fig. 14) for Stage B. If they are successively used to compute the components of column $N$, column $N - 1$, etc. each in $O(\log_2 N)$ time, then all of the components of the inertia matrix may be computed in $O(N \log_2 N)$ time.

If separate sets of processors are assigned to each of the columns of the inertia matrix, then the $O(\log_2 N)$ computation time may be achieved. For each of
columns \( N, N-1, \ldots, N/2 \), at most \( N/2 \) processors are required. For each of the next \( N/4 \) columns, at most \( N/4 \) processors are required. For the full \( N \)-degree-of-freedom manipulator then, a maximum of

\[
\left( \frac{N}{2} \right) \left( \frac{N}{2} \right) + \left( \frac{N}{4} \right) \left( \frac{N}{4} \right) + \left( \frac{N}{8} \right) \left( \frac{N}{8} \right) + \cdots + \left( \frac{N}{2^T} \right) \left( \frac{N}{2^T} \right)
\]

processors are required in order to reduce the computation to \( O(\log_2 N) \). In Eq. (4.19), \( l_T = \lceil \log_2 N \rceil \) and it may be rewritten as

\[
\sum_{i=1}^{\lceil \log_2 N \rceil} \left( \frac{N}{2^i} \right) \left( \frac{N}{2^i} \right) = N^2 \sum_{i=1}^{\lceil \log_2 N \rceil} \frac{1}{(2^i)^2} = N^2 \sum_{i=1}^{\lceil \log_2 N \rceil} \frac{1}{4^i} \leq N^2/3.
\]

Note that \( N^2/3 \) gives an upper bound on the total number of processors required.

In general, the computation for a given column in Stage \( B \) may be more evenly distributed over the \( \log_2 N \) levels, thus further reducing the total number of processors. For instance, for column 6 (Fig. 14), the computations for \( H_{3,6} \) and \( H_{4,6} \) may be moved to level 2, resulting in 2 processors being sufficient. If the \( N(N - 1)/2 \) computations required in Stage \( B \) \( (N - 1 \) for column \( N, N - 2 \) for column \( N - 1, \) etc.) were evenly distributed over the \( \log_2 N \) levels, then a total of \( N(N - 1)/(2 \log_2 N) \) processors would be needed. This gives a lower bound on the total number of processors needed for the \( O(\log_2 N) \) algorithm. The actual number of processors required for \( N = 8 \) is given in Table 4. Note that the 13 required lies within the bounds previously determined (10,22).

Table 5 summarizes the number of processors needed for a desired order of computation. As expected, as the order of the computation decreases, the total number of processors required increases. The information presented here should be quite helpful when considering any specific parallel implementation of the inertia matrix.
Table 4: Minimum Number of Processors Needed in Stage $B$ for $N = 8$

<table>
<thead>
<tr>
<th>Column</th>
<th>Number of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td><strong>13</strong> <strong>Total</strong></td>
</tr>
</tbody>
</table>

Table 5: Number of Processors vs. Order of Computation

<table>
<thead>
<tr>
<th>Order of Computation</th>
<th>Number of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N^2$</td>
<td>1</td>
</tr>
<tr>
<td>$N \log_2 N$</td>
<td>$N/2$</td>
</tr>
<tr>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>$\log_2 N$</td>
<td>$\frac{N(N-1)}{2\log_2 N} \leq p \leq N^2/3$</td>
</tr>
</tbody>
</table>
4.4 Implementation of the $O(\log_2 N)$ Algorithm on a Cube Architecture

A cube architecture is employed to implement the diagonal elements of the inertia matrix in $O(\log_2 N)$ time and the upper off-diagonal elements in $O(N)$ time. The reason for the latter is that the computation of the $O(N)$ algorithm for the off-diagonal elements may be performed more efficiently on a cube network than the corresponding $O(\log_2 N)$ algorithm. The coefficient of $N$ in the $O(N)$ computation for the off-diagonal elements is relatively small; whereas, the I/O communication of the corresponding $O(\log_2 N)$ computation on a cube network is relatively high. That is, the $O(\log_2 N)$ algorithm for the off-diagonal elements becomes I/O bound when mapped onto the cube architecture.

As in the previous chapter, the I/O and idle times are fully considered. Also, in writing the microcode for each Robotics Processor, all features of the processor including its inherent pipelining and parallelism are fully utilized. As before, the performance indices used to evaluate the cube architecture are the compute time, I/O time, initiation rate, CPU utilization, speedup, and total memory required. Fig. 15 shows the cube configuration implementing the $O(\log_2 N)$ algorithm for the diagonal elements listed in the BEGIN11 block in Table 3 and the $O(N)$ algorithm for the off-diagonal elements listed in the BEGIN2 block in Table 1, using the Robotics Processor discussed in Chapter II.

Fig. 15 shows that computation of the composite rigid body parameters, the diagonal elements, and all the columns of the inertia matrix are performed in parallel as depicted in Table 3 and Fig. 14; whereas, the computation of the off-diagonal elements within a column is performed serially as shown in Table 1. It has been found that if the computation of the off-diagonal elements was performed in parallel on the cube configuration, the execution time would be much longer.
Figure 15: $N$-Processor Cube Configuration ($N = 8$).
Since they require relatively few computational steps, the associated I/O overhead and the lack of parallelism among the various subtasks contribute very heavily to the increase in execution time.

The performance characteristics of the 1-processor, \( N \)-processor in column form and cube configuration are summarized in Table 6. By comparing the column configuration to the cube configuration, it may be noted that the execution time for the cube (676 \( \mu s \)) is slightly smaller than the column (680 \( \mu s \)). Initiation rate is higher for the column form than the cube architecture. Also, the cube architecture requires more memory than the column form. Table 6 indicates that the average CPU utilization is close to the same value for the column and cube configurations. But, it should be pointed out that the PE's for the cube architecture perform redundant operations during some of the clock cycles so as to allow local communications at each level of the computation. This fact was not taken into account in calculating the CPU utilization.

Overall, the implementation of the \( O(N) \) algorithm using \( N \) processors in column form appears to give the best results for computing the manipulator inertia matrix. While the cube configuration has some advantages, interprocessor communications become a bottleneck due to the processor interconnection scheme.

### 4.5 Summary and Conclusions

This chapter has outlined the development of an \( O(\log_2 N) \) parallel algorithm for computing the \( N \times N \) inertia matrix for a robot manipulator. A listing of the algorithm is given in Table 3, and its flow of computation and data is shown in Fig. 14. In each case, the parallelism inherent in the algorithm is explicitly shown.

A recursive doubling technique [18] was used to achieve computational reduction over the \( O(N) \) parallel algorithm listed in Table 1, which was given in
Table 6: Architecture Parameters as a Function of the Number of Processors Used. (Numbers correspond to $N = 8$)

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>1</th>
<th>$N$ Column</th>
<th>$N(cube)$ Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_c (\mu s)$</td>
<td>$17N^2 + 80N$</td>
<td>$64N + 48$</td>
<td>$27N + 99\log_2 N - 1$</td>
</tr>
<tr>
<td>I/O Time</td>
<td>84</td>
<td>120</td>
<td>164</td>
</tr>
<tr>
<td>$T_{io} (\mu s)$</td>
<td>$N^2 + 3N - 4$</td>
<td>$16N - 8$</td>
<td>$6N + 40\log_2 N - 4$</td>
</tr>
<tr>
<td>Execution Time</td>
<td>1812</td>
<td>680</td>
<td>676</td>
</tr>
<tr>
<td>$T_e (\mu s)$</td>
<td>$18N^2 + 83N - 4$</td>
<td>$80N + 40$</td>
<td>$33N + 139\log_2 N - 62$</td>
</tr>
<tr>
<td>Initiation Rate</td>
<td>$1/1812$</td>
<td>$1/444$</td>
<td>$1/606$</td>
</tr>
<tr>
<td>$IR (\mu s)^{-1}$</td>
<td>$18N^2 + 83N - 4$</td>
<td>$44N + 92$</td>
<td>$31N + 139\log_2 N - 59$</td>
</tr>
<tr>
<td>Average CPU Utilization %</td>
<td>100.0</td>
<td>65.0</td>
<td>69.4</td>
</tr>
<tr>
<td>Speedup ($\eta$)</td>
<td>1.0</td>
<td>2.66</td>
<td>2.68</td>
</tr>
<tr>
<td>Number of 32-bit Registers</td>
<td>245</td>
<td>119</td>
<td>142</td>
</tr>
<tr>
<td>Control Memory Size (K bits)</td>
<td>11.55</td>
<td>18.8</td>
<td>31.0</td>
</tr>
<tr>
<td>Total Memory (K bits)</td>
<td>19.3</td>
<td>22.6</td>
<td>35.4</td>
</tr>
</tbody>
</table>
Chapter III, in computing the diagonal elements of the inertia matrix. It avoids transformation of inertial quantities, associated with each link, to base coordinates. An $O(\log_2 N)$ algorithm, which uses the position and orientation transforms computed for the entire manipulator when determining the diagonal elements, was then formulated to calculate the upper off-diagonal elements of the inertia matrix. Additional computation to determine the transforms over a varying-size composite rigid body (fixed set of links) at the base end of the manipulator is also avoided.

Then, the order of computation as a function of the number of processors used is tabulated in Table 5. The last entry in Table 5 shows a lower and upper bound on the number of processors needed to achieve an $O(\log_2 N)$ computation time. In general, as the order of computation decreases, the total number of processors required increases.

Finally, the $O(\log_2 N)$ algorithm for the diagonal elements and the $O(N)$ algorithm for the off-diagonal elements have been mapped onto a cube architecture. A reasonable reduction in execution time is obtained for calculating the composite rigid body parameters and diagonal elements; however, the architecture does not lend itself very well to the computation of the $O(\log_2 N)$ algorithm for the off-diagonal elements. Further work towards efficient mapping of the $O(\log_2 N)$ algorithm onto parallel architectures while accounting for communications (I/O) overhead is essential.

The basic objective is to minimize the computational latency while maximizing CPU utilization. For the computation of the inertia matrix, the computational latency is dictated by the precedence relationships that exist among the processes. The algorithms developed in this chapter and the previous chapter strictly adhere to the precedence relationships. One way to further reduce the computational latency is to increase concurrent task processing on multiple processors by develop-
ing a new computational model which does not follow the precedence relationships strictly. Since the computation of the inertia matrix has to be performed constantly during the course of a trajectory, a time history of its past values is available to predict its future values. Taking this fact into account, the new computational model based upon prediction may be used to relax precedence, and thus, to increase parallelism and decrease the computational latency. This approach will be further pursued in the next chapter.
CHAPTER V

NON-STRICIT COMPUTATIONAL MODELS

5.1 Introduction

In this chapter, the computational aspects of the manipulator inertia matrix are revisited with a new and potentially very effective computational approach which leads to a highly parallel algorithm suitable for multiprocessor implementation. Normally, to achieve speedup and lower effective delays in computationally intensive algorithms, specifically, computation of the inertia matrix, parallel algorithms and architectures adhering to the precedence relationships among processes, have been the only avenues explored. That is, the precedence relationships among the processes dictate the flow of computation. This traditional computational approach is referred to as Strict Computational Model. Use of parallelism in algorithms and/or architectures which are based upon a Strict Computational Model, in general, have been effective methods to increase system speed. In fact, those proposed in the previous two chapters for the manipulator inertia matrix followed the Strict Computational Approach.

However, more often, as the degree of parallelism within an algorithm which is based upon a Strict Computational Model increases, the number of primitive operations needed increases correspondingly. This rather general conclusion may be simply observed by comparing the $O(N)$ parallel algorithm presented in Chapter III with the $O(\log_2 N)$ parallel algorithm presented in Chapter IV for the inertia
matrix. Thus, parallel algorithms based on a Strict Computational Model, in general, tend to escalate system complexity through the need for more processors with higher I/O communication which requires more complex interconnection schemes. All these factors adversely impact practical issues such as system cost, interconnection cost, etc. as well as more basic issues such as task scheduling, process synchronization, etc.

To overcome the above problems, parallel algorithms in conjunction with new computational models may be devised to relax interprocess precedence which appears within the task flow graph shown in Fig. 4 for computing the inertia matrix. This high degree of dependency and process coupling is a remnant of the set of recurrence equations needed to compute the inertia matrix. Since the computational precedence will not be adhered to strictly, the approach is referred to as Non-Strict Computation.

The inertia matrix needs to be updated constantly as the manipulator maneuvers in space. So, a time history of the past values of the inertia matrix is available to predict its future values. Thus, in this chapter, Non-Strict Computational Models based upon prediction are applied to the inertia matrix to increase parallelism by relaxing precedence among processes, and as a result, to decrease the effective computational delays. Subsequently, throughput is increased due to increased parallelism at the process level while adding little or no additional hardware complexity. Implementation of such an algorithm having a high degree of parallelism on a parallel architecture will result in a shorter critical path from input to output, and consequently, in a lower computational delay.

The only negative side effect of any Non-Strict Computational Model is the error introduced in the results because strict precedence is not adhered to any longer. This problem may be remedied to a great extent by predicting the intermediate
results passed among the processors. Furthermore, the degree of prediction is another degree of freedom in achieving less corrupted results. As a part of the analysis of the Non-Strict Computational Approach, then, the effect of the degree of prediction will be explored.

Besides the data corruption induced by the Non-Strict Computational Approach, there is still error in data values simply due to computational delay. However, since the computational delay caused by the Non-Strict Computational Approach is expected to be much lower than the computational delay due to the Strict Computational Approach, then perhaps, the compound error in the data values due to both the computational delay and the data corruption due to the non-strict computation may be less than the data error due to the sheer computational delay in the Strict Computation. This type of error analysis will be conducted in a later section where simulation results are presented.

Previously, Inverse Dynamics has generally been implemented using either a recursive Newton-Euler or Lagrange-Euler formulation since explicit determination of the terms of the dynamic equations of motion, given in Eq. (3.1), is not computationally efficient. However, if computation of the individual terms are performed at multiple rates, then it may be that the latter approach, which involves explicit computation of the inertia matrix, will be the most efficient. In particular, while the computation of the inertia matrix is intensive, the inertia matrix itself is slowly time-varying. Thus, its computation may be performed at a much lower rate than the computation of some of the other terms. This concept is referred to as multiple rate computation and it will be applied to compute the acceleration-dependent term in the dynamic equations of motion. That is, in order to conduct a quantitative performance analysis, joint torques are computed by \( \tau = H(q)\dot{q} \). Since the inertia matrix is slowly time-varying, its computation may
be performed at a much lower rate than the computation of the joint torques. Parallel architectures computing the inertia matrix will be idle and available for computing other tasks or terms such as the Jacobian. In so doing and predicting joint torques (extrapolating), noticeable improvement in CPU availability is achieved.

The Non-Strict Computational Approach applied to a simple inverted pendulum was introduced in [19] to relax precedence so as to reduce the effective computational delay from input to output. The task flow graph for the Inverse Dynamics computation of an inverted pendulum considered in [19] has a high degree of parallelism among the processes. A multiprocessor implementation of it, based upon a Strict Computational Model, would result in a significant speedup without even corrupting the data. However, the task flow graph for the inertia matrix given in Fig. 4 exhibits tightly-coupled sequential processes defined by a set of recurrence equations. This has placed a tight precedence relationships among the processes, and as a result, the degree of parallelism achievable through a Strict Computational Model is limited. Hence, the Non-Strict Computational Models based upon prediction are more effective to increase parallelism for the computation of the inertia matrix.

Also, the concept of prediction, zero-order prediction specifically, has been applied to the Inverse Dynamics problem to decompose the Newton-Euler formulation [72] in order to increase parallelism. In [72], the concept of non-strict computation was not formally and explicitly introduced. Furthermore, higher orders of prediction were not considered either, despite the fact that noticeable joint torque errors were obtained for fast trajectories. These issues plus such concepts as multiple rate computation and the effect of sampling frequency are fully considered in this chapter.
In the following, reference to three terms is constantly made: exact, strict, and non-strict computation. Exact computation refers to an ideal computation with no error and no delay; whereas, Strict Computation is the exact computation with computational delay included. On the other hand, Non-Strict Computation refers to a computation with less computational delay than the Strict Computation but with corrupted data.

In this chapter, a parallel algorithm based upon the Non-Strict Computational Model is presented. The development of the algorithm is based upon the $O(N)$ parallel algorithm for computing the inertia matrix given in Chapter III. The multiple rate concept is then discussed. Simulation results are then presented to show the effectiveness of the Non-Strict Computational Approach plus the multiple rate concept. Finally, the work is summarized, and conclusions are made.

### 5.2 An Overview of the $O(N)$ Parallel Algorithm

In Chapter III, the development of an $O(N)$ parallel algorithm to compute the manipulator inertia matrix was outlined. Various systolic architectures were proposed to implement the algorithm which reduces the order of computation from $O(N^2)$ to $O(N)$. The column configuration shown in Fig. 5 was concluded to have the best overall performance. The critical path for the computation involves the determination of $c_i$ and $E_i$ form $c_{i+1}$ and $E_{i+1}$. This fact was demonstrated in the timing diagram for the column configuration (Fig. 6). It is clear from Fig. 6 that the critical path is from the input of the $N^{th}$ processor to the output of the $1^{st}$ processor. It is this data dependency among processors that has placed a lower bound on the compute time delay. By taking a Non-Strict Computational Approach, which is the subject of the next section, we hope to relax this dependency.
5.3 Non-Strict Computational Algorithm

As mentioned previously, the Non-Strict Computational Model may be used effectively to relax precedence in computationally intensive tasks which need to be performed in real time. Relaxation of precedence leads to greater subprocess parallelism and a shorter critical path from input to output. Consequently, a lower compute time delay is achieved. The negative side effect of Non-Strict Computation is that the results are imprecise and corrupted. One way to improve the accuracy and to reduce data corruption is to ensure that the intermediate results produced by the nodes are carefully controlled to be close to the precise (or strict) values.

The use of prediction is one way to achieve the needed accuracy to ensure the integrity of the data. Simply, all or some of the parameters passed among nodes in a task graph may be predicted forward in time. Fig. 16 explicitly shows how predictors may be assigned to each node to relax precedence for the case of the column form shown in Fig. 5. In this case, the composite center of mass (c) and the composite moment of inertia (E) are predicted. Thus, processor i uses predicted values of (c_{i+1}) and (E_{i+1}) to compute (c_i) and (E_i) without waiting for the output of processor i + 1. This simply implies that the arrows connecting processors indicate data flow but not precedence in computation. As a result, all N processors may carry on their corresponding computation in parallel. Thus, maximum process-level parallelism is achieved.

This point is clearly shown in the timing diagram in Fig. 17 corresponding to the Non-Strict Algorithm given in Fig. 16. Note that, now, the critical path is dominated by the computation of processor N only. Comparison of Figs. 6 and 17 reveals the fact that maximum process-level parallelism is achieved. This results in
Figure 16: Non-Strict Computational Approach Showing the Use of Prediction.
Figure 17: Timing Diagram for Non-Strict Computation of the Inertia Matrix.
a speedup in the computation through a lower effective compute time delay from input to output. Thus, the effect of the Non-Strict Approach with prediction is to alter the critical path in order to lower the effective delay. Compute time delay is directly related to the phase margin and thus to the stability of a real-time control system. The lower the effective delay, the higher the margin of stability.

Fig. 18 shows this reduction in the effective delay pictorially for a general time-dependent variable \( \tau \). Three distinct curves corresponding to exact, strict, and non-strict computations are shown. The delay, \( \Delta t_s \), is the strict computational delay; whereas, \( \Delta t_{ns} \) is the non-strict computational delay. Note that, \( \Delta t_{ns} < \Delta t_s \) if the non-strict approach is properly applied. In addition, since the critical path
is shorter for the Non-Strict Computation than for the Strict Computation, the
initiation rate and thus the sampling rate is generally higher for the Non-Strict
Computation. That is, $\delta t_{ns} < \delta t_s$. Also, as shown in Fig. 18, the signal $T$ is
held constant during the entire sampling period for both the strict and non-strict
curves. This sample and hold behavior gives rise to the “stair-step” response. The
effective delay ($T_{ed}$) may then be approximated as the computational delay ($T_c$)
plus one-half of the sampling period ($T_s$). That is,

$$T_{ed} = T_c + \frac{1}{2} T_s,$$

where $T_{ed} = \Delta t_s$ for the Strict Computation or $T_{ed} = \Delta t_{ns}$ for the Non-Strict
Computation. Clearly, the effective delay is a function of two independent vari­
ables: computational delay and sampling frequency. As discussed previously, the
Non-Strict Computational model may be employed to reduce computational delay
and thus to reduce the effective delay. Also, suggested by Eq. (5.1), the higher the
sampling frequency, the lower the effective delay would be.

Thus, Non-Strict Computation with prediction increases process-level paral­
lelism. This directly impacts speedup through a lower effective delay, as mentioned
previously. These desired enhancements are attainable at the expense of corrupted
output results. From Fig. 18, note that the sampled points on the discrete non-
strict curve do not necessarily lie on the continuous dotted curve, which is the
exact curve delayed by the non-strict computational delay. Thus, the data are cor­
rupted due to both delay and Non-Strict Computation. The order of prediction is,
however, a degree of freedom which provides a compromise between the severity
of output data corruption and the intensity of the computation. By increasing
the order of prediction, additional computational steps are needed, but the output
error decreases. First-order prediction on $(c_i)$ and $(E_i)$ as opposed to zero-order
prediction decreases the error noticeably at the expense of very minimal computa-
tional steps. For zero-order prediction, simply the immediate previous value is
used for the next sampling time. This is just the simplest form of extrapolation
and gives:

\[ H_{j,i}(t) = H_{j,i}(c_i(t-1), E_i(t-1)) \quad \text{for} \quad i, j = 1, \ldots, N. \]  
(5.2)

First-order prediction, on the other hand, uses the last two data points to linearly
predict the current value of the function. This requires additional but minimal
computational steps. The trade off between the computational accuracy and the
extra computations incurred by the first-order predictor is well justified for the
computation of the inertia matrix as will be demonstrated in Section 5.4. So,
as needed, the order of the predictor may be increased to achieve more accurate
results at the expense of more computational steps. For the problem under study
here, a first-order predictor furnishes good results.

It has already been mentioned that the effective delay may be reduced by
lowering the compute time delay through Non-Strict Computation. Eq. (5.1) im-
plies that to further decrease the effective delay substantially, the sampling fre-
quency dependency should be eliminated or significantly diminished. By applying
a multiple rate concept combined with output prediction to either the Strict or
Non-Strict Computation, this may be accomplished. That is, as shown in Fig. 18
for the non-strict curve, by predicting \( \tau \) at a higher rate than the computation of
the intermediate results, the effective delay would be primarily a function of com-
putational delay, but not strongly a function of the sampling frequency. This idea
is sketched in the circled portion of Fig. 18 by small step-size curves approaching
the strict curve as the output rate increases or as the order of prediction on \( \tau \)
increases. Thus, a minimum bound on the effective delay is certainly achievable at
substantially lower sampling (computational) frequencies for the inertia matrix. A lower sampling frequency implies that CPU's are now available for other necessary computations. So, multiple rate computation in conjunction with prediction on the output results in a higher CPU availability. In addition, the sampling frequency does not play an important role in the determination of the effective delay any longer.

As mentioned in the Introduction, the acceleration-dependent part of the joint torques may be computed \( \tau = H(q)\ddot{q} \) for a quantitative performance analysis. The joint torques may be computed at a different rate than the computation of the inertia matrix. This is referred to as multiple rate computation of the joint torques. Thus, multiple rate computation in conjunction with prediction on the joint torques results in a higher CPU availability. That is, the CPU's are now available for other necessary computations (e.g. computation of the Jacobian) besides just the inertia matrix. The next section provides a number of quantitative results which demonstrate the above concepts.

5.4 Simulation Results

In order to analyze the performance of the Non-Strict Computational Algorithms, the PUMA 560 robot manipulator is commanded to go through a trajectory, based upon a fifth-order polynomial for joint position, at various acceleration rates and sampling frequencies. At the outset, four sets of parameters are specified: the initial position and orientation, the goal position and orientation, sampling frequency, and total traveling time \( t_f \). The initial position is arbitrarily chosen to be the zero position. The following Cartesian transform specifies the final position and orientation and causes the joint angles to sweep a wide range of the workspace.
All necessary dynamic parameters needed for the simulator are derived from [73]. The model for the PUMA 560 is summarized in Appendix A.

The position, velocity, and acceleration profiles for joint angle 1 are shown in Fig. 19 for \( t_f = 4 \text{ sec} \). Other joints exhibit similar profiles at lower magnitudes. Thus, most of the discussion in this section is based upon joint 1 only. It should be pointed out that a 4 sec total traveling time is the fastest time that the PUMA 560 is able to travel without exceeding the joint torque limits for the initial and final positions chosen.

As mentioned before, the joint torques are computed using the dynamic equations of motion, but using only the accelerated-dependent term affected by the inertia matrix. For joint 1, the torque computed by \( \tau = H(q)\ddot{q} \) is shown in Fig. 20 (exact case with no delay).

Assuming that the Strict Computation of the inertia matrix for a 6 degree of freedom manipulator \( (N = 6) \) takes approximately 1.75 msec, then, based upon the results obtained in Chapter III and Fig. 17, the initiation period for the Non-Strict Computation would be close to 60% of this value or approximately 1.0 msec. Thus, for Strict Computation of the manipulator inertia matrix, variation of the joint torques is the same as the exact case except that all the curves are delayed by 1.75 msec. Based upon a 1.0 msec initiation rate for the Non-Strict Computation of the inertia matrix, the \((a + b + c)\) computation (Fig. 17) is approximately 63% of the overall computation. Thus, the computational delay for the non-strict
Figure 19: Trajectory Profile for Joint Angle #1 (t_f = 4 sec).
Figure 20: Exact Computation of Joint Torque 1.
case will be set to 0.63 msec. The maximum sampling frequency for the Strict Computation is \( \frac{1}{770\text{ms}} = 570\text{Hz} \); whereas, the maximum sampling frequency for the Non-Strict Computation is \( \frac{1}{\text{ms}} = 1\text{KHz} \). The reason behind achieving a higher sampling frequency (from 570 to 1 KHz) is due to increased parallelism as mentioned previously. Thus, a higher speedup and a higher initiation rate are consequently obtained. It is clear that under the above operating conditions, an approximate 43% improvement in the initiation rate is achieved.

5.4.1 Non-Strict Results with Zero-Order Output Prediction

In order to first investigate the effects of sampling frequency and multiple rate computation, the simulator is used to compute joint torques at various sampling frequencies. The effective delay is computed by an incremental time advancement of the exact data points until a best fit is obtained with the non-strict data points. The standard deviation is used to evaluate the best fit. While searching for the best fit between the exact and non-strict data points, there is a point in time when standard deviation reaches its minimum value. This time is reported as the effective delay. Mathematically,

\[
\Delta t_{ns} : \min \left\{ (\delta T)^2 = \int_{t=0}^{t=\Delta} \left( T^{\text{exact}}(t + \Delta t_{ns}) - T^{\text{non-strict}}(t) \right)^2 dt \right\}
\]

where \( \Delta \) is the initial (startup) delay due to recursion in the non-strict computation and varies as a function of joint number. The above definition for obtaining the effective delay is applied to each joint separately.

Table 7 shows the effective computational delays for all 6 joints for various sampling frequencies. For every joint, three rows of data are given. The first row is the effective delay using Strict Computation, the second and third rows correspond to the zero-order and first-order prediction on internal I/O parameters.
Table 7: Effective Computational Delays with Zero-Order Output Prediction ($t_f = 4\ sec$). (First row of three – Strict, second row – Zero-Order Non-Strict, third row – First-Order Non-Strict)

<table>
<thead>
<tr>
<th>Joints</th>
<th>$f_s = 72.5\ Hz$</th>
<th>$f_s = 145\ Hz$</th>
<th>$f_s = 285\ Hz$</th>
<th>$f_s = 570\ Hz$</th>
<th>$f_s = 1\ KHz$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.66</td>
<td>5.19</td>
<td>3.47</td>
<td>2.62</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>9.11</td>
<td>4.81</td>
<td>2.71</td>
<td>1.68</td>
<td>1.22</td>
</tr>
<tr>
<td></td>
<td>7.52</td>
<td>4.07</td>
<td>2.36</td>
<td>1.51</td>
<td>1.12</td>
</tr>
<tr>
<td>2</td>
<td>8.67</td>
<td>5.19</td>
<td>3.47</td>
<td>2.62</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>7.55</td>
<td>4.09</td>
<td>2.36</td>
<td>1.51</td>
<td>1.13</td>
</tr>
<tr>
<td></td>
<td>7.54</td>
<td>4.08</td>
<td>2.36</td>
<td>1.51</td>
<td>1.13</td>
</tr>
<tr>
<td>3</td>
<td>8.68</td>
<td>5.21</td>
<td>3.48</td>
<td>2.62</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>7.53</td>
<td>4.09</td>
<td>2.36</td>
<td>1.51</td>
<td>1.14</td>
</tr>
<tr>
<td></td>
<td>7.54</td>
<td>4.09</td>
<td>2.36</td>
<td>1.51</td>
<td>1.14</td>
</tr>
<tr>
<td>4</td>
<td>8.64</td>
<td>5.17</td>
<td>3.46</td>
<td>2.61</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>7.43</td>
<td>4.01</td>
<td>2.32</td>
<td>1.49</td>
<td>1.09</td>
</tr>
<tr>
<td></td>
<td>7.53</td>
<td>4.05</td>
<td>2.35</td>
<td>1.50</td>
<td>1.10</td>
</tr>
<tr>
<td>5</td>
<td>8.66</td>
<td>5.19</td>
<td>3.47</td>
<td>2.62</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>7.53</td>
<td>4.08</td>
<td>2.36</td>
<td>1.51</td>
<td>1.12</td>
</tr>
<tr>
<td></td>
<td>7.54</td>
<td>4.08</td>
<td>2.36</td>
<td>1.51</td>
<td>1.12</td>
</tr>
<tr>
<td>6</td>
<td>8.67</td>
<td>5.20</td>
<td>3.47</td>
<td>2.62</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>7.53</td>
<td>4.08</td>
<td>2.36</td>
<td>1.50</td>
<td>1.13</td>
</tr>
<tr>
<td></td>
<td>7.53</td>
<td>4.08</td>
<td>2.36</td>
<td>1.51</td>
<td>1.13</td>
</tr>
</tbody>
</table>
passed among processors, respectively. The time increment used in computing the effective delay is 10 \mu \text{sec}; so, the resolution for the data points in Table 7 is \pm 10 \mu \text{sec}.

From Table 7, it may be observed that for a given joint, the effective delay, in general, decreases in a given column. That is, Non-Strict Computation offers a lower effective delay than the Strict Computation, as is expected. For example, there is about a 42% reduction in the effective delay for joint angle 1 at 570 Hz, from 2.62 \text{msec} to 1.51 \text{msec} corresponding to first-order prediction. Note that the major reduction in the effective delay, about 36%, is due to Non-Strict Computation (from 2.62 \text{msec} to 1.68 \text{msec}) and only 6% reduction in effective delay is due to the order of prediction (from 1.68 \text{msec} to 1.51 \text{msec}). Thus, the order of prediction does not have a major impact on the effective delay. A more reasonable comparison is the effective delay between the Strict Computation at its highest frequency (570 Hz) and the Non-Strict Computation (first-order prediction) at its highest frequency (1 KHz). In this case, there is about 57% reduction in the effective delay, from 2.62 \text{msec} to 1.12 \text{msec}. This reduction holds for all joints and corresponds to a speedup factor of 2.3.

By scanning a given row for any of the joints, it is obvious that the effective delay decreases substantially as the sampling frequency increases for either Strict or Non-Strict Computation. There is about a 70% reduction in the effective delay for joint 1 by increasing the sampling frequency from 72.5 Hz to 570 Hz in the Strict Computation; whereas, there is approximately 85% reduction in the effective delay by increasing the sampling frequency from 72.5 Hz to 1 KHz for the Non-Strict Computation (zero-order or first-order) for the same joint.

The above findings have reemphasized two major results: the effectiveness of the Non-Strict Computation in reducing the effective delay and more importantly
the dependency of the effective delay on the sampling frequency. Close observation of the numbers in Table 7 confirm the validity of Eq. (5.1). For example, at $t_s = j_s = \frac{1}{570} = 1.75$ msec, the effective delay is $0.87$ msec (one-half sampling period) plus $1.75$ msec (strict computational delay) which is equal to $2.62$ msec, and is equal to the number obtained through simulation. For the Non-Strict Computation, the computational delay is only $0.63$ msec as opposed to $1.75$ msec. Consequently, for the Non-Strict Computation, the sampling frequency has a higher weighting factor than the computational delay toward the determination of the effective delay.

5.4.2 Non-Strict Results with Multiple Rate Computation

As it was proposed in a previous section, multiple rate computation plus output prediction may be incorporated to eliminate or to diminish the effect of the sampling frequency. Results similar to those given in Table 7 are tabulated in Table 8 in which first-order output prediction on joint torques is applied. Now, for joint angle 1, there is a 63% reduction in the effective delay, from $1.75$ msec at $f_s = 570$ Hz to $0.64$ msec at $f_s = 1$ KHz. This corresponds to a factor of 2.7 speedup. Also, comparing the two tables shows a much lower effective delay at a given frequency for all the joints. In Table 8, the effective delay is more influenced by computational delay and less by the sampling frequency. This result is very pronounced at high frequencies. By scanning the rows for a given joint, it is clear that the effective delay approaches its minimum bound, the computational delay which is equal to $0.63$ msec, at much lower frequencies. For example, the third row for any joint indicates that the effective delay converges to the computational delay at frequencies as low as 72.5 Hz. This frequency corresponds to a $13.8$ msec sampling period as compared to a $1.0$ msec time needed to compute the manip-
Table 8: Effective Computational Delays with First-Order Output Prediction ($t_f = 4$ sec). (First row of three – Strict, second row – Zero-Order Non-Strict, third row – First-Order Non-Strict)

<table>
<thead>
<tr>
<th>Joints</th>
<th>$f_s = 72.5$ Hz</th>
<th>$f_s = 145$ Hz</th>
<th>$f_s = 285$ Hz</th>
<th>$f_s = 570$ Hz</th>
<th>$f_s = 1$ KHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.76</td>
<td>1.75</td>
<td>1.75</td>
<td>1.75</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2.26</td>
<td>1.39</td>
<td>1.00</td>
<td>0.82</td>
<td>0.74</td>
</tr>
<tr>
<td></td>
<td>0.65</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td>2</td>
<td>1.77</td>
<td>1.75</td>
<td>1.75</td>
<td>1.75</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0.69</td>
<td>0.65</td>
<td>0.65</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td></td>
<td>0.67</td>
<td>0.65</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td>3</td>
<td>1.73</td>
<td>1.74</td>
<td>1.75</td>
<td>1.75</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0.63</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td></td>
<td>0.63</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td>4</td>
<td>1.85</td>
<td>1.77</td>
<td>1.76</td>
<td>1.75</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0.65</td>
<td>0.62</td>
<td>0.62</td>
<td>0.63</td>
<td>0.63</td>
</tr>
<tr>
<td></td>
<td>0.74</td>
<td>0.66</td>
<td>0.65</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td>5</td>
<td>1.78</td>
<td>1.76</td>
<td>1.75</td>
<td>1.75</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0.68</td>
<td>0.65</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td></td>
<td>0.68</td>
<td>0.65</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td>6</td>
<td>1.75</td>
<td>1.75</td>
<td>1.75</td>
<td>1.75</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0.65</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td></td>
<td>0.65</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
</tr>
</tbody>
</table>
ulator inertia matrix. In other words, by using a sampling frequency of 72.5 Hz, every processor is now available 12.8 msec of the time. This implies a 93\% CPU availability.

By employing multiple rate computation with output prediction, the following three major results are attainable:

1. A good speedup factor,

2. A low effective delay which is more strongly related to the computational delay and less dependent upon the sampling frequency, and

3. A high CPU availability for low sampling frequencies.

5.4.3 Error Analysis

The only negative side effect of Non-Strict Computation is the output error due to data corruption. This effect is indirectly displayed in Tables 7 and 8. For example in Table 7, for the Non-Strict Computation at 1 KHz with zero-order output prediction, the effective delay is about 0.5 msec (one-half sampling period) plus 0.63 msec (non-strict computational delay) which is equal to 1.13 msec as compared 1.22 ± 0.01 msec for zero-order prediction and 1.12 ± 0.01 msec for first-order prediction. The major discrepancy between 1.13 and 1.22 msec is because of data corruption due to the non-strict nature of the computation. But, note that, 1.13 msec is very much comparable with 1.12 msec. This simply emphasizes the fact that higher-order prediction (first-order in this case) reduces the error significantly. This result may also be revealed from Table 8 by simply comparing the second and the third row for joint angle 1. For instance, at a 72.5 Hz sampling frequency, the effective delay is 2.26 msec when zero-order prediction is employed;
whereas, this number approaches its minimum value of 0.63 msec when first-order prediction is employed.

To more directly show the effect of first-order prediction on the joint torques computed using $\tau = H(q)\ddot{q}$, the following performance indices are computed.

\[
\begin{align*}
\tau^{es} &= \tau^{exact} - \tau^{strict}, \\
\tau^{en} &= \tau^{exact} - \tau^{non-strict}, \\
\tau^{bn} &= \tau^{bestfit} - \tau^{non-strict},
\end{align*}
\]

where non-strict corresponds to both zero-order and first-order prediction. Overall, this corresponds to five different data sets which are graphically shown in Fig. 21 for joint #1 only. The percentage of the peak-to-peak variation of all five curves is used as a performance measure of output error due to delay and/or data corruption. Thus, comparison of $\tau^{en}$ and $\tau^{bn}$ for zero-order prediction based upon this performance measure reveals that approximately 83% of the error is due to delay and 17% of the error is due to non-strict data corruption. On the other hand, comparison of $\tau^{en}$ and $\tau^{bn}$ for first-order prediction indicates that 98% of the error is due to delay and only 2% of the error is due to non-strict data corruption. This simply demonstrates that first-order prediction significantly reduces the error. Comparison of the two $\tau^{bn}$ curves for zero-order and first-order prediction is another way to restate the above finding. Simply put, a 90% improvement in error is achieved when first-order prediction is employed. Comparison of $\tau^{es}$ and $\tau^{bn}$ for first-order prediction indicates a 98% overall improvement in error due to either delay or data corruption. Finally, the torque error at higher frequencies, for example 1 KHz which are not explicitly shown, is generally lower by an order of magnitude.
Figure 21: Joint Torque Error for Joint Torque #1 at \( f_s = 72.5 \text{ Hz} \).
The results obtained above indicate that a combination of first-order output prediction and first-order prediction on intermediate result values passed between processors results in the best overall response. First-order output prediction allows a reduction in the acceptable sampling frequency (72.5 Hz instead of 1 KHz), increased CPU availability (by about 93%), and a 2.7 speedup factor (i.e. lower effective delay); whereas, first-order prediction on intermediate result values reduces the error created by Non-Strict Computation to a negligible amount.

5.4.4 Effect of Trajectory Profile

The effective delays illustrated in Tables 7 and 8 vary as the trajectory changes and, of course, as the acceleration changes. To demonstrate the latter effect, the total traveling time is changed. By increasing the acceleration, no change in speedup is expected. But, the sampling period significantly impacts the effective delay. Under the exact operating conditions as before, with the exception of $t_f = 0.4 \text{ sec}$, a 0.64 msec effective delay is achieved at an approximate 290 Hz sampling frequency instead of the 72.5 Hz obtained at $t_f = 4 \text{ sec}$. This corresponds to a 71% CPU availability which is still substantial. In addition, reducing the total traveling time by a factor of 10, which is not realizalbe with the PUMA 560, increases the absolute differential torque error by a factor of 1000. Nonetheless, similar improvements in the percentage of error is attainable when first-order prediction is applied.

When lowering the total traveling time and accelerations, the effective delay approaches the computational delay at frequencies lower than 72 Hz. This simply implies even higher CPU availabilities. Also, the magnitude of the differential torque error decreases correspondingly. Thus, acceleration profile has a direct impact on the effective delay and CPU availability.
5.5 Summary and Conclusions

This chapter has outlined the development of a highly parallel algorithm based on the Non-Strict Computational Approach for the manipulator inertia matrix. The computation of the inertia matrix is recursive in nature with a high degree of coupling among processes. However, a high degree of parallelism can be achieved by relaxing precedence. Consequently, a shorter critical path from input to output, a higher speedup and throughput, and lower effective computational delay are obtained at the expense of corrupted data. In order to ensure the integrity of the results, prediction has been applied with precedence relaxation so as to control the error between the non-strict and strict values. In addition, the order of prediction provides an additional degree of freedom to lower the severity of data imprecision at the expense of extra but minimal computational steps.

As mentioned previously, the lower bound for the complexity of the computation of the inertia matrix, based on Strict Computation, is of $O(\log_2 N)$ as developed in Chapter IV. However, the Non-Strict Computational technique employed here achieves an absolute lower bound on the computation of the inertia matrix. That is, the order of computation is constant ($O(k)$) and is not a function of $N$, the number of degrees of freedom of the manipulator.

Multiple rate computation combined with output prediction has also been applied to determine the acceleration-dependent term of the dynamic equations of motion. The computation of the inertia matrix was performed at much lower rates than the rate of computation of the joint torques with highly improved performance for the case of Non-Strict Computation. In so doing, the effect of sampling rate, which greatly influences the effective computational delays as given in Eq. (5.1), was drastically diminished for both the strict and non-strict case. This led to an
approximate 90% CPU availability during which the parallel architecture implementing the inertia matrix is free. This time may be utilized to compute other tasks.

All the foregoing concepts have been demonstrated by simulating a multiprocessor architecture computing the acceleration-dependent part of the dynamic equation of motion for the PUMA 560. The effective delay is computed for both the Strict and Non-Strict (for both zero-order and first-order prediction) Computations at various sampling frequencies as shown in Table 7. Table 8 presents similar data when multiple rate computation combined with first-order prediction on the output is used. Comparison of the data in the two tables simply reflects the fact that sampling frequency has a negligible effect on the effective delay when joint torques are computed at a higher rate with first-order output prediction than the computation of the inertia matrix. Also, a reasonable effective delay is achievable at much lower sampling frequencies leading to a high CPU availability.

The joint torque errors due to both computational delay and Non-Strict Computation for joint #1, as a typical case, are shown in Fig. 21. Comparison of the curves in the figure indicates that the error may be noticeably reduced if the order of prediction is increased. Finally, the effect of acceleration and total traveling time have been discussed.

This chapter, in general, has presented a practical case study and the foundation for the Non-Strict Computational Approach in highly coupled algorithms. Further applications of the Non-Strict Computational Approach need to be explored. Then, a formalization of the approach may be presented for a general class of computationally intensive problems. Related scheduling issues, both static and dynamic, for a multiprocessor architecture, are essential in attempting to effectively apply the Non-Strict Computational Models.
CHAPTER VI

SUMMARY AND CONCLUSIONS

6.1 Summary

This chapter presents a summary of this dissertation. Then, specific concluding remarks are stated and used to project and recommend future work in this field of study.

The computational aspects of the manipulator inertia matrix used in dynamic control and real-time simulation has been approached through parallel algorithms and architectures. The development of an $O(N)$ parallel algorithm based upon recursive computation of the inertial parameters of sets of composite rigid bodies [8] was first formulated. The algorithm was generally based on employment of the computationally efficient Newton-Euler method for Inverse Dynamics. The algorithm is tabulated in Table 1 showing much of the parallelism inherent in the algorithm.

One- and two-dimensional systolic architectures consisting of 1, $N$, and $N(N + 1)/2$ processors were presented for implementing the $O(N)$ parallel algorithm. The architectures presented were based upon a VLSI Robotics Processor [14] to exploit fine-grain parallelism. I/O time and idle time due to processor synchronization as well as CPU utilization and on-chip memory size were fully included in the evaluation of all proposed architectures. These performance parameters along with speedup and latency indicate the feasibility and effectiveness of
the designs. Good results are obtained for $N$-processor and $N(N + 1)/2$-processor configurations which give a compute time delay of $O(N)$. Overall, the column configuration of $N$ processors exhibited the best performance when cost-effectiveness and interconnection cost, which are both related to the number of processors used, are considered. While, in general, increasing the number of processors will decrease the total execution time delay, the results indicate that increased communications overhead and idle time (due to increased problems of synchronization) negate many of the advantages of going to more processors. In addition, the average CPU utilization decreases and thus decreases the overall cost-effectiveness. However, the initiation rate increases significantly with increasing numbers of processors. A higher initiation rate may have a positive impact on control system stability.

The development of an $O(\log_2 N)$ parallel algorithm for computing the manipulator inertia matrix was presented next. A listing of the algorithm is given in Table 3, and its flow of computation and data is shown in Fig. 14 where the inherent parallelism in the algorithm is explicitly shown. As before, the algorithm is based upon the composite rigid body method. Recursive doubling was used to reformulate the linear recurrence equations which were required to compute the diagonal elements of the inertia matrix. It resulted in $O(\log_2 N)$ levels of computation. Previously, the inertial quantities, associated with each link, were transformed to base coordinates; however, the procedure presented avoids transformation of inertial quantities to base coordinates. Computation of the off-diagonal elements involves $N$ linear recurrences of varying-size and a new method, which avoids redundant computation of position and orientation transforms for the manipulator, was developed.

The relationship between the number of processors required and the order of the computation was also given for several versions of the algorithms for the iner-
tia matrix. Table 5 shows a lower and upper bound on the number of processors needed to achieve an $O(\log_2 N)$ computation time. In general, as the order of computation decreases, the total number of processors required increases. Finally, the algorithm was mapped onto a cube architecture and its performance was compared against the systolic architectures presented earlier. It was concluded that the column configuration results in the best overall response when both the performance parameters and practical issues such as system cost and interconnection cost are taken into account.

The algorithms presented in Chapters III and IV use a Strict Computational Model in which the computation strictly adheres to the precedence dictated in the task graph. However, in Chapter V, a new computational model was employed which relaxes precedence. It was referred to as a Non-Strict Computational Model and increases parallelism and to thus decrease the effective computational delay by shortening the critical path from input to output. Lower effective delays were achieved by increasing throughput due to increased parallelism at subprocess level while requiring little additional complexity. The drawback behind the Non-Strict Computational Model is that intermediate and thus the final result values are corrupted. Prediction was employed along with precedence relaxation so as to maintain control over the differences between strict and non-strict data values. The degree of prediction was increased to limit the severity of data corruption at the expense of additional but minimal computational steps.

In Chapter IV, the lower bound for the computation of the inertia matrix was shown to be of $O(\log_2 N)$ when strict computational precedence is followed. However, in Chapter V, the absolute lower bound of $O(k)$ computation time, where $k$ is just a constant, was achieved with the Non-Strict Computational Approach.
The concept of multiple rate computation was then incorporated to achieve a noticeable improvement in CPU availability by eliminating or greatly diminishing the sampling frequency influence on the effective computational delay. This resulted in better than a 90% CPU idle time during which the parallel architecture used to compute the inertia matrix may be engaged in computation of some other tasks. That is, high CPU availability was achieved.

All the concepts in Chapter V were demonstrated by simulating a multiprocessor architecture computing the acceleration-dependent part of the dynamic equations of motion for the PUMA 560. The effective delay was computed for both the Strict and Non-Strict Computations (for both zero-order and first-order prediction) at various sampling frequencies as shown in Table 7. Table 8 presents similar data when multiple rate computation along with first-order prediction on the output is used. Comparison of the data in the two tables reflect the fact that sampling frequency has a negligible effect on the effective delay when joint torques are computed at a higher rate, with first-order output prediction, than the computation of the inertia matrix. Reasonable effective delays were achieved at much lower sampling frequencies leading to a high CPU availability.

The order of prediction was then increased and it was concluded that the error in the result values decreases noticeably. The higher the degree of prediction, the lower the error would be. However, higher orders of prediction would result in additional computational steps. Thus, there is a compromise between the order of prediction and computational complexity. For the inertia matrix, it was concluded that first-order prediction, which requires very minimal computation time, would be adequate.

Finally, the effects of acceleration and total travel time were investigated. Increasing the acceleration had no effect on the speedup. However, it exhibited a
much more pronounced impact on the effective delay and the differential torque error. The latter increased proportionally as acceleration increased since higher acceleration results in higher rate of change of joint torques. The error induced in the non-strict computation of joint torques due to higher acceleration adversely effected the effective delay. That is, a lower bound on the effective delay is obtained at frequencies higher than the case of lower acceleration. Thus, CPU availability diminishes as acceleration increases. On the other hand, decreasing the total traveling time and acceleration would result in a higher CPU availability.

Various algorithmic and architectural approaches have been introduced in this dissertation to achieve real-time computation of the inertia matrix. However, much work remains to be done. In the remainder of this chapter, possible related areas of future research will be discussed.

6.2 Suggestions for Future Research

The algorithms and architectures presented here merely establish the groundwork, the foundation needed to lower the computational intensity of dynamic control schemes needed for real-time control. Much work remains to be done to fully realize an effective approach toward limiting the computational bottleneck that exists in implementing sophisticated control schemes for current and the next generation of robotic mechanisms performing complex tasks.

With regard to Chapter III, various other parallel architectures with different interconnection schemes need to be examined. A CAD tool is essential in making fair and quick comparisons among various configurations. The underlying comparison criteria should be based upon a dynamic cost function which varies from application to application.
Also, further work toward efficient mapping the $O(\log_2 N)$ parallel algorithm into parallel architectures is essential. Processor interconnection schemes and communication (I/O) overhead are among the issues that have to be considered. The basic objective is to minimize the computational latency while maximizing CPU utilization and reducing cost.

In general, as far as the algorithmic aspects of the inertia matrix are concerned, an intrinsic, coordinate free, decomposition of the algorithm with no regard to architecture is instrumental in depicting maximum inherent parallelism within the algorithm. Then, various architectures and other extrinsic issues which are architecture dependent such as coordinate frame assignment and projection should be proposed to efficiently implement the algorithms. It is believed that such a strategy toward algorithm and architecture design, in general, is an effective approach to attain realizable solutions when computational issues are of concern.

In Chapter V, the concept of Non-Strict Computation of the manipulator inertia matrix was introduced. The work presented there is just the foundation for extended research work that may follow. In general, the concept of Non-Strict Computation appears to be so versatile that it may be effectively applied to a wide range of scientific problems. Application of the Non-Strict Computational Approach in other areas should be investigated. Then, once the ground work is laid out, a general formulation may follow.

Prediction could be further extended to not only the output variables, but also to the input variables such as for the acceleration when computing the torques based on the inertia matrix.

Specifically, the following research problems need to be conducted. Scheduling of processes need to be performed either statically or dynamically depending on the problem. For example, computation of Inverse Dynamics using the dynamic equa-
tions of motion, which requires explicit computation of the inertia matrix, requires dynamic scheduling. Thus, compilers and schedulers are needed to fully detect parallelism once the hardware resources are specified. Computation of Inverse Dynamics directly using the dynamic equations of motion should be conducted using Non-Strict Computational Models in conjunction with multiple rate computation. This approach needs to be compared with the more traditional but efficient approaches such as the Newton-Euler and Lagrange Euler formulations.

Non-Strict Computational Models based upon other techniques besides prediction need to be studied. For example, the inertia matrix is defined by a six-dimensional vector space which is fully defined by six linearly-independent eigenvectors. It is expected that only a few of such vectors change very often. If this is so, then only the few dominating eigenvectors need to be computed depending on the robot position and orientation in space. This approach also requires dynamic scheduling, but perhaps static scheduling is sufficient if the robot trajectory is well known in advance.

The algorithms and architectures presented in this dissertation constitute a comprehensive solution to the computation of the manipulator inertia matrix in real time. Also, this work has laid the foundation for further investigation of the Non-Strict Computational Model which may be applied to a wide range of scientific applications. The results obtained and the research extensions suggested should lead to greater improvement in efficiently computing the inertia matrix, in specific, and a wide class of computationally intensive problems, in general.
APPENDIX A

COORDINATE ASSIGNMENT AND LINK PARAMETERS

The following two sections lay out the coordinate frames assignment for a
general linkage mechanism and then specifically apply it to the PUMA 560 robot
manipulator. The notation is followed throughout the dissertation and also used
within the simulator. The kinematic and dynamic parameters, used in the simu­
lator for PUMA 560, are then tabulated.

A.1 Notation and Coordinate Frames

The coordinate frames used are derived from the Denavit-Hartenberg nota­
tion [74], but differ in a few details. For this reason, we refer to it as modified
Denavit-Hartenberg notation. Below is the definition of the four modified Denavit-
Hartenberg link parameters and Fig. 22 clearly depicts the location of the i and
i+1 coordinate frames.

\[ a_{i+1} \triangleq \text{the distance from } z_i \text{ to } z_{i+1} \text{ measured along } x_i, \]
\[ \alpha_{i+1} \triangleq \text{the angle between } z_i \text{ and } z_{i+1} \text{ measured about } x_i, \]
\[ d_{i+1} \triangleq \text{the distance from } x_i \text{ to } x_{i+1} \text{ measured along } z_{i+1}, \text{ and} \]
\[ \theta_{i+1} \triangleq \text{the angle between } x_i \text{ and } x_{i+1} \text{ measured about } z_{i+1}. \]

A complete treatment of link transformation is given in [75,76] and the re­
sulting homogeneous transformation based upon the modified Denavit-Hartenberg
Figure 22: Link Frames and Parameter Assignment.

Figure 23: PUMA 560 in Zero Position.
parameters is given below,

\[ iT_{i+1} = \begin{bmatrix} c\theta_{i+1} & -s\theta_{i+1} & 0 & a_{i+1} \\ s\theta_{i+1}c\alpha_{i+1} & c\theta_{i+1}c\alpha_{i+1} & -s\alpha_{i+1} & -s\alpha_{i+1}d_{i+1} \\ s\theta_{i+1}s\alpha_{i+1} & c\theta_{i+1}s\alpha_{i+1} & c\alpha_{i+1} & c\alpha_{i+1}d_{i+1} \\ 0 & 0 & 0 & 1 \end{bmatrix} \]  

(A.1)

where \( c\theta_{i+1} \) and \( s\theta_{i+1} \) denote \( \cos \theta_{i+1} \) and \( \sin \theta_{i+1} \), respectively. The same definition applies to \( c\alpha_{i+1} \) and \( s\alpha_{i+1} \). This homogeneous transformation is made up of a \( 3 \times 3 \) rotational transformation signified by \( iU_{i+1} \) and a \( 3 \times 1 \) translational transformation indicated by \( ip_{i+1} \). In matrix form, it may be represented as

\[ iT_{i+1} = \begin{bmatrix} iU_{i+1} & ip_{i+1} \\ 0 & 1 \end{bmatrix}. \]  

(A.2)

The vector \( ip_{i+1} \) is also shown in Fig. 22.

A.2 PUMA 560 link parameters

The PUMA 560 in the zero position with attached coordinate frames following the Fig. 22 definition is shown in Fig. 23. The zero position is the configuration where all the \( x \)-axes are lined up. All of the four modified Denavit-Hartenberg parameters for each link are tabulated in Table 9. The position of the center of gravity for each link is listed in Table 10. Each link's dynamic parameters are given in Table 11. Finally, joint angle and maximum joint torque limits are given in Table 12. All numerical data was extracted from [73].
Table 9: Modified Denavit-Hartenberg Parameters for PUMA 560

<table>
<thead>
<tr>
<th>Link</th>
<th>$\alpha_i$ (degrees)</th>
<th>$\theta_i$ (degrees)</th>
<th>$\dot{\theta}_i$ (meters)</th>
<th>$d_i$ (meters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>$\theta_1$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>-90</td>
<td>$\theta_2$</td>
<td>0</td>
<td>0.2435</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>$\theta_3$</td>
<td>0.4318</td>
<td>-0.0934</td>
</tr>
<tr>
<td>4</td>
<td>90</td>
<td>$\theta_4$</td>
<td>-0.0203</td>
<td>0.4331</td>
</tr>
<tr>
<td>5</td>
<td>-90</td>
<td>$\theta_5$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>90</td>
<td>$\theta_6$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 10: Link Centers of Gravity for PUMA 560

<table>
<thead>
<tr>
<th>Link</th>
<th>$r_x$ (m)</th>
<th>$r_y$ (m)</th>
<th>$r_z$ (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.068</td>
<td>0.006</td>
<td>-0.016</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>-0.070</td>
<td>0.014</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>-0.019</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0.032</td>
</tr>
</tbody>
</table>
Table 11: Dynamic Parameters for PUMA 560

<table>
<thead>
<tr>
<th>Link</th>
<th>mass (kg)</th>
<th>$I_{xx}(kg \cdot m^2)$</th>
<th>$I_{yy}(kg \cdot m^2)$</th>
<th>$I_{zz}(kg \cdot m^2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.35</td>
</tr>
<tr>
<td>2</td>
<td>17.40</td>
<td>0.130</td>
<td>0.524</td>
<td>0.539</td>
</tr>
<tr>
<td>3</td>
<td>4.80</td>
<td>0.066</td>
<td>0.0125</td>
<td>0.086</td>
</tr>
<tr>
<td>4</td>
<td>0.82</td>
<td>$1.80 \times 10^{-3}$</td>
<td>$1.80 \times 10^{-3}$</td>
<td>$1.30 \times 10^{-3}$</td>
</tr>
<tr>
<td>5</td>
<td>0.34</td>
<td>$0.3 \times 10^{-3}$</td>
<td>$0.3 \times 10^{-3}$</td>
<td>$0.4 \times 10^{-3}$</td>
</tr>
<tr>
<td>6</td>
<td>0.09</td>
<td>$0.15 \times 10^{-3}$</td>
<td>$0.15 \times 10^{-3}$</td>
<td>$0.04 \times 10^{-3}$</td>
</tr>
</tbody>
</table>

Table 12: Joint Angle Limits and Maximum Joint Torques for PUMA 560

<table>
<thead>
<tr>
<th>Joint</th>
<th>Joint Limit (degrees)</th>
<th>Max. Torque (N-m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$160^\circ \geq \theta_1 \geq -160^\circ$</td>
<td>97.6</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>186.4</td>
</tr>
<tr>
<td>3</td>
<td>$225^\circ \geq \theta_3 \geq -45^\circ$</td>
<td>89.4</td>
</tr>
<tr>
<td>4</td>
<td>$170^\circ \geq \theta_4 \geq -170^\circ$</td>
<td>24.2</td>
</tr>
<tr>
<td>5</td>
<td>$135^\circ \geq \theta_5 \geq -135^\circ$</td>
<td>20.1</td>
</tr>
<tr>
<td>6</td>
<td>$170^\circ \geq \theta_6 \geq -170^\circ$</td>
<td>21.3</td>
</tr>
</tbody>
</table>
The following is a brief description of the simulator written to conduct both performance and error analysis on the Strict and Non-Strict Computation of the inertia matrix. The effective computational delay is computed as a measure of algorithm performance by minimizing standard deviation as described in Chapter V. The simulator is written in Pascal on a SUN workstation. The PUMA 560 robot is the manipulator used within the simulator. A high-level description of the simulator is given in Fig 24 in the form of a flow chart.

The PUMA is first initialized to an initial position and orientation. The zero-position, where all x-axes of the local coordinates are lined up, is used as the initial robot position. The operator or the motion planner then specifies the desired goal position and orientation in the Cartesian coordinate system, the total traveling time, and the sampling frequency. The simulator then solves for the joint torques by applying the acceleration-dependent term of the general dynamic equation of motion given in Eq. (1.1). That is, the effects of the inertia matrix in computing the joint torques is of interest here.

An inverse kinematics routine is called to convert a goal position and orientation specified in Cartesian coordinates to its corresponding joint coordinates. Given the initial and final position of the manipulator, a trajectory generation routine is written to solve for position, velocity, and acceleration profiles. The
Compute Performance Parameters

Compute Joint Torques

Compute the Manipulator Inertia Matrix

Compute Position, Velocity, and Acceleration Profiles (Trajectory Generation)

Convert Cartesian Coordinates to Joint Coordinates (Inverse Kinematics)

Specify the Next Manipulator Position & Orientation, Total Travel Time, and Sampling Time

Initialize Manipulator Position & Orientation

Figure 24: Flow Chart of the Simulator.
routine is based on a fifth-order polynomial imposing constraints on initial and final position, velocity, and acceleration parameters. Two constraints are imposed by position boundaries:

\[
\begin{align*}
\theta(t_0) &= \theta_0, \\
\theta(t_f) &= \theta_f,
\end{align*}
\] (B.1)

where \(\theta(t_0)\) is the initial position and \(\theta(t_f)\) is the final position of the joints. Two additional constraints are imposed by meeting the continuity of velocity at the initial and final time:

\[
\begin{align*}
\dot{\theta}(t_0) &= 0, \\
\dot{\theta}(t_f) &= 0.
\end{align*}
\] (B.2)

Finally, two more boundary conditions are imposed by continuity of acceleration. That is,

\[
\begin{align*}
\ddot{\theta}(t_0) &= 0, \\
\ddot{\theta}(t_f) &= 0.
\end{align*}
\] (B.3)

Of course, to meet the above six constraints, a polynomial of at least fifth degree is required. Thus, the following fifth-order polynomial is employed within the trajectory generator.

\[
\theta(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5. \] (B.4)

Differentiating the above polynomial twice results in the joint velocity and acceleration along this path.

\[
\begin{align*}
\dot{\theta}(t) &= a_1 + 2a_2 t + 3a_3 t^2 + 4a_4 t^3 + 4a_5 t^4, \\
\ddot{\theta}(t) &= 2a_2 + 6a_3 t + 12a_4 t^2 + 20a_5 t^3.
\end{align*}
\] (B.5)
Combining Eqs. (B.1), (B.2), and (B.3) with Eqs. (B.4) and (B.5) yields six equations in six unknowns. Solving these equations for the six unknown polynomial coefficients results in,

\begin{align*}
a_0 &= \theta_0, \\
a_1 &= 0, \\
a_2 &= 0, \\
a_3 &= \frac{10}{t_f^3}(\theta_f - \theta_0), \\
a_4 &= -\frac{15}{t_f^2}(\theta_f - \theta_0), \\
a_5 &= \frac{6}{t_f^2}(\theta_f - \theta_0).
\end{align*}

More elaborate path planning such as fifth-order polynomials with via points or even higher order polynomials may be devised; however, for the purposes of the study undertaken in Chapter V, a fifth-order polynomial ensuring smooth and continuous velocity and acceleration suffices. The following two sections contain the two part program listings of the simulator.

B.1 Program Listing of the Simulator, Part 1: Computation of Joint Torques

This section contains the program listings for the first part of the simulator. Mainly, it computes the joint torques based upon the acceleration-dependent term of the equations of motion. It simulates a multiprocessor architecture in which one processor is dedicated to a joint. Both Strict and Non-Strict Computation of the inertia matrix may be initiated.
program invdyn(input, output, GoalPosDat,
     path1, path2, path3, path4, path5, path6,
     TorqueDat1, TorqueDat2, TorqueDat3,
     TorqueDat4, TorqueDat5, TorqueDat6,
     PredTorqueDat1, PredTorqueDat2, PredTorqueDat3,
     PredTorqueDat4, PredTorqueDat5, PredTorqueDat6,
     StrictTorqueDat1, StrictTorqueDat2,
     StrictTorqueDat3, StrictTorqueDat4,
     StrictTorqueDat5, StrictTorqueDat6);

{*************************************************************}
{*************************************************************}
{*************************************************************}
{*************************************************************}
{*************************************************************}

#include "const.h"
#include "vars.h"
#include "invdyn.h"
begin
  rewrite(path1); rewrite(path2); rewrite(path3);
  rewrite(path4); rewrite(path5); rewrite(path6);
  rewrite(TorqueDat1); rewrite(TorqueDat2);
  rewrite(TorqueDat3) ; rewrite(TorqueDat4);
  rewrite(TorqueDat5); rewrite(TorqueDat6);
  rewrite(PredTorqueDat1); rewrite(PredTorqueDat2);
  rewrite(PredTorqueDat3); rewrite(PredTorqueDat4);
  rewrite(PredTorqueDat5); rewrite(PredTorqueDat6);
  rewrite(StrictTorqueDat1); rewrite(StrictTorqueDat2);
  rewrite(StrictTorqueDat3); rewrite(StrictTorqueDat4);
  rewrite(StrictTorqueDat5); rewrite(StrictTorqueDat6);
  reset(GoalPosDat,'GoalPosDat');

  writeln;
  write('Enter Total Time (in seconds): '); readln(FinalTime);

  write('Enter Sampling Time (in seconds): '); readln(DeltaTime);

  write('Strict or Non-Strict Computation? [1 or 2]: '); readln(Sflag);
  if Sflag = 2 then begin
    write('Zero-order or First-Order Prediction? [1 or 2]: '); readln(PredictionFlag);
    end;
  writeln;
  init;

  [* Read the final goal position from the GoalPosDat file *]
  for i := 1 to 3 do begin
    readln(GoalPosDat,T[i,1],T[i,2],T[i,3],T[i,4]);
  end;

  invkin;

  for i := 1 to N do FinalTheta[i] := theta[i];

  trajectory;

  writeln;
end.
(* This routine initializes the robot initial position *)
(* kinematic and dynamic linkage parameters *)

procedure init;
begin

  Mass[7] := 0.0;

  U[7,1,1] := 0.0; U[7,1,2] := 0.0; U[7,1,3] := 0.0;
  U[7,2,1] := 0.0; U[7,2,2] := 0.0; U[7,2,3] := 0.0;
  U[7,3,1] := 0.0; U[7,3,2] := 0.0; U[7,3,3] := 0.0;

c[7,1] := 0.0; c[7,2] := 0.0; c[7,3] := 0.0;

  p[1,1] := p11; p[1,2] := p12; p[1,3] := p13;

  s[1,1] := s11; s[1,2] := s12; s[1,3] := s13;
  s[3,1] := s31; s[3,2] := s32; s[3,3] := s33;
  s[4,1] := s41; s[4,2] := s42; s[4,3] := s43;
  s[5,1] := s51; s[5,2] := s52; s[5,3] := s53;
  s[6,1] := s61; s[6,2] := s62; s[6,3] := s63;

  E[7,1,1] := 0.0; E[7,1,2] := 0.0; E[7,1,3] := 0.0;
  E[7,2,1] := 0.0; E[7,2,2] := 0.0; E[7,2,3] := 0.0;
  E[7,3,1] := 0.0; E[7,3,2] := 0.0; E[7,3,3] := 0.0;

  II[1,1,1] := II111; II[1,1,2] := II112; II[1,1,3] := II113;
  II[1,2,1] := II121; II[1,2,2] := II122; II[1,2,3] := II123;
  II[1,3,1] := II131; II[1,3,2] := II132; II[1,3,3] := II133;
  II[2,1,1] := II211; II[2,1,2] := II212; II[2,1,3] := II213;
  II[3,1,1] := II311; II[3,1,2] := II312; II[3,1,3] := II313;
  II[4,1,1] := II411; II[4,1,2] := II412; II[4,1,3] := II413;
II[5,1,1] := I5I11; II[5,1,2] := I5I12; II[5,1,3] := I5I13;
II[6,1,1] := I6I11; II[6,1,2] := I6I12; II[6,1,3] := I6I13;

MaxTorque[1] := MaxTaul;
MaxTorque[2] := Max Tau2;
MaxTorque[3] := Max Tau3;
MaxTorque[4] := Max Tau4;
MaxTorque[5] := Max Tau5;
MaxTorque[6] := Max Tau6;

for l := 1 to N do
begin
  for i := 1 to N+1 do
  begin
    for j := 1 to 3 do
    begin
      Nc[l,i,j] := 0.0;
    end;
    for k := 1 to 3 do
    begin
      NE[l,i,j,k] := 0.0;
    end;
  end;
end;

(* Initial position of the robot *)
for i := 1 to N do
begin
  theta[i] := 0.0;
  InitialTheta[i] := theta[i];
end;

end;
#include "const.h"
#include "vars.h"
#include "invdyn.h"

{*******************************************************
[* File: invkin.p *
[* Programmer: M. Amin-Javaheri *
[* Date: Jan. 10, 1989 *
[* Program: Inverse Kinematics, Computes the PUMA560 *
[* joint angles given a desired position in *
* the Cartesian coordinate system. *
[* *
[* Input *
[* Parameters: T (4X4): transformation matrix and initial *
* values of joint angles *
[* Procedures *
[* Called: arctan2 -- computes ArcTan2(x) *
[* *
{*******************************************************

procedure invkin;

function arctan2;
begin
  if Y >= 0.0 then
    SIGN := 1.0
  else
    SIGN := -1.0;

  if abs(X) > 0.00001 then { X IS NON-ZERO }
    if X > 0.0 then
      arctan2 := arctan(Y / X)
    else
      arctan2 := arctan(Y / X) + PI * SIGN
    { END IF X }
  else
    arctan2 := PI / 2.0 * SIGN;
end;

begin
  [* Compute Joint Angle 1 *]
  temp[1,1] := d2 + d3;
  temp[2,2] := sqrt(sqr(T[1,4]) + sqr(T[2,4]) -sqr(temp[1,1]));
  temp1[1,1] := arctan2(T[2,4], T[1,4]) -
    arctan2(temp[1,1], temp[2,2]);
  temp1[2,2] := arctan2(T[2,4], T[1,4]) -
    arctan2(temp[1,1], -temp[2,2]);
if ((temp1[1,1] >= MinLimit1) & (temp1[1,1] <= MaxLimit1)) then 
valid[1] := true;
if ((temp1[2,2] >= MinLimit1) & (temp1[2,2] <= MaxLimit1)) then 
valid[2] := true;
if ((valid[1] = true) & (valid[2] = true)) then 
begin
  diff[1] := abs(temp1[1,1] - InitialTheta[1]);

  if (diff[1] > diff[2]) then
    theta[1] := temp1[2,2]
  else
    theta[1] := temp1[1,1];
end
else
begin
  if (valid[1] = true) then
    theta[1] := temp1[1,1]
  else
    if (valid[2] = true) then
      theta[1] := temp1[2,2]
    else
      writeln('***** angle 1 in error *****');
end;

[* Compute Joint Angle 3 *]
valid[1] := false;

temp1[1,1] := (sqr(T[1,4]) + sqr(T[2,4]) + sqr(T[3,4]) - 
sqr(d2 + d3) - sqr(d4) - sqr(a3) - sqr(a4))/(2.0*a3);

if ((temp1[1,1] >= MinLimit3) & (temp1[1,1] <= MaxLimit3)) then 
valid[1] := true;
if ((temp1[2,2] >= MinLimit3) & (temp1[2,2] <= MaxLimit3)) then 
valid[2] := true;
if ((valid[1] = true) & (valid[2] = true)) then 
begin
  diff[1] := abs(temp1[1,1] - InitialTheta[3]);

  if (diff[1] > diff[2]) then
  else
    theta[3] := temp1[1,1];
end
else
    begin
      if (valid[1] = true) then
        theta[3] := temp[1,1]
      else
        if (valid[2] = true) then
        else
          writeln('***** angle 3 in error *****');
    end;

  {* Compute Joint Angle 2 *}
  temp[1,1] := -(a4 + a3*cos(theta[3]))*T[3,4] +
    (T[1,4]*cos(theta[1]) + T[2,4]*sin(theta[1]))*
    (a3*sin(theta[3]) + d4);
  temp[2,2] := (a3*sin(theta[3]) + d4)*T[3,4] +
    (T[1,4]*cos(theta[1]) + T[2,4]*sin(theta[1]))*
    (a3*cos(theta[3]) + a4);
  theta23 := arctan2(temp[1,1],temp[2,2]);

  {* Compute Joint Angle 4 *}
  temp[1,1] := -T[1,3]*sin(theta[1]) + T[2,3]*cos(theta[1]);
  temp[2,2] := T[1,3]*cos(theta[1])*cos(theta23) +
    T[2,3]*sin(theta[1])*cos(theta23) -
    T[3,3]*sin(theta23);
  temp[1,1] := arctan2(temp[1,1],temp[2,2]);
  temp[2,2] := PI + temp[1,1];
  if (temp[1,1] >= MinLimit4) & (temp[1,1] <= MaxLimit4) then
    valid[1] := true;
  if (temp[2,2] >= MinLimit4) & (temp[2,2] <= MaxLimit4) then
    valid[2] := true;
  if ((valid[1] = true) & (valid[2] = true)) then
    begin
      diff[1] := abs(temp[1,1] - InitialTheta[4]);
      if (diff[1] > diff[2]) then
      else
        theta[4] := temp[1,1];
    end
  else
    begin
      if (valid[1] = true) then
        theta[4] := temp[1,1]
      else
        if (valid[2] = true) then
        else
          writeln('***** angle 4 in error *****');
    end;
{* Compute Joint Angle 5 *}
valid[1] := false;

\[
tmp[1,1] := T[1,3]*(\cos(\theta[1]) \times \cos(\theta[23]) \times \\
\cos(\theta[4]) - \sin(\theta[1]) \times \sin(\theta[4])) + \\
T[2,3]*(\sin(\theta[1]) \times \cos(\theta[23]) \times \\
\cos(\theta[4]) + \cos(\theta[1]) \times \sin(\theta[4])) - \\
T[3,3]*\sin(\theta[23]) \times \cos(\theta[4]);
\]

\[
tmp[2,2] := T[1,3]*\cos(\theta[1]) \times \sin(\theta[23]) + \\
T[2,3]*\sin(\theta[1]) \times \sin(\theta[23]) + \\
T[3,3]*\cos(\theta[23]);
\]

\[
templ[1,1] := \arctan2(tmp[1,1],tmp[2,2]);
\]

\[
templ[2,2] := -templ[1,1];
\]

if ((templ[1,1] >= MinLimit5) & (templ[1,1] <= MaxLimit5)) then
valid[1] := true;
if ((templ[2,2] >= MinLimit5) & (templ[2,2] <= MaxLimit5)) then
valid[2] := true;
if ((valid[1] = true) & (valid[2] = true)) then
begin
  \[
  \text{diff}[1] := \text{abs}(\text{templ}[1,1] - \text{InitialTheta}[5]);
  \text{diff}[2] := \text{abs}(\text{templ}[2,2] - \text{InitialTheta}[5]);
  \text{if } (\text{diff}[1] > \text{diff}[2]) \text{ then}
    \theta[5] := \text{templ}[2,2];
  \text{else}
    \theta[5] := \text{templ}[1,1];
  \end
end
else
begin
  \text{if } \text{valid}[1] = \text{true} \text{ then}
    \theta[5] := \text{templ}[1,1];
  \text{else}
    \text{if } \text{valid}[2] = \text{true} \text{ then}
      \theta[5] := \text{templ}[2,2];
    \text{else}
      \text{writeln('***** angle 5 in error *****');}
\end

{* Compute Joint Angle 6 *}
valid[1] := false;

\[
tmp[1,1] := T[1,1]*(-\cos(\theta[1]) \times \cos(\theta[23]) \times \\
\sin(\theta[4]) - \sin(\theta[1]) \times \cos(\theta[4])) + \\
T[2,1]*(-\sin(\theta[1]) \times \cos(\theta[23]) \times \\
\sin(\theta[4]) + \cos(\theta[1]) \times \cos(\theta[4])) + \\
T[3,1]*\sin(\theta[23]) \times \sin(\theta[4]);
\]
temp[2,2] := T[1,1]*((cos(theta[1])*cos(theta23)*
    cos(theta[4]) - sin(theta[1])*sin(theta[4]))*
    cos(theta[5]) - cos(theta[1])*sin(theta23)*
    sin(theta[5])) +
T[2,1]*((sin(theta[1])*cos(theta23)*
    cos(theta[4]) + cos(theta[1])*sin(theta[4]))*
    cos(theta[5]) - sin(theta[1])*sin(theta23)*
    sin(theta[5])) +
T[3,1]*(-sin(theta23)*cos(theta[4])* 
    cos(theta[5]) - cos(theta23)*sin(theta[5]));

temp[1,1] := arctan2(temp[1,1],temp[2,2]);

templ[1,2] := PI + templ[1,1];

if ((templ[1,1] > MinLimit6) & (templ[1,1] <= MaxLimit6)) then
  valid[1] := true;

  valid[2] := true;

if ((valid[1] = true) & (valid[2] = true)) then
  begin
    diff[1] := abs(templ[1,1] - InitialTheta[6]);
    if (diff[1] > diff[2]) then
    else
      theta[6] := templ[1,1];
  end;
else
  begin
    if (valid[1] = true) then
      theta[6] := templ[1,1]
    else
      if (valid[2] = true) then
      else
        writeln('***** angle 6 in error *****');
  end;
end;
procedure trajectory;
begin
for i := 1 to N do
begin
  InitialTheta[i] := InitialTheta[i]*180.0/PI;
  FinalTheta[i] := FinalTheta[i]*180.0/PI;
  a[i,0] := InitialTheta[i];
  a[i,1] := 0.0;
  a[i,2] := 0.0;
  a[i,3] := 10.0*(FinalTheta[i] - InitialTheta[i])/
            (FinalTime*sqr(FinalTime));
  a[i,4] := -15.0*(FinalTheta[i] - InitialTheta[i])/
             sqr(sqr(FinalTime));
  a[i,5] := 6.0*(FinalTheta[i] - InitialTheta[i])/
             (FinalTime*sqr(sqr(FinalTime)));
end;
Ttime := 0.0;
tt := 1;
while (Ttime <= FinalTime+0.0001) do
begin
for i := 1 to N do
begin
  position[i,tt] := a[i,0] + a[i,1]*Ttime +
    a[i,2]*sqr(Ttime) + a[i,3]*sqr(Ttime)*Ttime +
    a[i,4]*sqr(sqr(Ttime)) +
    a[i,5]*sqr(sqr(Ttime))*Ttime;
end;
end;
velocity[\textit{i,tt}] := a[\textit{i,1}] + 2.0*a[\textit{i,2}]*Ttime +
3.0*a[\textit{i,3}]*\text{sqr}(Ttime) +
4.0*a[\textit{i,4}]*\text{sqr}(Ttime)*Ttime +
5.0*a[\textit{i,5}]*\text{sqr}(\text{sqr}(Ttime));

acceleration[\textit{i,tt}] := 2.0*a[\textit{i,2}] + 6.0*a[\textit{i,3}]*Ttime +
12.0*a[\textit{i,4}]*\text{sqr}(Ttime) +
20.0*a[\textit{i,5}]*\text{sqr}(Ttime)*Ttime;

\text{theta[\textit{i}]} := \text{position[\textit{i,tt}]}*\text{PI}/180.0;
end;

\text{writeln(path1,Ttime:8:5,' ',position[1,tt]:9:5,' ',}
\text{velocity[1,tt]:9:5,' ',acceleration[1,tt]:9:5);}

\text{writeln(path2,Ttime:8:5,' ',position[2,tt]:9:5,' ',}
\text{velocity[2,tt]:9:5,' ',acceleration[2,tt]:9:5);}

\text{writeln(path3,Ttime:8:5,' ',position[3,tt]:9:5,' ',}
\text{velocity[3,tt]:9:5,' ',acceleration[3,tt]:9:5);}

\text{writeln(path4,Ttime:8:5,' ',position[4,tt]:9:5,' ',}
\text{velocity[4,tt]:9:5,' ',acceleration[4,tt]:9:5);}

\text{writeln(path5,Ttime:8:5,' ',position[5,tt]:9:5,' ',}
\text{velocity[5,tt]:9:5,' ',acceleration[5,tt]:9:5);}

\text{writeln(path6,Ttime:8:5,' ',position[6,tt]:9:5,' ',}
\text{velocity[6,tt]:9:5,' ',acceleration[6,tt]:9:5);}

\text{if Sflag = 1 then JointTorques}
\text{else Ninertia;}

\text{tt := tt + 1;}
\text{Ttime := Ttime + DeltaTime;}
end;
#include "const.h"
#include "vars.h"
#include "invdyn.h"

{** File: JointTorques.p **}
{** Programmer: M. Amin-Javaheri **}
{** Date: Jan. 10, 1989 **}

{** Program: Computes joint torques **}
{** Inputs: acceleration and the inertia matrix **}

{** Procedures **}
{** Called: inertia -- Computes the Inertia Matrix **}

procedure JointTorques;
begin

inertia;

for k := 1 to N do
begin
  torque[tt,k] := 0.0;
  for l := 1 to N do
    torque[tt,k] := torque[tt,k]+H[k,l]*acceleration[l,tt];

  case k of
    6 : begin
      write(TorqueDat6,Ttime:8:5,' ');
      writeln(TorqueDat6,torque[tt,k]:12:8,' ');
      write(StrictTorqueDat6,Ttime+StrictDelay:8:5,' ');
      writeln(StrictTorqueDat6,torque[tt,k]:12:8,' ');
      end;
    5 : begin
      write(TorqueDat5,Ttime:8:5,' ');
      writeln(TorqueDat5,torque[tt,k]:12:8,' ');
      write(StrictTorqueDat5,Ttime+StrictDelay:8:5,' ');
      writeln(StrictTorqueDat5,torque[tt,k]:12:8,' ');
      end;
    4 : begin
      write(TorqueDat4,Ttime:8:5,' ');
      writeln(TorqueDat4,torque[tt,k]:12:8,' ');
      write(StrictTorqueDat4,Ttime+StrictDelay:8:5,' ');
      writeln(StrictTorqueDat4,torque[tt,k]:12:8,' ');
      end;
   end;
end;
3: begin
    write(TorqueDat3, Ttime:8:5, ' ');
    writeln(TorqueDat3, torque[tt,k]:12:8, ' ');
    write(StrictTorqueDat3, Ttime+StrictDelay:8:5, ' ');
    writeln(StrictTorqueDat3, torque[tt,k]:12:8, ' ');
end;

2: begin
    write(TorqueDat2, Ttime:8:5, ' ');
    writeln(TorqueDat2, torque[tt,k]:12:8, ' ');
    write(StrictTorqueDat2, Ttime+StrictDelay:8:5, ' ');
    writeln(StrictTorqueDat2, torque[tt,k]:12:8, ' ');
end;

1: begin
    write(TorqueDat1, Ttime:8:5, ' ');
    writeln(TorqueDat1, torque[tt,k]:12:8, ' ');
    write(StrictTorqueDat1, Ttime+StrictDelay:8:5, ' ');
    writeln(StrictTorqueDat1, torque[tt,k]:12:8, ' ');
end;

    if torque[tt,k] > MaxTorque[k] then
        writeln('*** Joint Torque ',k:1:0, ' is out of range');
    end;

    if tt mod 70 <> 0 then write('.'
else writeln('.');

end.
```plaintext
#include "const.h"
#include "vars.h"
#include "invdyn.h"

{***************
{ File: inertia.p

{ Programmer: M. Amin-Jawaheri
{ Date: Jan. 10, 1989
{ Program: Inertia Matrix, Computes the symmetric
6X6 Inertia Matrix given a set of joint angles.
{ Input: joint angles

{***************

procedure inertia;
begin

{ Compute the U matrix *
for i := N-1 downto 0 do
begin
  U[i+1,1,1] := cos(theta[i+1]);
  U[i+1,1,2] := -sin(theta[i+1]);
  U[i+1,1,3] := 0.0;
  U[i+1,2,1] := sin(theta[i+1])*cos(alpha[i+1]*PI/180.0);
  U[i+1,2,2] := cos(theta[i+1])*cos(alpha[i+1]*PI/180.0);
  U[i+1,2,3] := -sin(alpha[i+1]*PI/180.0);
  U[i+1,3,1] := sin(theta[i+1])*sin(alpha[i+1]*PI/180.0);
  U[i+1,3,2] := cos(theta[i+1])*sin(alpha[i+1]*PI/180.0);
  U[i+1,3,3] := cos(alpha[i+1]*PI/180.0);
end;

for i := N downto 1 do
begin

{ Compute composite mass *
Mass[i] := Mass[i+1] + mass[i];

{ Compute composite center of mass *
for j := 1 to 3 do
begin
  temp[i,j] := 0.0;
  for k := 1 to 3 do
begin
    temp[i,j] := temp[i,j] + U[i+1,j,k]*c[i+1,k];
  end;
  c[i,j] := (1/Mass[i])*mass[i]*s[i,j] +
            Mass[i+1]*(temp[i,j] + p[i+1,j]);
end;
```
/* Compute composite inertia matrix for rigid bodies */
/* i through N, Initialization */
for j := 1 to 3 do
begin
  temp[i,j] := 0.0;
  for k := 1 to 3 do
    begin
      mat5[i,j,k] := 0.0;
      mat6[i,j,k] := 0.0;
    end;
end;

/* Computation of U*c, (U*c + p - c), and (s - c) */
for j := 1 to 3 do
begin
  for k := 1 to 3 do
    begin
      temp[i,j] := temp[i,j] + U[i+1,j,k]*c[i+1,k];
    end;
  temp[i,j] := temp[i,j] + p[i+1,j] - c[i,j];
end;

/* Formation of dot product -- result is an scalar */
t := sqr(temp[i,1]) + sqr(temp[i,2]) + sqr(temp[i,3]);
t1 := sqr(temp[i,1]) + sqr(temp[i,2]) + sqr(temp[i,3]);

/* Multiplication of t and t1 by a 3X3 identity matrix */
for j := 1 to 3 do
begin
  for k := 1 to 3 do
    begin
      if (j<k) then
        begin
          mat1[i,j,k] := 0.0;
          mat2[i,j,k] := 0.0;
        end;
      else
        begin
          mat1[i,j,k] := t;
          mat2[i,j,k] := t1;
        end;
    end;
end;

/* Computation of V times V transpose */
for j := 1 to 3 do
begin
  for k := 1 to 3 do
    begin
      mat3[i,j,k] := (temp[i,j]*temp[i,k]);
      mat4[i,j,k] := (temp[i,j]*temp[i,k]);
    end;
end;
/* Computation of U*E*U */
for l := 1 to 3 do
begin
  for j := 1 to 3 do
  begin
    for k := 1 to 3 do
    begin
      mat5[i,l,j] := mat5[i,l,j]+U[i+1,l,k]*E[i+1,k,j];
    end;
  end;
end;

for l := 1 to 3 do
begin
  for j := 1 to 3 do
  begin
    for k := 1 to 3 do
    begin
      mat6[i,l,j] := mat6[i,l,j] + mat5[i,l,k]*U[i+1,j,k];
    end;
  end;
end;

/* Compute the inertia matrix */
for j := 1 to 3 do
begin
  for k := 1 to 3 do
  begin
    E[i,j,k] := mat6[i,j,k] + II[i,j,k] + Mass[i+1]*(mat1[i,j,k] - mat3[i,j,k]) + mass[i]*(mat2[i,j,k] - mat4[i,j,k]);
  end;
end;

/* Compute joint ith force */
f[i,1] := -Mass[i]*c[i,2];
f[i,2] := Mass[i]*c[i,1];
f[i,3] := 0.0;

/* Compute joint ith moment */
n[i,1] := E[i,1,3] + (c[i,2]*f[i,3] - c[i,3]*f[i,2]);
n[i,2] := E[i,2,3] + (c[i,3]*f[i,1] - c[i,1]*f[i,3]);
n[i,3] := E[i,3,3] + (c[i,1]*f[i,2] - c[i,2]*f[i,1]);

/* Form (i,i)th element of the inertia matrix */
H[i,i] := n[i,3];
{* Newton-Euler backward recursion *}
for ii := i-1 downto 1 do
begin

{* Compute U*f *}
for j := 1 to 3 do
begin
  temp[ii,j] := 0.0;
  for k := 1 to 3 do
  begin
    temp[ii,j] := temp[ii,j] + U[ii+1,j,k]*f[ii+1,k];
  end;
  f[ii,j] := temp[ii,j];
end;

{* Compute moments *}
temp[ii,1] := p[ii+1,2]*f[ii+1,3] - p[ii+1,3]*f[ii+1,2] + n[ii+1,1];
temp[ii,2] := p[ii+1,3]*f[ii+1,1] - p[ii+1,1]*f[ii+1,3] + n[ii+1,2];
temp[ii,3] := p[ii+1,1]*f[ii+1,2] - p[ii+1,2]*f[ii+1,1] + n[ii+1,3];
for j := 1 to 3 do
begin
  templ[ii,j] := 0.0;
  for k := 1 to 3 do
  begin
    templ[ii,j] := templ[ii,j] + U[ii+1,j,k]*temp[ii,k];
  end;
  n[ii,j] := templ[ii,j];
end;
H[ii,i] := n[ii,3];
H[i,ii] := H[ii,i];
end;
end;
# include "const.h"
#include "vars.h"
#include "invdyn.h"

{*******************************************************************************}

{*******************************************************************************}

procedure Ninertia;
begin

(* Compute the U matrix *)
for i := N downto 0 do
begin
  U[i+1,1,1] := cos(theta[i+1]);
  U[i+1,1,2] := -sin(theta[i+1]);
  U[i+1,1,3] := 0.0;
  U[i+1,2,1] := sin(theta[i+1]) * cos(alpha[i+1] * PI/180.0);
  U[i+1,2,2] := cos(theta[i+1]) * cos(alpha[i+1] * PI/180.0);
  U[i+1,2,3] := -sin(alpha[i+1] * PI/180.0);
  U[i+1,3,1] := sin(theta[i+1]) * sin(alpha[i+1] * PI/180.0);
  U[i+1,3,2] := cos(theta[i+1]) * sin(alpha[i+1] * PI/180.0);
  U[i+1,3,3] := cos(alpha[i+1] * PI/180.0);
end;

(* Compute composite mass *)
for i := N downto 1 do
begin
  Mass[i] := Mass[i+1] + mass[i];
end;
if tt >= 6 then
  begin
    proc(6);
    proc(5);
    proc(4);
    proc(3);
    proc(2);
    proc(1);
  end
else
  begin
    if tt = 1 then
      proc(6)
    else
      begin
        if tt = 2 then
          begin
            proc(6);
            proc(5);
          end
        else
          begin
            if tt = 3 then
              begin
                proc(6);
                proc(5);
                proc(4);
              end
            else
              begin
                if tt = 4 then
                  begin
                    proc(6);
                    proc(5);
                    proc(4);
                    proc(3);
                  end
                else
                  begin
                    proc(6);
                    proc(5);
                    proc(4);
                    proc(3);
                    proc(2);
                  end;
              end;
          end;
    end;
  end;
if tt mod 70 <> 0 then write(’.’)
else writeln(’.’);
procedure proc;
begin

(* Compute joint ith force *)
if PredictionFlag = 1 then
begin
  Nf[tt,i,1] := -Mass[i]*Nc[tt-1,i,2];
  Nf[tt,i,2] := Mass[i]*Nc[tt-1,i,1];
  Nf[tt,i,3] := 0.0;
end
else
begin
  Nf[tt,i,1] := -Mass[i]*(2.0*Nc[tt-1,i,2]-Nc[tt-2,i,2]);
  Nf[tt,i,2] := Mass[i]*(2.0*Nc[tt-1,i,1]-Nc[tt-2,i,1]);
  Nf[tt,i,3] := 0.0;
end;

(* Compute joint ith moment *)
if PredictionFlag = 1 then
begin
  Nn[tt,i,1] := NE[tt-1,i,1,3]+(Nc[tt-1,i,2]*Nf[tt,i,3]-
    Nc[tt-1,i,3]*Nf[tt,i,2]);

  Nn[tt,i,2] := NE[tt-1,i,2,3]+(Nc[tt-1,i,3]*Nf[tt,i,1]-
    Nc[tt-1,i,1]*Nf[tt,i,3]);

  Nn[tt,i,3] := NE[tt-1,i,3,3]+(Nc[tt-1,i,1]*Nf[tt,i,2]-
    Nc[tt-1,i,2]*Nf[tt,i,1]);
end
else
    begin
        Nn[i,t,1] := 2.0*NE[t-1,i,1,3] - NE[t-2,i,1,3] +
            ((2.0*Nc[t-1,i,2] - Nc[t-2,i,2])*Nf[t,i,3] -
            (2.0*Nc[t-1,i,3] - Nc[t-2,i,3])*Nf[t,i,2]);
        Nn[i,t,2] := 2.0*NE[t-1,i,2,3] - NE[t-2,i,2,3] +
            ((2.0*Nc[t-1,i,3] - Nc[t-2,i,3])*Nf[t,i,1] -
            (2.0*Nc[t-1,i,1] - Nc[t-2,i,1])*Nf[t,i,3]);
        Nn[i,t,3] := 2.0*NE[t-1,i,3,3] - NE[t-2,i,3,3] +
            ((2.0*Nc[t-1,i,1] - Nc[t-2,i,1])*Nf[t,i,2] -
            (2.0*Nc[t-1,i,2] - Nc[t-2,i,2])*Nf[t,i,1]);
    end;

    (* Form (i,i)th element of the inertia matrix *)
    H[i,i] := Nn[t,i,3];

    (* Newton-Euler backward recursion *)
    for ii := i-1 downto 1 do
        begin
            (* Compute U*f *)
            for j := 1 to 3 do
                begin
                    temp[ii,j] := 0.0;
                    for k := 1 to 3 do
                        begin
                            temp[ii,j] := temp[ii,j] + U[ii+1,j,k]*Nf[t,i+1,k];
                        end;
                    Nf[t,ii,j] := temp[ii,j];
                end;

            (* Compute moments *)
            temp[ii,1] := pi[i+1,3]*Nf[t,ii+1,3] -
                pi[i+1,1]*Nf[t,ii+1,3] + Nn[t,ii+1,3];
            temp[ii,2] := pi[i+1,3]*Nf[t,ii+1,1] -
                pi[i+1,1]*Nf[t,ii+1,3] + Nn[t,ii+1,2];
            temp[ii,3] := pi[i+1,1]*Nf[t,ii+1,2] -
                pi[i+1,2]*Nf[t,ii+1,3] + Nn[t,ii+1,1];
            for j := 1 to 3 do
                begin
                    temp[ii,j] := 0.0;
                    for k := 1 to 3 do
                        begin
                            temp[ii,j] := temp[ii,j] + U[ii+1,j,k]*temp[ii,k];
                        end;
                    Nn[t,ii,j] := temp[ii,j];
                end;
            H[ii,i] := Nn[t,ii,3];
            H[i,ii] := H[ii,i];
        end;
(* Compute joint torques *)
PredTorque[tt,i] := 0.0;
for i := 1 to N do
begin
    PredTorque[tt,i] := PredTorque[tt,i] +
        H[i,1]*acceleration[1,tt];
end;
case i of
    6 : begin
        write(PredTorqueDat6,Ttime+delay:11:8," ");
        writeln(PredTorqueDat6,PredTorque[tt,i]:12:8," ");
    end;
    5 : begin
        write(PredTorqueDat5,Ttime+delay:11:8," ");
        writeln(PredTorqueDat5,PredTorque[tt,i]:12:8," ");
    end;
    4 : begin
        write(PredTorqueDat4,Ttime+delay:11:8," ");
        writeln(PredTorqueDat4,PredTorque[tt,i]:12:8," ");
    end;
    3 : begin
        write(PredTorqueDat3,Ttime+delay:11:8," ");
        writeln(PredTorqueDat3,PredTorque[tt,i]:12:8," ");
    end;
    2 : begin
        write(PredTorqueDat2,Ttime+delay:11:8," ");
        writeln(PredTorqueDat2,PredTorque[tt,i]:12:8," ");
    end;
    1 : begin
        write(PredTorqueDat1,Ttime+delay:11:8," ");
        writeln(PredTorqueDat1,PredTorque[tt,i]:12:8," ");
    end
end;

(* Compute composite center of mass *)
for j := 1 to 3 do
begin
    temp[i,j] := 0.0;
    for k := 1 to 3 do
begin
    temp[i,j] := temp[i,j] + U[i+1,j,k]*Nc[tt-1,i+1,k];
end;
Nc[tt,i,j] := (1/Mass[i])*mass[i]*s[i,j] +
        Mass[i+1]*(temp[i,j] + p[i+1,j]));
end;
{* Compute composite inertia matrix for rigid bodies *}
{* i through N, Initialization *}
for j := 1 to 3 do
begin
  temp[i,j] := 0.0;
  for k := 1 to 3 do
  begin
    mat5[i,j,k] := 0.0;
    mat6[i,j,k] := 0.0;
  end;
end;

{* Computation of U*c, (U*c + p - c), and (s - c) *}
for j := 1 to 3 do
begin
  for k := 1 to 3 do
  begin
    temp[i,j] := temp[i,j] + U[i+1,j,k]*Nc[tt-1,i+1,k];
  end;
  temp[i,j] := temp[i,j] + p[i+1,j] - Nc[tt-1,i,j];
  temp[i,j] := s[i,j] - Nc[tt-1,i,j];
end;

{* Formation of dot product — result is an scalar *}
t := sqrt(temp[i,1]) + sqrt(temp[i,2]) + sqrt(temp[i,3]);
t1 := sqrt(temp[i,1]) + sqrt(temp[i,2]) + sqrt(temp[i,3]);

{* Multiplication of t and t1 by a 3X3 identity matrix *}
for j := 1 to 3 do
begin
  for k := 1 to 3 do
  begin
    if (j<>k) then
    begin
      mat1[i,j,k] := 0.0;
      mat2[i,j,k] := 0.0;
    end
    else
    begin
      mat1[i,j,k] := t;
      mat2[i,j,k] := t1;
    end;
  end;
end;

{* Compute V times V transpose *}
for j := 1 to 3 do
begin
  for k := 1 to 3 do
  begin
    mat3[i,j,k] := (temp[i,j]*temp[i,k]);
    mat4[i,j,k] := (temp[i,j]*temp[i,k]);
  end;
end;
{$* Compute U*E*U *}$
for $l := 1$ to $3$ do 
begin 
  for $j := 1$ to $3$ do 
  begin 
    for $k := 1$ to $3$ do 
    begin 
      $mat5[l,l,j] := mat5[l,l,j] + \ U[l,l,l,k]*NE[l,l,l,k,j]$; 
      end; 
    end; 
  end; 
for $l := 1$ to $3$ do 
begin 
  for $j := 1$ to $3$ do 
  begin 
    for $k := 1$ to $3$ do 
    begin 
      $mat6[l,l,j] := mat6[l,l,j]+mat5[l,l,k]*U[l,l,j,k]$; 
      end; 
    end; 
  end; 
{$* Compute the inertia matrix *}$
for $j := 1$ to $3$ do 
begin 
  for $k := 1$ to $3$ do 
  begin 
  end; 
end;
const

PI = 3.141593;
N = 6;   (* Number of degrees of freedom *)

(* Link twist angles *)
alphal = 0.0;   alpha2 = -90.0;   alpha3 = 0.0;
alphal4 = 90.0; alpha5 = -90.0;   alpha6 = 90.0;

(* Link length in meter *)
a1 = 0.0; a2 = 0.0; a3 = 0.4318;
a4 = -0.0203; a5 = 0.0; a6 = 0.0;

(* Link offset *)
d1 = 0.0; d2 = 0.2435; d3 = -0.0934;
d4 = 0.4331; d5 = 0.0; d6 = 0.0;

(* Mass of links *)
mass1 = 100.0; mass2 = 17.4; mass3 = 4.8;
mass4 = 0.82; mass5 = 0.34; mass6 = 0.09;

(* Position of the center of gravity *)
(* in the local coordinate system *)
s11 = 0.0; s12 = 0.0; s13 = 100.0;
s21 = 0.068; s22 = 0.006; s23 = -0.016;
s31 = 0.0; s32 = -0.07; s33 = 0.014;
s41 = 0.0; s42 = 0.0; s43 = -0.019;
s51 = 0.0; s52 = 0.0; s53 = 0.0;
s61 = 0.0; s62 = 0.0; s63 = 0.032;

(* distance between two adjacent coordinate system *)
p11 = 0.0; p12 = 0.0; p13 = 0.0;
p21 = 0.0; p22 = 0.2435; p23 = 0.0;
p31 = 0.4318; p32 = 0.0; p33 = -0.0934;
p41 = -0.0203; p42 = -0.4331; p43 = 0.0;
p51 = 0.0; p52 = 0.0; p53 = 0.0;
p61 = 0.0; p62 = 0.0; p63 = 0.0;

II1I1 = 0.0; II1I2 = 0.0; II1I3 = 0.0;
II1I2 = 0.0; II1I22 = 0.0; II1I23 = 0.0;
II1I3 = 0.0; II1I32 = 0.0; II1I33 = 0.35;

II2I1 = 0.13; II2I12 = 0.0; II2I13 = 0.0;
II2I2 = 0.0; II2I22 = 0.524; II2I23 = 0.0;
II2I3 = 0.0; II2I32 = 0.0; II2I33 = 0.539;
\[
\begin{align*}
I_{311} &= 0.066; & I_{312} &= 0.0; & I_{313} &= 0.0; \\
I_{321} &= 0.0; & I_{322} &= 0.0125; & I_{323} &= 0.0; \\
I_{331} &= 0.0; & I_{332} &= 0.0; & I_{333} &= 0.086; \\
I_{411} &= 1.8e-3; & I_{412} &= 0.0; & I_{413} &= 0.0; \\
I_{421} &= 0.0; & I_{422} &= 1.8e-3; & I_{423} &= 1.3e-3; \\
I_{431} &= 0.0; & I_{432} &= 0.0; & I_{433} &= 0.0; \\
I_{511} &= 0.3e-3; & I_{512} &= 0.0; & I_{513} &= 0.0; \\
I_{521} &= 0.0; & I_{522} &= 0.3e-3; & I_{523} &= 0.0; \\
I_{531} &= 0.0; & I_{532} &= 0.0; & I_{533} &= 0.4e-3; \\
I_{611} &= 0.15e-3; & I_{612} &= 0.0; & I_{613} &= 0.0; \\
I_{621} &= 0.0; & I_{622} &= 0.15e-3; & I_{623} &= 0.0; \\
I_{631} &= 0.0; & I_{632} &= 0.0; & I_{633} &= 0.04e-3; \\
\end{align*}
\]

\{'* Joint Limits *'

\begin{align*}
\text{MaxLimit}_1 &= 2.7925; & \text{MinLimit}_1 &= -2.7925; \\
\text{MaxLimit}_2 &= 0.0; & \text{MinLimit}_2 &= 0.0; \quad \text{(unimportant)} \\
\text{MaxLimit}_3 &= 3.9270; & \text{MinLimit}_3 &= -0.7854; \\
\text{MaxLimit}_4 &= 2.9670; & \text{MinLimit}_4 &= -2.9670; \\
\text{MaxLimit}_5 &= 2.3562; & \text{MinLimit}_5 &= -2.3562; \\
\text{MaxLimit}_6 &= 2.9670; & \text{MinLimit}_6 &= -2.9670; \\
\end{align*}

\{'* Maximum Joint Torques in N-m *'

\begin{align*}
\text{MaxTau}_1 &= 97.6; & \text{MaxTau}_2 &= 186.4; & \text{MaxTau}_3 &= 89.4; \\
\text{MaxTau}_4 &= 24.2; & \text{MaxTau}_5 &= 20.1; & \text{MaxTau}_6 &= 21.3; \\
\end{align*}

\{'* Delay within the Strict Computation *'

\begin{align*}
\text{StrictDelay} &= 1.74E-3; \\
\end{align*}

\{'* Non-Strict Computation Delay *'

\begin{align*}
\text{delay} &= 6.3E-4; \\
\end{align*}

146
type
array2b = array[1..2] of boolean;
array2r = array[1..2] of real;
array6 = array[1..6] of real;
array7 = array[1..7] of real;
array200 = array[1..20000] of real;
array44 = array[1..4,1..4] of real;
array2006 = array[1..20000,1..6] of real;
array63 = array[1..6,1..3] of real;
array65 = array[1..6,0..5] of real;
array66 = array[1..6,1..6] of real;
array73 = array[1..7,1..3] of real;
array6200 = array[1..6,1..20000] of real;
array20063 = array[0..20000,1..6,1..3] of real;
array20066 = array[0..20000,1..6,1..6] of real;
array20067 = array[0..20000,1..6,1..7] of real;
array20073 = array[0..20000,1..7,1..3] of real;
array200733 = array[0..20000,1..7,1..3,1..3] of real;
array633 = array[1..6,1..3,1..3] of real;
array733 = array[1..7,1..3,1..3] of real;
character = packed array[1..20] of char;

var
i, ii, (* Index from link N to 1 *)
j,k,l,jj,tt: integer; (* Index from link i-1 to 1 *)
theta : array6; (* Joint angles *)
alpha : array6; (* Link twist angles *)
Mass : array7; (* Composite mass *)
mass : array6; (* Mass of individual links *)
U : array733; (* Transformation matrix *)
mat1 : array633; (* temporary matrix location *)
mat2 : array633; (* temporary matrix location *)
mat3 : array633; (* temporary matrix location *)
mat4 : array633; (* temporary matrix location *)
mat5 : array633; (* temporary matrix location *)
mat6 : array633; (* temporary matrix location *)
E : array733; (* Composite inertia matrix *)
NE : array200733; (* Non-Strict composite IM *)
II : array633; (* Link moment of the IM *)
temp : array63; (* Temporary location *)
temp1 : array63; (* Temporary location *)
c : array73; (* Comp. center of mass *)
Nc : array20073; (* Comp. center of mass, N/S *)
p : array63; (* vector from i to i+1 coord. *)
s : array63; (* vector, i to center of mass *)
f : array63; (* Force vector *)
NF : array20063; (* Non-Strict force vector *)
n : array63; (* Moment vector *)
Nn : array20063;  /* Non-strict moment vector */
H : array66;  /* Inertia matrix */
t, t1 : real;  /* Temporary locations */
Y, X : real;  /* Temporary locations */
T : array44;  /* Initial trans. matrix */
valid : array2b;  /* Flags in invkin routine */
diff : array2r;  /* Flags in invkin routine */
theta23 : real;  /* theta2 + theta3 */
SIGN : real;  /* Flag in arctan2 routine */
InitialTheta, FinalTheta : array6;  /* Initial joint angles */
FinalTime, DeltaTime, Ttime : real;  /* Sampling time period */
acceleration, velocity, position : array6200;  /* 6X1 acceleration vector */
a : array65;  /* 6X1 position vector */
torque : array2006;  /* 6X1 wrench */
PredTorque : array2006;  /* 6X1 predicted wrench */
MaxTorque : array6;  /* 6X1 joint torque limits */
PredictionFlag, Sflag : real;  /* Zero/first-order pred. flag */

{ 3X3 orientation matrix resolved in the reference  }
{ coordinate system  }
r11, r12, r13, r21, r22, r23, r31, r32, r33 : real;

{ corresponding 3X1 position vector  }
Px, Py, Pz : real;

{ 3X3 orientation matrix resolved in the first  }
{ coordinate system  }
t11, t12, t13, t21, t22, t23, t31, t32, t33 : real;

{ corresponding 3X1 position vector  }
Plx, Ply, Plz : real;

{ Data files  }
GoalPosDat,

path1, path2, path3, path4, path5, path6,

TorqueDat1, TorqueDat2, TorqueDat3,
TorqueDat4, TorqueDat5, TorqueDat6,

StrictTorqueDat1, StrictTorqueDat2, StrictTorqueDat3,
StrictTorqueDat4, StrictTorqueDat5, StrictTorqueDat6,

PredTorqueDat1, PredTorqueDat2, PredTorqueDat3,
PredTorqueDat4, PredTorqueDat5, PredTorqueDat6 : text;
procedure init; external;
procedure invkin; external;
procedure trajectory; external;
procedure JointTorques; external;
procedure inertia; external;
procedure Ninertia; external;
procedure proc(i : integer); external;

invdyn : invkin.o init.o inertia.o proc.o Ninertia.o \ 
        trajectory.o JointTorques.o invdyn.o 
        pc -ffpa -O invkin.o init.o inertia.o proc.o \ 
        Ninertia.o trajectory.o JointTorques.o \ 
        invdyn.o -o invdyn

invkin.o : invkin.p const.h vars.h invdyn.h 
pc -c -ffpa -O invkin.p

inertia.o : inertia.p const.h vars.h invdyn.h 
pc -c -ffpa -O inertia.p

Ninertia.o : Ninertia.p const.h vars.h invdyn.h 
pc -c -ffpa -O Ninertia.p

trajectory.o : trajectory.p const.h vars.h invdyn.h 
pc -c -ffpa -O trajectory.p

proc.o : proc.p const.h vars.h invdyn.h 
pc -c -ffpa -O proc.p

JointTorques.o : JointTorques.p const.h vars.h invdyn.h 
pc -c -ffpa -O JointTorques.p

init.o : init.p const.h vars.h invdyn.h 
pc -c -ffpa -O init.p

invdyn.o : invdyn.p const.h vars.h invdyn.h 
pc -c -ffpa -O invdyn.p
B.2 Program Listing of the Simulator, Part 2: Performance Analysis

This section contains the second part of the simulator used for computing the effective delay and also computing the joint torque errors.

```
program std(input,output,outfile,
  TorqueDat1,TorqueDat2,TorqueDat3,
  TorqueDat4,TorqueDat5,TorqueDat6,
  PredTorqueDat1,PredTorqueDat2,PredTorqueDat3,
  PredTorqueDat4,PredTorqueDat5,PredTorqueDat6,
  emaDat1,emaDat2,emaDat3,emaDat4,emaDat5,emaDat6,
  bmaDat1,bmaDat2,bmaDat3,bmaDat4,bmaDat5,bmaDat6);

type
  array200 = array[1..20000] of real;

const
  [* Incremental time by which the curves are shifted *]
  inc = 10.0E-6;
  [* Sampling rate for computing standard deviation *]
  hinc = 1.0E-4;
```
var

{* Running indecies *}
i, j, k, l, e, n;

{* Offset to data files to eliminate the transients *}
offset,

{* Number data points in TorqueDat files *}
Npoints,

{* Equal to 1 for zero output prediction, 1 for first *}
Iindex,

{* Strict or non-strict flag *}
flag,

{* Prediction flag *}
PredictionFlag,

{* Output prediction flag *).
OutPredFlag : integer;

{* Exact torque data files *}
TorqueDat1, TorqueDat2, TorqueDat3,
TorqueDat4, TorqueDat5, TorqueDat6,

{* Predicted torque data files *}
PredTorqueDat1, PredTorqueDat2, PredTorqueDat3,
PredTorqueDat4, PredTorqueDat5, PredTorqueDat6,

{* holds the effective delays *}
outfile,

{* Exact - Actual data files *}
emaDat1, emaDat2, emaDat3,
emaDat4, emaDat5, emaDat6,

{* Best Fit - Actual data files *}
bmaDat1, bmaDat2, bmaDat3,
bmaDat4, bmaDat5, bmaDat6 : text;

 tt, tl, t2,            /* Time */
 Torque,               /* Torque data array */
 PredTorque : array200; /* Predicted torque data array */
h,                    /* Instanteous sampling time */
Etau, Xtau,            /* Exact torque at a given time */
Ntau,                 /* Non-strict torque */
Btau,                 /* Best fit torque */
acc,                  /* Accumulating sum of errors */
std, oldstd,          /* Standard deviation */
effectiveDelay,       /* Effective delay */
diff,                 /* Torque difference */
midTime : real;        /* Midpoint in time */
procedure p1;
begin
  e := 1;
  for j := 1 to i-1 do begin
    midTime := (t2[j]+t2[j+1])/2.0;
    while ((midTime > t1[e]) & (e <= Npoints)) do e := e + 1;
    if midTime <= t1[e] then begin
      Etau := (Torque[e]-Torque[e-1])/(t1[e]-t1[e-1])*
                 (midTime-t1[e-1]) + Torque[e-1];
      diff := Etau-PredTorque[j];
      case k of
        1: writeln(emaDat1,midTime:10:7,' ',diff:13:9);
        2: writeln(emaDat2,midTime:10:7,' ',diff:13:9);
        3: writeln(emaDat3,midTime:10:7,' ',diff:13:9);
        4: writeln(emaDat4,midTime:10:7,' ',diff:13:9);
        5: writeln(emaDat5,midTime:10:7,' ',diff:13:9);
        6: writeln(emaDat6,midTime:10:7,' ',diff:13:9);
      end;
    end;
  end;
end;

procedure p2;
begin
  e := 1;
  if OutPredFlag = 1 then Iindex := 1
  else Iindex := 2;
  for j := Iindex to i-1 do begin
    midTime := (t2[j]+t2[j+1])/2.0;
    if OutPredFlag = 1 then Xtau := PredTorque[j]
    else
      Xtau := (PredTorque[j] - PredTorque[j-1])/
                (t2[j]-t2[j-1])*midTime-t2[j]) + PredTorque[j];
    while ((midTime > tt[e]) & (e <= Npoints)) do e := e + 1;
    if midTime <= tt[e] then begin
      Btau := (Torque[e]-Torque[e-1])/(tt[e]-tt[e-1])*
                  (midTime-tt[e-1]) + Torque[e-1];
      diff := Btau - Xtau;
      case k of
        1: writeln(bmaDat1,midTime:10:7,' ',diff:13:9);
        2: writeln(bmaDat2,midTime:10:7,' ',diff:13:9);
        3: writeln(bmaDat3,midTime:10:7,' ',diff:13:9);
        4: writeln(bmaDat4,midTime:10:7,' ',diff:13:9);
        5: writeln(bmaDat5,midTime:10:7,' ',diff:13:9);
        6: writeln(bmaDat6,midTime:10:7,' ',diff:13:9);
      end;
    end;
  end;
end;
begin
  reset(PredTorqueDat1,'PredTorqueDat1');
  reset(PredTorqueDat2,'PredTorqueDat2');
  reset(PredTorqueDat3,'PredTorqueDat3');
  reset(PredTorqueDat4,'PredTorqueDat4');
  reset(PredTorqueDat5,'PredTorqueDat5');
  reset(PredTorqueDat6,'PredTorqueDat6');

  reset(TorqueDat1,'TorqueDat1');
  reset(TorqueDat2,'TorqueDat2');
  reset(TorqueDat3,'TorqueDat3');
  reset(TorqueDat4,'TorqueDat4');
  reset(TorqueDat5,'TorqueDat5');
  reset(TorqueDat6,'TorqueDat6');

  rewrite(emaDat1); rewrite(emaDat2); rewrite(emaDat3);
  rewrite(emaDat4); rewrite(emaDat5); rewrite(emaDat6);

  rewrite(bmaDat1); rewrite(bmaDat2); rewrite(bmaDat3);
  rewrite(bmaDat4); rewrite(bmaDat5); rewrite(bmaDat6);

  rewrite(outfile);

  writeln;
  write('Strict or Non-Strict [1 or 2]? ');
  readln(flag);

  if flag = 1 then offset := 1
  else
    begin
      write('Interprocessor Prediction:
            Zero-order or First-order [ 1 or 2]? ');
      readln(PredictionFlag);

      write('Output Prediction:
            Zero-order or First-order [ 1 or 2]? ');
      readln(OutPredFlag);

      if PredictionFlag = 1 then
        begin
          if OutPredFlag = 1 then offset := 3
          else offset := 4;
        end
      else
        begin
          if OutPredFlag = 1 then offset := 4
          else offset := 5;
        end;
    end;

  write('Measurement analysis for joint #?[Enter 0 to QUIT] ');
  readln(k);
while k <> 0 do
begin
  case k of
  1: begin
    i := 0; Npoints := 0;
    while not eof(PredTorqueDat1) do
      begin
        i := i + 1;
        readln(PredTorqueDat1,t2[i],PredTorque[i]);
      end;
    while not eof(TorqueDat1) do
      begin
        Npoints := Npoints + 1;
        readln(TorqueDat1,t1[Npoints],Torque[Npoints]);
      end;
    p1;
  end;
  end;

  2: begin
    i := 0; Npoints := 0;
    while not eof(PredTorqueDat2) do
      begin
        i := i + 1;
        readln(PredTorqueDat2,t2[i],PredTorque[i]);
      end;
    while not eof(TorqueDat2) do
      begin
        Npoints := Npoints + 1;
        readln(TorqueDat2,t1[Npoints],Torque[Npoints]);
      end;
    p1;
  end;

  3: begin
    i := 0; Npoints := 0;
    while not eof(PredTorqueDat3) do
      begin
        i := i + 1;
        readln(PredTorqueDat3,t2[i],PredTorque[i]);
      end;
    while not eof(TorqueDat3) do
      begin
        Npoints := Npoints + 1;
        readln(TorqueDat3,t1[Npoints],Torque[Npoints]);
      end;
    p1;
  end;

end;
4: begin
    i := 0; Npoints := 0;
    while not eof(PredTorqueDat4) do
        begin
            i := i + 1;
            readln(PredTorqueDat4, t2[i], PredTorque[i]);
        end;
        while not eof(TorqueDat4) do
            begin
                Npoints := Npoints + 1;
                readln(TorqueDat4, t1[Npoints], Torque[Npoints]);
            end;
        pi;
    end;

5: begin
    i := 0; Npoints := 0;
    while not eof(PredTorqueDat5) do
        begin
            i := i + 1;
            readln(PredTorqueDat5, t2[i], PredTorque[i]);
        end;
        while not eof(TorqueDat5) do
            begin
                Npoints := Npoints + 1;
                readln(TorqueDat5, t1[Npoints], Torque[Npoints]);
            end;
        pl;
    end;

6: begin
    i := 0; Npoints := 0;
    while not eof(PredTorqueDat6) do
        begin
            i := i + 1;
            readln(PredTorqueDat6, t2[i], PredTorque[i]);
        end;
        while not eof(TorqueDat6) do
            begin
                Npoints := Npoints + 1;
                readln(TorqueDat6, t1[Npoints], Torque[Npoints]);
            end;
        pl;
    end;
for j := 1 to Npoints do tt[j] := t1[j];
std := 1.0E9;
l := 0;
repeat
  oldstd := std;
  acc := 0.0;
  h := t2[offset] + hinc;
  e := 1;
  n := 1;
  while (h < t2[i-1]) do
    begin
      while (h > tt[e]) do
        e := e + 1;
      Etau := (Torque[e] - Torque[e-1])/(tt[e]-tt[e-1])* (h-tt[e-1]) + Torque[e-1];
      while (h > t2[n]) do
        n := n + 1;
      if OutPredFlag = 1 then Ntau := PredTorque[n-1]
      else
        Ntau := (PredTorque[n-1] - PredTorque[n-2])/
        (t2[n-1]-t2[n-2])* (h-t2[n-1]) + PredTorque[n-1];
      acc := acc + sqr(Etau - Ntau);
      h := h + hinc;
    end;
  std := sqrt(acc);
  for j := 1 to Npoints do tt[j] := tt[j] + inc;
  l := l + 1;
until(std > oldstd);
writeln(outfile,'  torque r  f)c:l:0,
       ' effective delay (in msec)= ',(l-1)*inc*1.0E3:8:3,
       ' std = ',oldstd:13:8);
for j := 1 to Npoints do tt[j] := tt[j] - inc;
p2;
writeln;
write('Measurement analysis for joint #?
[Enter 0 to QUIT ] ');
readln(k);
end;
end.
REFERENCES


163


