Parallel Algorithms for Dynamics of Robotic Manipulators

by

Christopher Burke Pond
B Math , University of Waterloo, 1991

ACCEPTED
ULTY OF GRADUATE STUDIES
[redacted]
DEAN
E Sept 2/93

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF APPLIED SCIENCE
in the
Department of Mechanical Engineering

We accept this thesis as conforming
to the required standard

[redacted]

Dr I Sharf, Supervisor (Dept of Mechanical Engineering)

[redacted]

Dr Y Stepanenko, Department Member (Dept of Mechanical Engineering)

[redacted]

Dr K F Li, Outside Member (Dept of Electrical and Computer Engineering)

[redacted]

Dr G C Shoja, External Examiner (Dept of Computer Science)

University of Victoria

Supervisor: Dr. Inna Sharf

Abstract

The advent of relatively inexpensive parallel computing systems has motivated the investigation of parallel algorithms and architectures as a means of achieving more efficient robot dynamics algorithms. In this thesis, both the inverse dynamics and the forward (simulation) dynamics of rigid body manipulators are considered. The parallelism inherent in six inverse dynamics formulations is analysed to determine the most efficient algorithm, both theoretically and on a network of transputers. An extensive comparison of the parallel performance of the algorithms is made by incorporating a hardware model of the transputer into a scheduling algorithm which searches for the best assignment of tasks to processors. Thus, new results comparing the performance of the algorithms including communication costs are presented. Although the logarithmic Recursive Newton-Euler algorithm for inverse dynamics is theoretically the fastest, its performance is constrained by the need for more tasks and more communications. The Resolved Newton-Euler algorithm is shown to execute the fastest on a transputer network, a result which is supported by analysis. A recently proposed macroparallel simulation dynamics algorithm is implemented as a proof-of-concept. The analysis of this algorithm concentrates on parallel performance and comparison with its serial implementation. Performance models are partly based on experimental measurements rather than theoretical predictions and are used to assess the effects of serial parts of the algorithm on speedup and total execution time. Results indicate that minor serial computations in the algorithm can have a significant effect on the overall parallel performance.

Examiners

[redacted]

Dr I Sharf, Supervisor (Dept of Mechanical Engineering)

[redacted]

Dr Y Stepanenko, Department Member (Dept of Mechanical Engineering)

[redacted]

Dr K F Li, Outside Member (Dept of Electrical and Computer Engineering)

[redacted]

Dr G C Shoja, External Examiner (Dept of Computer Science)

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my advisor, Professor Inna Sharf, for her constant assistance and guidance throughout my program, and for her financial support. Her motivating influence is greatly appreciated.

I am also grateful for the technical assistance of Professor Alan Wagner of the Department of Computer Science, University of British Columbia. He has provided me with the opportunity and parallel computer facilities to conduct the experiments discussed in this thesis.

To my parents

Chapter 1

Introduction

1.1 Problem Statement

A growing number of robotics applications are requiring a new generation of manipulators. These new manipulators may be lightweight, moving at high speeds, or may be operated remotely over large distances. Such applications in turn demand ever more sophisticated controllers to cope with the increased complexity of the manipulators and their environments. Lightweight manipulators tend to exhibit structural flexibility, which can have a significant effect on the manipulator's motion even at moderate speeds. Teleoperation also leads to transmission delays which must be accounted for.

To meet the challenges of the new applications, it is therefore necessary to model and control the motion of these new manipulators. However, the goals of accurate modelling and high-speed control are not entirely compatible; achieving one makes the satisfaction of the other more difficult. Improved mathematical models of manipulator dynamics help us to predict and control the motion of advanced manipulators, but they are more complex and time-consuming to solve. On the other hand, high-speed control allows less time in which to evaluate the dynamics equations. Efficient dynamics algorithms thus form a necessary part of the reconciliation of these two goals.

There are two dynamics problems commonly associated with robotic manipulators, both of which are addressed in this thesis. The first is the *inverse dynamics* problem, which is to calculate the control forces necessary to achieve a desired motion. This problem is often solved periodically as part of a control algorithm as a means of using a dynamics model of the manipulator to account for the dynamics effects in the control law. The second problem is the *simulation* (or *forward*) *dynamics* problem, the solution of which produces the accelerations of the manipulator at a particular time resulting from the given control forces. The accelerations are then integrated to determine the velocities and positions of the manipulator at the next time step of the simulation.

To achieve accurate modelling and high-speed control, it is desirable to solve the dynamics problems as quickly as possible. Obviously, improvements in speed can be realized by using faster computer systems. In addition to faster processors, parallel processing offers an intuitively appealing method of decreasing program execution time. Many researchers have proposed and implemented parallel algorithms for the inverse dynamics problem, such as LEE & CHANG [1986], HASHIMOTO & KIMURA [1989] and IZAGUIRRE *et al.* [1992]. On the other hand, parallel simulation dynamics algorithms are still few and remain an active area of research. New parallel algorithms can be created either by specifically formulating an algorithm in a manner suitable for parallel computation, or by partitioning an existing serial algorithm into tasks. For example, BINDER & HERZOG [1986] propose an approximate inverse dynamics algorithm which uses prediction to eliminate precedence constraints, while ZHENG & HEMAMI [1986] formulate the inverse dynamics in state space. Several other inverse dynamics are discussed in Chapter 2 in detail. For the simulation dynamics problem, researchers have concentrated on parallelizing existing serial algorithms as in LEE & CHANG [1988] and scheduling them on microprocessor systems [KASAHARA *et al.*, 1987].

To further reduce the time required to solve a problem it is desirable to improve the efficiency of the algorithm being used. In this thesis, we are only concerned with total execution time and, hence, the term *efficient* refers to the relative speed with which an algorithm solves a given problem. Other efficiency issues, such as memory usage and speedup per number of processors, are not addressed. This is because memory and processors are relatively inexpensive and, for the applications considered here, the execution time is not limited by the availability of these resources. Indeed, experimental results presented in the thesis confirm that the optimal number of processors for typical robotic applications is less than the size of most networks in practice.

Typically, it is much simpler to propose a parallel algorithm than it is to predict its performance on a given computer system. To guide the design and improvement of parallel algorithms, it is helpful to have an idea of the factors which affect the performance of the new algorithm. Computational complexity analysis of *serial* algorithms has been established as a useful method for comparing the performance of serial algorithms, partly because it is independent of the system hardware and architecture. Comparing the performance of *parallel* algorithms, however, is not as simple. To determine the total execution time of a parallel algorithm in terms of the number of floating-point operations required, we must specify which operations can be performed in parallel. This depends on the choice of tasks and on the levels of parallelism being exploited, which in turn determine the cost of each task. Specialized hardware to exploit the lowest levels of parallelism is not always available or affordable. There is also additional overhead introduced in the parallel implementation For example, the communication costs incurred in message-passing architectures are difficult to deduce from the serial implementation of an algorithm Since the performance of parallel algorithms can be highly dependent on the parallel computer architecture, the architecture must be included in the algorithm design process and any subsequent performance comparisons

Many researchers have proposed or implemented parallel computer architectures for the robot dynamics problems For the inverse dynamics problem, some designs employ commercial microprocessors [NIGAM & LEE, 1985, ZHANG & PAUL, 1986], while others use specialized processors such as custom-built microprocessors [KOKAJI, 1986], bit-serial processors [RAHMAN & MEYER, 1987], Very-Long-Instruction-Word processors [ANDERSSON, 1989], or processors custom-designed for robotics applications [FIJANY & BEJCZY, 1991, SADAYAPPAN *et al* 1989] Similarly, for the simulation dynamics problem, LIAO & CHERN [1985] have proposed using array processors, while MCMILLAN *et al* [1991] have used a Cray Y-MP8 supercomputer

Recent technological advances have made practical, affordable, parallel computer systems commercially available In particular, the *transputer* has become a popular component of relatively inexpensive parallel computer systems Each transputer constitutes a single processing element in a message-passing architecture With four serial communications links, the transputer's modular design allows networks of transputers to be easily assembled and connected to host computers To fully support a parallel processing environment, the transputer also has a micro-coded scheduler which allows each transputer to execute multiple processes Many of the quantitative results presented in this thesis are dependent on data obtained on a transputer network and are therefore specific to transputers For the inverse dynamics problem, (though not for the simulation dynamics algorithms), many researchers have published similar experimental results for transputers which will be cited for comparison.

1.2 Thesis Outline

This thesis is divided into two main chapters, addressing first the inverse dynamics problem and then the simulation dynamics problem. The objective of this thesis is to investigate the performance of several robot dynamics algorithms by quantitative comparison of performance models with implementation results. Though the numerical performance measurements are specific to transputer networks, the accompanying analysis is more general. The analysis of the inverse dynamics algorithms can be applied to other message-passing architectures, while the analysis of the simulation dynamics algorithm is less specific and can be applied to shared-memory architectures as well. All algorithms are applied to rigid body, serial chain manipulators.

Though researchers continue to propose new parallel inverse dynamics algorithms for robotics, a comprehensive comparison of their performance is not currently available. Chapter 2 makes such a comparison for a representative sample of inverse dynamics algorithms, both theoretically in terms of operation counts, and experimentally in terms of execution time. First, a standard basis for comparison is established in §2.1 by determining which level of parallelism will be exploited, what the common notation for expressing the algorithms will be, and how computation costs will be tabulated. After stating the inverse dynamics algorithms, the maximum parallelism inherent in each at the chosen level is depicted in a *task graph* which shows each task as a node connected to the nodes upon which it depends. The task graphs provide a visually intuitive understanding of the parallel properties of the algorithms and the concepts relevant to the performance analysis. In particular, the length of the longest path through the graph represents the total computational cost of the algorithm independent of computer architecture Analytical expressions for this quantity are given for each of the inverse dynamics algorithms considered

Since communication costs depend on the parallel computer system, a hardware model of the transputer is proposed and verified in §2 4 1 This model is incorporated into a *scheduler* which searches for the most efficient assignment of tasks to processors The resulting schedules make accurate predictions of the actual execution speeds of each of the algorithms An extensive comparison of the best schedules is then conducted which reinforces some of the analytical performance results and brings to light some important differences

The purpose of Chapter 3 is to analyze the performance of a recently proposed simulation dynamics algorithm specifically designed for parallel implementation Unlike other simulation dynamics algorithms, the algorithm exploits a *macro* level of parallelism unavailable to other algorithms Because robot simulation dynamics algorithms are more complex than inverse dynamics algorithms in general, practicality dictates that performance analysis techniques different from those of Chapter 2 must be applied Instead of using a hardware model, the performance analysis uses timing measurements from serial and parallel implementations to determine which parts of the algorithm most affect parallel performance The experimental results and performance analysis presented in Chapter 3 thus provide a necessary proof of concept of the macroparallel simulation dynamics algorithm

Finally, Chapter 4 concludes this thesis with a summary of the results

Chapter 2

Parallel Inverse Dynamics Algorithms

Currently, there exists a wide variety of inverse dynamics algorithms for robotic manipulators. Some follow different formulations, and some employ different computational techniques, but one of the primary goals of each is to solve the inverse dynamics problem as quickly as possible. Most of these algorithms were not specifically designed for parallel implementation but, nevertheless, all algorithms have some inherent parallelism.

The first step in comparing these algorithms is to classify them at a *conceptual* level, either according to formulation or according to large-scale computational structure, such as recursion. Six algorithms were chosen which exemplify the main conceptual differences in inverse dynamics algorithms. They are: Recursive Newton-Euler (RECNE and RECNEb), Resolved Newton-Euler (RESNE), Classical Lagrangian (CLASSLE), Recursive Lagrangian (RECLE), and a hybrid algorithm (HYBRID).

It is also necessary to establish a common basis for *performance* comparison of the algorithms which is independent of hardware and architecture Most importantly, the algorithms must be expressed within a common level of parallelism Another obvious impediment to comparison is the variety of notation in use, not only among algorithms based on the same formulations, but also for the same quantities in different algorithms The next section of this chapter defines the level of parallelism and the notation used for comparison of the six aforementioned algorithms

With the standard basis of comparison established, §2 2 proceeds to describe the formulations of the algorithms, their differences, their advantages and disadvantages, and the tasks associated with each one In §2 3, the parallelism inherent in each algorithm is presented graphically as a directed task graph Task graphs allow the minimum theoretical execution time to be determined as the length of the longest path through the graph

More realistic estimates of the total execution time must include communication costs, which in turn depend on the location of the tasks on the processors Section 2 4 describes *scheduling algorithms* which assign the tasks onto a set of processors in such a way as to minimize the total execution time (which would result in an *optimal* schedule) For each task, a schedule names the processor on which the task executes and the start time of the task

Instead of implementing the code for each algorithm, the schedules themselves can be used for performance prediction A hardware model of a transputer-based message-passing multicomputer is described in §2 4 and its accuracy is verified In the final section, we compare the schedules obtained to determine the fastest algorithm and the optimal number of processors

2.1 A Standard Basis for Comparison

This section describes the standards chosen to allow an easy and fair comparison of the computational methods and costs for each of the algorithms.

2.1.1 Levels of Parallelism

For every algorithm, there are various levels at which parallelism can be exploited. In FIJANY & BEJCZY [1991], three levels of parallelism are described for the inverse dynamics problem of a serial chain manipulator: the *problem*, *link*, and *matrix-vector* levels. The problem level, or *macro* level, assigns the calculations for each body in the chain to a distinct processor. At the next lower level, the link level, calculations which pertain to a particular body are performed in parallel if possible. Finally, at the matrix-vector level, as the name suggests, multiplications and additions required for matrix-vector calculations, such as matrix-vector multiplication, are performed in parallel. It should be noted that the matrix-vector level of parallelism can be applied to all of the inverse dynamics algorithms considered here. Usually, specialized hardware is necessary for efficient use of parallelism at this level [SADAYAPPAN *et al.*, 1989, FIJANY & BEJCZY, 1991]. Since such hardware is expensive, and often not commercially available, we do not consider parallelism at the matrix-vector level.

In this chapter, we concentrate on link level parallelism only. This is a natural level in which to express the algorithms since each task generally corresponds to the calculation of some physical quantity. There are exceptions, however, especially in RECLE, where intermediate tasks are introduced to facilitate recursion. Note that by choosing the tasks at the link level first, the link level includes the problem level as a special case. Thus, the problem level assigns all tasks pertaining to a particular link to the same processor. Fixing the task assignment in this way, however, precludes the most efficient scheduling of tasks

2.1.2 Nomenclature

The first algorithm described in this chapter is the recursive Newton-Euler algorithm RECNE The notation used is taken from YOSHIKAWA [1990] The same notation is used whenever an identical quantity appears in the other algorithms For subsequent algorithms, the notation of the main reference paper is preserved, with the restriction that quantities common to previous algorithms are referred to by the existing notation The choice of coordinate frames for RECNE does not follow YOSHIKAWA [1990] but instead follows FIJANY & BEJCZY [1991] in which the body frame is placed at the distal end of the link (Figure 2 1) for a savings of a few operations Whether or not such a choice of coordinate frames results in a similar savings for the other algorithms is unclear However, the difference should be very minor It is more important that the same choice of frames be used for all of the algorithms to allow common notation and so that similar computations in different algorithms become apparent

Unless otherwise stated, superscripts and subscripts are chosen such that the symbol

$$ {}^{j}\mathbf{X}_{i,rs} $$

represents the quantity $\mathbf{X}$ for the ith link expressed in the jth coordinate frame Σ_j A comma among the subscripts represents partial differentiation with respect to the generalized coordinates with the succeeding indices, here implied to be q_r and q_s Also, vector quantities such as acceleration and velocity are defined relative to an inertial frame regardless of the frame in which they are expressed (Since the base frame, Σ_0, may be accelerating, the inertial frame is implied to be Σ_{-1}) Overdots represent differentiation with respect to time as usual and $\tilde{\mathbf{x}}$ is the skew-symmetric matrix representation of the operator $\mathbf{x}\times$ Note that vectors and matrices appear in boldface

The following summarizes the notation used for the fundamental quantities in the inverse dynamics algorithms

$\mathbf{e}_z$ The standard basis vector $[0, 0, 1]^T$

${}^i\mathbf{F}_i$ Total force acting on link i

${}^i\mathbf{f}_i$ Total interbody force exerted on link i by link $i-1$

${}^i\mathbf{g}$ Acceleration due to gravity

g_i Gravitational torques

${}^i\mathcal{I}_i$ Constant inertia tensor with respect to Σ_i

${}^i\mathbf{J}_i$ Constant inertia tensor with respect to the centre of mass of link i

M_{ij} Elements of the $n \times n$ inertia matrix $\mathbf{M}$

m_i Mass of link i

${}^i\mathbf{N}_i$ Total torque acting on link i

${}^j\mathbf{n}_i$ Total interbody torque exerted on link i by link $i-1$

n Number of joints/degrees-of-freedom of the manipulator

${}^i\hat{\mathbf{p}}_i$ Constant position of the origin of Σ_i with respect to the origin of Σ_{i-1}

${}^j\mathbf{p}_i$ Position of the origin of Σ_i with respect to the origin of Σ_j

${}^j\ddot{\mathbf{p}}_i$ Linear acceleration of Σ_i

q_i Angular joint displacement, generalized variable for a revolute joint

${}^{j}\mathbf{R}_i$ Rotation matrix representing the orientation of Σ_i relative to Σ_j

${}^{i}\hat{\mathbf{s}}_i$ Constant position of the centre of mass of link i with respect to the origin of Σ_i

${}^{i}\mathbf{s}_i$ Linear acceleration of the centre of mass of link i

τ_i Control torque applied to link i about z-axis of Σ_{i-1}

${}^{i}\boldsymbol{\omega}_i$ Angular velocity

${}^{i}\boldsymbol{\omega}_i$ Angular acceleration

A schematic diagram of link i is shown in Figure 2 1 illustrating the associated position vectors and points of application of the forces acting on link i. The coordinate frame Σ_{i-1} is located at the *proximal* (to the base) end of the link while Σ_i is located at the *distal* end and is fixed to link i.

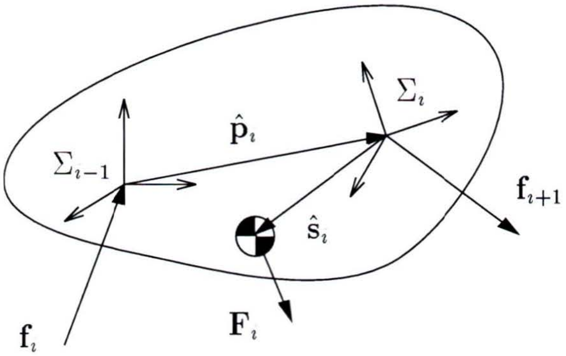

Figure 2 1 Schematic diagram of link i

2.1.3 Computational Costs

The most common method of comparing the computational cost of an algorithm, and the first method applied in this chapter, is to count the number of operations required. The next section provides tables listing the required tasks and their costs for each algorithm. At the link level of parallelism, the cost of each task can be expressed in terms of the operations given in Table 2.1. Each operation is listed with its equivalent cost in terms of scalar floating-point multiplications (M), additions (A), and trigonometric function evaluations (T) for 3 dimensional quantities. In some instances, the cost will be given in terms of M, A, and T directly where such operations are explicit in the algorithm.

Operation	Symbol	Cost
Matrix-matrix multiplication	MM	27M+18A
Matrix-vector multiplication	MV	9M+6A
Matrix addition	MA	9A
Vector cross-product	VC	6M+3A
Vector dot product	VD	3M+2A
Vector outer product	VO	9M
Vector addition	VA	3A
Scalar-matrix product	SM	9M
Scalar-vector product	SV	3M
Trace	TR	2A

Table 2.1. Task operation costs for 3 dimensional vectors.

It is useful to give an example of how these operation costs are evaluated. Consider the following expressions taken from the algorithms of the next section.

Task	Cost
${}^{i}\boldsymbol{\omega}_i = {}^{i-1}\mathbf{R}_i{}^T({}^{i-1}\boldsymbol{\omega}_{i-1} + \mathbf{e}_z q_i)$	1SV+1VA+1MV

$$
M_{ij} \;=\; \begin{array}{l} \mathbf{e}_z^{T\,i-1}\mathbf{R}_{j-1}{}^{j-1}\mathbf{I}_j\mathbf{e}_z \\ \quad + \left(\mathbf{e}_z \times {}^{i-1}\mathbf{p}_{j-1}\right)^{T\,i-1}\mathbf{R}_{j-1}\left(\mathbf{e}_z \times {}^{j-1}\mathbf{L}_j\right) \end{array} \qquad \begin{cases} 3\text{MV}+2\text{VD} & \\ \;+2\text{VC}+\text{A} & i < j \\ 1\text{MV}+1\text{VD} & i = j \end{cases}
$$

Because of the unique structure of $\mathbf{e}_z$, the evaluation of $\mathbf{e}_z q_i$ in the first expression can clearly be implemented with no cost using efficient programming. Since we are attempting to compare the efficiencies of the algorithms, however, the effect of programming technique complicates the analysis. We therefore distinguish between the algorithm and its implementation as a program, and ignore the program. Thus, in our cost analysis, we consider only those computational properties of the tasks, such as sparseness, which are *explicit* in the task description. The costs given for the two expressions above follow by considering $\mathbf{e}_z$ to be an arbitrary 3-dimensional vector.

There are some exceptions, however. If, for some values of its indices a quantity is zero or the identity matrix, then another cost is quoted to take this into account. The cost for M_{ij} above reflects this, since for $i = j$, ${}^{i-1}\mathbf{p}_{j-1} = \mathbf{0}$ and ${}^{i-1}\mathbf{R}_{j-1}$ is the identity matrix. Also, operations between constant quantities, such as the addition of two constant vectors, cost nothing since they can be performed offline. Rather than defining a new symbol for the result, the quantities remain distinct for clarity. The task descriptions list the costs explicitly to illustrate both of these simplifications.

To establish a common basis of comparison, we have chosen to express the costs of the inverse dynamics algorithms in terms of the operations of Table 2.1. From the discussion above, however, it is clear that some costs can be reduced by expanding the vectors and matrices in terms of their components and simplifying the resulting expressions symbolically. As we will see later in this chapter, two of the algorithms could benefit significantly from exploitation of the sparseness of some of the matrices used. Several researchers have used symbolic simplification not only to express the algorithms but also to tailor the equations to a specific manipulator of interest [IZA-

GUIRRE *et al.*, 1992, VUKOBRATOVIĆ *et al.*, 1988, LI & SANKAR, 1992]. Published performance results incorporating these techniques will be cited later for comparison. This thesis does not, however, consider symbolic simplification.

2.2 Formulations

Although robot inverse dynamics algorithms are usually formulated for both revolute and prismatic joints, we consider in this chapter only the case of serial chain manipulators with single degree-of-freedom joints revolute about $\mathbf{e}_z$ of coordinate frame Σ_{i-1}. This simplifies the performance analysis yet does not significantly limit the applicability of the comparisons.

We also have not accounted for the application of arbitrary end-effector forces, but, unless explicitly indicated otherwise, have assumed them to be zero. The primary reason for doing so is to exclude any tasks which use quantities related to links or coordinate frames beyond n in order to simplify the depiction of the task graphs. The complete analysis can be easily extended to include end-effector forces.

2.2.1 Recursive Newton-Euler

The Recursive Newton-Euler algorithm (RECNE) is probably the best known dynamics formulation in robotics and has been used in many applications because of its simplicity and efficiency [XIAO & CHENG, 1991, FIJANY & BEJCZY, 1991]. The equations of motion of a rigid body are derived from Newton's second law and Euler's equation. The computational cost is greatly reduced if calculations for each link are performed in the coordinate frame fixed to that link, Σ_i, instead of in the base frame Σ_0. In this case, the inertia tensor in Euler's equation, ${}^{i}\mathbf{J}_i$, is constant. This choice of coordinate frames requires that the velocities and accelerations of link i (with respect to Σ_{-1}), and the forces and torques on the link, be expressed in Σ_i Since the velocity of the base of the manipulator with respect to the inertial frame Σ_{-1} is known (usually $\mathbf{0}$), the calculation of the velocities of the links can begin at that end and proceed along the length of the chain in what is known as *forward recursion* Similarly, for a known force on the manipulator's end-effector (assumed to be zero here), the interbody forces can be calculated for each link beginning at the last and proceeding to the base in what is referred to as *backward recursion* RECNE has proven to have a very efficient computational structure and is currently the most efficient general serial algorithm for inverse dynamics [FEATHERSTONE, 1987]

Table 2 2 lists the quantities calculated for each link i according to the algorithm of YOSHIKAWA [1990] (except that Σ_i has been moved from the proximal end of the link to the distal end) Since calculations are performed within a link's local coordinate frame, but involve vectors associated with neighbouring links expressed in their own coordinate frames, equation (2 1) calculates the coordinate rotation matrices ${}^{i-1}\mathbf{R}_i$ These are necessary to transform a vector from one coordinate frame Σ_i to the neighbouring link's coordinate frame Σ_{i-1} This first step is common to all of the algorithms described in this chapter Equations (2 2–2 5) express the forward kinematic recursion for the velocities and accelerations Newton's law is then applied (2 6) to obtain the total force ${}^i\mathbf{F}_i$ on body i and, similarly, Euler's equation (2 7) calculates the total torque The backward recursion (2 8, 2 9) calculates the interbody forces ${}^i\mathbf{f}_i$, and the interbody torques ${}^i\mathbf{n}_i$, exerted on link i by link $i-1$ Finally, the interbody torques are projected onto the axis of rotation to determine the desired applied torques as in equation (2 12) Note that gravity can be included in an efficient manner by setting ${}^0\mathbf{p}_0 = {}^0\mathbf{g}$ The gravitational acceleration of each body will then be carried along implicitly by the forward recursion

Task	Cost	
${}^{i-1}\mathbf{R}_i$	4M+2T	(2 1)
${}^{i}\boldsymbol{\omega}_i = {}^{i-1}\mathbf{R}_i{}^T({}^{i-1}\boldsymbol{\omega}_{i-1} + \mathbf{e}_z q_i)$	1SV+1VA+1MV	(2 2)
${}^{i}\boldsymbol{\omega}_i = {}^{i-1}\mathbf{R}_i{}^T({}^{i-1}\boldsymbol{\omega}_{i-1} + \mathbf{e}_z \ddot{q}_i + {}^{i-1}\boldsymbol{\omega}_{i-1} \times \mathbf{e}_z q_i)$	2SV+2VA+1VC+1MV	(2 3)
${}^{i}\ddot{\mathbf{p}}_i = {}^{i-1}\mathbf{R}_i{}^T\,{}^{i-1}\mathbf{p}_{i-1} + {}^{i}\boldsymbol{\omega}_i \times {}^{i}\hat{\mathbf{p}}_i + {}^{i}\boldsymbol{\omega}_i \times ({}^{i}\boldsymbol{\omega}_i \times {}^{i}\hat{\mathbf{p}}_i)$	1MV+3VC+2VA	(2 4)
${}^{i}\ddot{\mathbf{s}}_i = {}^{i}\ddot{\mathbf{p}}_i + {}^{i}\boldsymbol{\omega}_i \times {}^{i}\hat{\mathbf{s}}_i + {}^{i}\boldsymbol{\omega}_i \times ({}^{i}\boldsymbol{\omega}_i \times {}^{i}\hat{\mathbf{s}}_i)$	3VC+2VA	(2 5)
${}^{i}\mathbf{F}_i = m_i\,{}^{i}\ddot{\mathbf{s}}_i$	1SV	(2 6)
${}^{i}\mathbf{N}_i = {}^{i}\mathbf{J}_i\,{}^{i}\boldsymbol{\omega}_i + {}^{i}\boldsymbol{\omega}_i \times ({}^{i}\mathbf{J}_i\,{}^{i}\boldsymbol{\omega}_i)$	2MV+1VC+1VA	(2 7)
${}^{i}\mathbf{f}_i = \begin{cases} {}^{i}\mathbf{F}_i + {}^{i}\mathbf{f}_{i+1} \\ {}^{i}\mathbf{F}_i \end{cases}$	1VA $i < n$ 0 $i = n$	(2 8)
${}^{i}\mathbf{n}_i = \begin{cases} {}^{i}\mathbf{N}_i + {}^{i}\mathbf{n}_{i+1} + ({}^{i}\hat{\mathbf{p}}_i + {}^{i}\hat{\mathbf{s}}_i) \times {}^{i}\mathbf{F}_i + {}^{i}\hat{\mathbf{p}}_i \times {}^{i}\mathbf{f}_{i+1} \\ {}^{i}\mathbf{N}_i + ({}^{i}\hat{\mathbf{p}}_i + {}^{i}\hat{\mathbf{s}}_i) \times {}^{i}\mathbf{F}_i \end{cases}$	2VC+3VA $i < n$ 1VC+1VA $i = n$	(2 9)
${}^{i-1}\mathbf{f}_i = {}^{i-1}\mathbf{R}_i\,{}^{i}\mathbf{f}_i$	1MV	(2 10)
${}^{i-1}\mathbf{n}_i = {}^{i-1}\mathbf{R}_i\,{}^{i}\mathbf{n}_i$	1MV	(2 11)
$\tau_i = {}^{i-1}\mathbf{n}_i \quad \mathbf{e}_z$	1VD	(2 12)

Table 2 2 RECNE algorithm and computation costs

It should be pointed out that equations (2 10) and (2 11) can be included directly in the backward recursion (equations 2 8 and 2 9) by replacing ${}^{i}\mathbf{f}_{i+1}$ with ${}^{i}\mathbf{R}_{i+1}{}^{i+1}\mathbf{f}_{i+1}$ However, the calculations for each link would then require both of the rotation matrices ${}^{i-1}\mathbf{R}_i$ and ${}^{i}\mathbf{R}_{i+1}$ Equations (2 10) and (2 11) were introduced as separate tasks by FIJANY & BEJCZY [1991] to avoid the dependency on ${}^{i}\mathbf{R}_{i+1}$ Eliminating this dependency leads to a more efficient parallel implementation by reducing the amount of communication which may be required

Many variations of the RECNE algorithm exist which simplify the computation in certain cases. For example, the operator Ω was introduced by LI & SANKAR [1992] to eliminate the common subexpression

$$\begin{aligned} \Omega &= {}^{i}\dot{\tilde{\boldsymbol{\omega}}}_{i} + {}^{i}\tilde{\boldsymbol{\omega}}_{i}\,{}^{i}\tilde{\boldsymbol{\omega}}_{i} & (2.13) \\ &= {}^{i}\dot{\boldsymbol{\omega}}_{i} \times + {}^{i}\boldsymbol{\omega}_{i} \times ({}^{i}\boldsymbol{\omega}_{i} \times) \end{aligned}$$

appearing in equations (2.4) and (2.5) of the forward recursion. It can be shown, however, that this operator costs about fifty percent more than the two expressions which it replaces. A savings can be realized if symbolic simplification is used to evaluate Equation 2.13 as was originally intended by Li & Sankar.

Another variation of RECNE is proposed by LATHROP [1985] using the technique of *logarithmic* parallelism. LEE & CHANG [1986] pursue this technique by formulating RECNE in the base frame, which requires that all quantities be expressed or transformed into that frame (hence we will denote this algorithm RECNEb). They show that the resulting homogeneous linear recurrence relations can be evaluated using a recursive doubling technique which has a computational complexity of $O(\lg n)$ (where $\lg \triangleq \log_2$) rather than $O(n)$.[1] Figure 2.2 shows the overlapped binary tree structure used to accomplish this. Computation proceeds from the leaves to the roots of the binary trees, where each leaf corresponds to a different link in the chain. Thus, the evaluation time is $O(\lg n)$, the height of the tree. The recursive calculations for ${}^{0}\mathbf{R}_{i}$, ${}^{0}\boldsymbol{\omega}_{i}$, ${}^{0}\dot{\boldsymbol{\omega}}_{i}$, ${}^{0}\mathbf{p}_{i}$, ${}^{0}\mathbf{f}_{i}$, and ${}^{0}\mathbf{n}_{i}$ are all performed using this technique. The other tasks require only a constant amount of computation, resulting in an $O(\lg n)$ computational complexity for the entire algorithm.

The original motivation for using the body coordinate frames was to reduce both the number of rotation matrices needed and the amount of computation by using a

---

[1] In this thesis, we will follow the notation employed in the robotics literature and use $O(\,)$ to represent any asymptotic bound and not strictly an asymptotic upper bound.

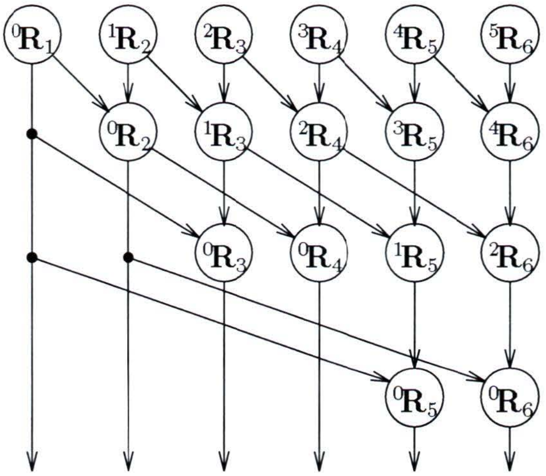

Figure 2.2 Recursive doubling technique

constant matrix ${}^{i}\mathbf{J}_i$. In fact, the cost of evaluating ${}^{0}\mathbf{J}_i$ is so great that RECNEb also evaluates Euler's equation (2.7) in the body coordinate frames and transforms the results back to the base frame. The cost of the extra transformations, however, is more than offset by the logarithmic parallelism as will be shown in §2.3.

2.2.2 Resolved Newton-Euler

Hashimoto & Kimura derived a new inverse dynamics algorithm which they called the Resolved Newton-Euler (RESNE) algorithm, so named because it resolves the task of calculating the manipulator inverse dynamics into subtasks [HASHIMOTO & KIMURA, 1989]. It differs from RECNE, however, because it is based on Kane's dynamics equations and uses the concept of partial velocities to determine the desired torques. Thus, the calculation of velocities and torques is different, but other quantities are calculated in the same manner as in RECNE.

In presenting the algorithm in Table 2.3, we use a right superscript to indicate the ith partial velocity. Thus, equations (2.15–2.17) calculate the ith partial angular and linear velocities for link i. To calculate the angular velocity of link i, the products of the partial angular velocities and the corresponding joint rates are summed, beginning with equation 2.18. As shown in equations 2.19, the addition of each term to the sum is listed as a separate task, where the right superscript in parentheses indicates the loop index in the sum. The calculation of the partial velocities and the summing for the angular velocity are performed in descending order, beginning with the current link index. For consistency with the other algorithms described in this chapter, the loop index is omitted when the summing for the angular velocity is complete, thus, ${}^i\boldsymbol{\omega}_i \equiv {}^i\boldsymbol{\omega}_i^{(1)}$.

The calculations of (2.19) are performed in descending order so that local information from the current link is utilized first, then information from its neighbour, and so on to the base body. Thus, information which must travel the farthest is needed last, allowing time for it to propagate along the chain. These steps essentially accomplish the forward kinematics recursion, similarly to RECNE, except that the recursion is for partial velocities. The balanced flow of information will become more apparent when the task graph is described in the next section.

The following equations, from (2.20) to (2.24), perform the forward acceleration recursion and then employ Newton's and Euler's equations, exactly as in RECNE. All of the necessary information is now available to calculate the torques according to Kane's equations. The value of each torque results from summing the contributions from each of the partial velocities and, as before, each term of the sum has been given as a separate task (2.25).

By setting ${}^0\ddot{\mathbf{p}}_0 = {}^0\mathbf{g}$, the algorithm as stated can also account for the acceleration due to gravity, similarly to RECNE. This is unlike what was done in HASHIMOTO & KIMURA [1989] where a forward recursion was used to find each ${}^i\mathbf{g}$, the gravitational acceleration in Σ_i.

Task	Cost	
${}^{i-1}\mathbf{R}_i$	4M+2T	(2.14)
${}^{i}\boldsymbol{\omega}_i^i = {}^{i-1}\mathbf{R}_i^T \mathbf{e}_z$	1MV	(2.15)
${}^{i}\mathbf{p}_i^i = {}^{i}\boldsymbol{\omega}_i^i \times {}^{i}\hat{\mathbf{p}}_i$	1VC	(2.16)
${}^{i}\mathbf{s}_i^i = {}^{i}\mathbf{p}_i^i + {}^{i}\boldsymbol{\omega}_i^i \times {}^{i}\hat{\mathbf{s}}_i$	1VC+1VA	(2.17)
${}^{i}\boldsymbol{\omega}_i^{(i)} = {}^{i}\boldsymbol{\omega}_i^i q_i$	1SV	(2.18)
For $k = i-1, \ldots, 1$		
$\Big[\ {}^{i}\boldsymbol{\omega}_i^k = {}^{i-1}\mathbf{R}_i^T\, {}^{i-1}\boldsymbol{\omega}_{i-1}^k$	1MV	(2.19)
$\ \ {}^{i}\mathbf{p}_i^k = {}^{i-1}\mathbf{R}_i^T\, {}^{i-1}\mathbf{p}_{i-1}^k + {}^{i}\boldsymbol{\omega}_i^k \times {}^{i}\hat{\mathbf{p}}_i$	1MV+1VC+1VA	
$\ \ {}^{i}\mathbf{s}_i^k = {}^{i}\mathbf{p}_i^k + {}^{i}\boldsymbol{\omega}_i^k \times {}^{i}\hat{\mathbf{s}}_i$	1VC+1VA	
$\ \ {}^{i}\boldsymbol{\omega}_i^{(k)} = {}^{i}\boldsymbol{\omega}_i^{(k+1)} + {}^{i}\boldsymbol{\omega}_i^k q_k\ \Big]$	1SV+1VA	
${}^{i}\boldsymbol{\omega}_i = {}^{i-1}\mathbf{R}_i^T({}^{i-1}\boldsymbol{\omega}_{i-1} + \mathbf{e}_z q_i + {}^{i-1}\boldsymbol{\omega}_{i-1} \times \mathbf{e}_z q_i)$	2SV+1MV+1VC+2VA	(2.20)
${}^{i}\mathbf{p}_i = {}^{i-1}\mathbf{R}_i^T\, {}^{i-1}\mathbf{p}_{i-1} + {}^{i}\boldsymbol{\omega}_i \times {}^{i}\hat{\mathbf{p}}_i + {}^{i}\boldsymbol{\omega}_i \times ({}^{i}\boldsymbol{\omega}_i \times {}^{i}\hat{\mathbf{p}}_i)$	1MV+3VC+2VA	(2.21)
${}^{i}\mathbf{s}_i = {}^{i}\mathbf{p}_i + {}^{i}\boldsymbol{\omega}_i \times {}^{i}\hat{\mathbf{s}}_i + {}^{i}\boldsymbol{\omega}_i \times ({}^{i}\boldsymbol{\omega}_i \times {}^{i}\hat{\mathbf{s}}_i)$	3VC+2VA	(2.22)
${}^{i}\mathbf{F}_i = m_i\, {}^{i}\mathbf{s}_i$	1SV	(2.23)
${}^{i}\mathbf{N}_i = {}^{i}\mathbf{J}_i\, {}^{i}\boldsymbol{\omega}_i + {}^{i}\boldsymbol{\omega}_i \times ({}^{i}\mathbf{J}_i\, {}^{i}\boldsymbol{\omega}_i)$	2MV+1VC+1VA	(2.24)
For $k = 1, \ldots, i-1$		
$\Big[\tau_k^{(i)} = \tau_k^{(i-1)} + {}^{i}\mathbf{N}_i \cdot {}^{i}\boldsymbol{\omega}_i^k + {}^{i}\mathbf{F}_i \cdot {}^{i}\mathbf{s}_i^k$	2VD+2A	(2.25)
$\tau_i^{(i)} = {}^{i}\mathbf{N}_i \cdot {}^{i}\boldsymbol{\omega}_i^i + {}^{i}\mathbf{F}_i \cdot {}^{i}\mathbf{s}_i^i$	2VD+1A	(2.26)

Table 2.3: RESNE algorithm and computation costs

Although RESNE requires partial velocities which were not necessary in RECNE, the backward recursion for the interbody forces has been completely eliminated and replaced by a forward running sum which accumulates the scalar torque values from the base to the tip (2.25–2.26).

2.2.3 Classical Lagrangian

Besides the Newton-Euler formulation, the dynamics equations of mechanical systems are often formulated using an approach based on the Lagrangian of the system and employing the Euler-Lagrange equations [HOLLERBACH, 1980]. The resulting equations are generally more compact and more easily analyzed than those of the Newton-Euler formulation, but are also more computationally expensive. The classical Lagrangian (CLASSLE) algorithm detailed in Table 2.4 follows that of YOSHIKAWA [1990].

In this formulation, the applied torques are expressed as a sum of the contributions from the inertial, velocity-dependent, and gravitational force terms, which in matrix form can be expressed as

$$\boldsymbol{\tau} = \mathbf{M}\ddot{\mathbf{q}} + \dot{\mathbf{q}}^T\mathbf{C}\dot{\mathbf{q}} + \mathbf{g} \tag{2.27}$$

In the above, $\boldsymbol{\tau}$ is the vector of joint torques, $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ are the vectors of joint velocities and accelerations, $\mathbf{M}$ is the inertia matrix, $\mathbf{C}$ is a third order tensor, and $\mathbf{g}$ is the vector of gravitational torques. The main task is to determine the elements M_{ij}, C^i_{jk}, and g_i of the matrices $\mathbf{M}$, $\mathbf{C}^i$, and $\mathbf{g}$. Table 2.4 gives the expressions for each of these elements, and it was the original idea of VUKOBRATOVIĆ *et al.* [1988] to calculate these elements in parallel.

	Task	Cost	
$^{i-1}\mathbf{T}_i$		$6\text{M}+2\text{T}$	(2.28)
$^{j}\mathbf{T}_i$	$= {}^{j}\mathbf{T}_k{}^{k}\mathbf{T}_i \quad 1 < i-j \leq n, \quad k = j + \left\lceil \frac{i-j}{2} \right\rceil$	1MM	(2.29)
$^0\mathbf{T}_{i,j}$	$= {}^0\mathbf{T}_{j-1}\boldsymbol{\Delta}^{j-1}\mathbf{T}_i$	$\begin{cases} 2\text{MM} & 1 \neq j \leq i \\ 1\text{MM} & 1 = j \leq i \end{cases}$	(2.30)
$^0\mathbf{T}_{i,jk}$	$= {}^0\mathbf{T}_{k-1}\boldsymbol{\Delta}^{k-1}\mathbf{T}_{j-1}\boldsymbol{\Delta}^{j-1}\mathbf{T}_i$	$\begin{cases} 4\text{MM} & 1 \neq k < j \leq i \\ 3\text{MM} & 1 = k < j \leq i \\ 2\text{MM} & 1 \neq k = j \leq i \end{cases}$	(2.31)
C^i_{jk}	$= \sum_{l=j}^{n} \mathrm{tr}\left({}^0\mathbf{T}_{l,jk}{}^l\mathbf{H}_l{}^0\mathbf{T}_{l,i}{}^T\right)$	$(n-j+1)(2\text{MM}+1\text{TR})$ $+(n-j)\text{A}$	(2.32)
M_{ij}	$= \sum_{l=i}^{n} \mathrm{tr}\left({}^0\mathbf{T}_{l,j}{}^l\mathbf{H}_l{}^0\mathbf{T}_{l,i}{}^T\right)$	$(n-i+1)(2\text{MM}+1\text{TR})$ $+(n-i)\text{A}$	(2.33)
g_i	$= -\sum_{j=i}^{n} m_j{}^0\mathbf{g}^T{}^0\mathbf{T}_{j,i}{}^j\hat{\mathbf{s}}_j$	$(n-i+1)(1\text{MV}+1\text{VD})$ $+(n-i)\text{A}$	(2.34)
τ_i'	$= \sum_{j=1}^{i} M_{ij}q_j + \sum_{j=i+1}^{n} M_{ji}q_j$	$n\text{M}+(n-1)\text{A}$	(2.35)
τ_i''	$= -\sum_{j=1}^{i-1}\sum_{k=1}^{j-1} C^j_{ik}(2q_jq_k) + \sum_{j=i+1}^{n}\sum_{k=1}^{j-1} C^i_{jk}(2q_jq_k)$	$[n(n-1)/2-i+1]\,\text{M}$ $+[n(n-1)/2-i]\,\text{A}$ $(+\text{A} \text{ if } i=n)$	(2.36)
τ_i'''	$= -\sum_{j=1}^{i-1} C^j_{ij}q_j^2 + \sum_{j=i+1}^{n} C^i_{jj}q_j^2$	$(n-1)\text{M}+(n-2)\text{A}$ $(+\text{A} \text{ if } i=n)$	(2.37)
τ_i	$= \sum_{j=1}^{n} M_{ij}q_j + \sum_{j=1}^{n}\sum_{k=1}^{n} C^i_{jk}q_jq_k + g_i$		(2.38)
	$= \tau_i' + \tau_i'' + \tau_i''' + g_i$	3A	(2.39)

Table 2.4: CLASSLE algorithm and computation costs

Here, ${}^{i}\mathbf{r}_i$ is the homogeneous position vector from the origin of Σ_i to a volume element dv of link i and ρ is the density of the volume element [YOSHIKAWA, 1990; PAUL, 1981]. The second inertial moment ${}^{i}\mathbf{G}_i$ is defined by equation (2.42) but using a 3-dimensional vector $\mathbf{r}_i$.

Instead of evaluating (2.27) to determine the torques all in one step, intermediate tasks are introduced in equations (2.35) to (2.37) which correspond to physical forces. In particular, $\boldsymbol{\tau}'$ represents the inertial torques, $\boldsymbol{\tau}''$ are the torques resulting from Coriolis forces, and $\boldsymbol{\tau}'''$ corresponds to centrifugal forces. Finally, the contributions of each of these is summed in (2.39) to determine the applied torques.

Since the matrices in equations (2.28–2.34) are expressed in homogeneous form and hence are 4-dimensional, the computation costs appearing in the right column of Table 2.4 represent the corresponding number of multiplications and additions for 4-dimensional quantities. Thus, the costs for matrix-vector operations are higher than those given in Table 2.1 for 3-dimensional quantities, highlighting an inherent computational disadvantage of the CLASSLE algorithm.

Although the expressions for the kinematic transformations are simplified by using ${}^{j}\mathbf{T}_i$, the sparseness of ${}^{j}\mathbf{T}_i$ (shown in Equation 2.40) makes the algorithm computationally more expensive than if the quantities ${}^{j}\mathbf{R}_i$ and ${}^{j}\mathbf{p}_i$ were used separately, as is done in the other algorithms. Multiplication by the matrix $\boldsymbol{\Delta}$ (as in equations 2.30 and 2.31) also counts as 64 multiplications and 48 additions, although it can be done in only 4 multiplications with specialized code. As mentioned earlier, in order to keep the cost comparison of the algorithms fair, all costs apply to the tasks of the algorithm as they are stated here and do not account for any improvements which may be possible in the code.

In cases where we do take advantage of some special properties, the appropriate parts of the algorithm are restated to make these properties explicit. For example, in the CLASSLE algorithm above, the elements of the coefficient matrices M_{ij} and C^i_{jk} are calculated as separate tasks. The symmetry and antisymmetry properties of these matrices are now an explicit part of the formulation and are exploited to reduce the total computational burden by reducing the number of tasks. As proven in VUKOBRATOVIĆ & KIRĆANSKI [1985], $\mathbf{M}$ and $\mathbf{C}^i$ are symmetric matrices and $C^j_{ik} = -C^i_{jk}$ for $k \leq i \leq j$. This implies that only $n(n+1)/2$ elements M_{ij} and $n(n-1)(n+1)/3$ elements C^i_{jk} need to be calculated.

Further performance improvements can be made by restating CLASSLE in terms of 3×3 matrices and vectors. By partitioning the 4×4 matrices of Equations (2.28–2.34) as is done in Equation (2.40) for ${}^j\mathbf{T}_i$, the costs of the tasks were reevaluated in terms of 3×3 matrix-vector operations. The resulting total cost for the algorithm is 30% to 40% lower than that of CLASSLE presented here. However, scheduling experiments presented later in the chapter show that such an improvement is not enough to make CLASSLE competitive.

2.2.4 Recursive Lagrangian

Computationally, the CLASSLE formulation of the preceding section can not match the efficiency of RECNE on a serial computer. To improve its efficiency, Hollerbach combined two features of RECNE with the CLASSLE algorithm – the recursive structure, and the use of 3×3 rotation matrices instead of the 4×4 homogeneous transformation matrices [HOLLERBACH, 1980]. These improvements reduced the $O(n^4)$ computational complexity of CLASSLE to $O(n)$, which is the same complexity as RECNE. The result is the Recursive Lagrangian algorithm, which we denote RECLE.

We can begin to understand the RECLE algorithm if we combine the expressions for C^i_{jk}, M_{ij}, g_i, and τ_i from (2.32), (2.33), (2.34), and (2.38) into

$$\tau_i = \sum_{l=i}^{n} \left[\mathrm{tr}\left({}^0\mathbf{T}_{l,i} \, {}^l\mathbf{H}_l \, {}^0\mathbf{T}_l^{\,T} \right) - m_l \, {}^0\mathbf{g}^T \, {}^0\mathbf{T}_{l,i} \, {}^l\hat{\mathbf{s}}_l \right] \tag{2.44}$$

where

$${}^0\mathbf{T}_l = \sum_{j=1}^{l} {}^0\mathbf{T}_{l,j} q_j + \sum_{j=1}^{i} \sum_{k=1}^{i} {}^0\mathbf{T}_{l,jk} q_j q_k \tag{2.45}$$

Instead of evaluating ${}^0\mathbf{T}_l$ as given above, the RECLE algorithm computes its components ${}^0\mathbf{R}_i$ and ${}^0\mathbf{p}_i$ recursively. First, ${}^0\mathbf{R}_i$ is expressed recursively in (2.48). Its second derivative gives a recursive equation for ${}^0\mathbf{R}_i$ in (2.50). In analogy with the 4×4 $\mathbf{\Delta}$ of the CLASSLE algorithm, Zalzala & Morris [1991] used the symbol $\mathbf{Q}$ to represent the constant 3×3 upper-left submatrix of $\mathbf{\Delta}$ defined so that ${}^{i-1}\mathbf{R}_{i,i} = \mathbf{Q}\,{}^{i-1}\mathbf{R}_i$. Hence, $\mathbf{Q}$ and $\mathbf{Q}^2$ are sparse matrices. Having computed ${}^0\mathbf{R}_i$, the linear accelerations ${}^0\mathbf{p}_i$ can then be calculated recursively using (2.52).

The strategy of replacing explicit sums by recursive expressions is continued for some of the other quantities. First, the sum operation is distributed over both terms in (2.44) and the trace and sum operations are exchanged in the first term using Equation (2.30) for ${}^0\mathbf{T}_{l,i}$. Then, $\mathbf{D}_i$ and $\mathbf{c}_i$ are defined to be the resulting sums in each of the two terms of (2.44) respectively. Recursive expressions can be derived for the 4-dimensional quantities $\mathbf{D}_i$ and $\mathbf{c}_i$ using the recursive property of ${}^0\mathbf{R}_i$ given in (2.48). Here, however, the algorithm is restated using 3-dimensional matrices by partitioning the 4-dimensional homogeneous matrices as previously discussed. The definitions of $\mathbf{D}_i$ and $\mathbf{c}_i$ remain the same (though referring to 3-dimensional quantities) while another recursive expression for $\mathbf{e}_i$ is used to calculate the extra terms created by reducing the size of $\mathbf{D}_i$. Other intermediate calculations are presented as separate tasks and are marked with a * superscript as in Zalzala & Morris [1991].

Task	Cost	
${}^{i-1}\mathbf{R}_i$	4M+2T	(2.46)
${}^{i-1}\mathbf{p}_i = {}^{i-1}\mathbf{R}_i {}^i\hat{\mathbf{p}}_i$	1MV	(2.47)
${}^0\mathbf{R}_i = {}^0\mathbf{R}_{i-1} {}^{i-1}\mathbf{R}_i$	1MM	(2.48)
${}^0\dot{\mathbf{R}}_i = \begin{cases} \dot{q}_i \mathbf{Q} {}^{i-1}\mathbf{R}_i & i = 1 \\ \left({}^0\dot{\mathbf{R}}_{i-1} + {}^0\mathbf{R}_{i-1}\mathbf{Q}\dot{q}_i\right) {}^{i-1}\mathbf{R}_i & i > 1 \end{cases}$	1MM+1SM 2MM+1MA+1SM	(2.49)
${}^0\ddot{\mathbf{R}}_i = \begin{cases} \left(\mathbf{Q}^2\dot{q}_i^2 + \mathbf{Q}\ddot{q}_i\right) {}^{i-1}\mathbf{R}_i & i = 1 \\ \big[{}^0\ddot{\mathbf{R}}_{i-1} + 2{}^0\dot{\mathbf{R}}_{i-1}\mathbf{Q}\dot{q}_i \\ \quad + {}^0\mathbf{R}_{i-1}\left(\mathbf{Q}^2\dot{q}_i^2 + \mathbf{Q}\ddot{q}_i\right)\big] {}^{i-1}\mathbf{R}_i & i > 1 \end{cases}$	1MM+1MA+2SM+1M 3MM+3MA+3SM+2M	(2.50)
${}^0\mathbf{R}_{i,i} = \begin{cases} \mathbf{Q} {}^{i-1}\mathbf{R}_i & i = 1 \\ {}^0\mathbf{R}_{i-1}\mathbf{Q} {}^{i-1}\mathbf{R}_i & i > 1 \end{cases}$	1MM 2MM	(2.51)
${}^0\ddot{\mathbf{p}}_i = {}^0\ddot{\mathbf{p}}_{i-1} + {}^0\ddot{\mathbf{R}}_i {}^i\hat{\mathbf{p}}_i$	1MV+1VA	(2.52)
$\mathbf{e}_i^* = m_i {}^0\ddot{\mathbf{R}}_i {}^i\hat{\mathbf{s}}_i$	1MV	(2.53)
$\mathbf{e}_i = \begin{cases} \mathbf{e}_{i+1} + \mathbf{e}_i^* + m_i {}^0\ddot{\mathbf{p}}_i & i < n \\ \mathbf{e}_i^* + m_i {}^0\ddot{\mathbf{p}}_i & i = n \end{cases}$	2VA+1SV 1VA+1SV	(2.54)
$\mathbf{D}_i^* = m_i {}^i\hat{\mathbf{s}}_i {}^0\ddot{\mathbf{p}}_i{}^T + {}^i\mathbf{G}_i {}^0\ddot{\mathbf{R}}_i{}^T$	1MM+1VO+1MA	(2.55)
$\mathbf{D}_i = \begin{cases} {}^i\mathbf{R}_{i+1}\mathbf{D}_{i+1} + {}^i\mathbf{p}_{i+1}\mathbf{e}_{i+1}{}^T + \mathbf{D}_i^* & i < n \\ \mathbf{D}_i^* & i = n \end{cases}$	1MM+1VO+2MA 0	(2.56)
$\mathbf{g}_i^* = {}^0\mathbf{R}_{i,i}{}^T {}^0\mathbf{g}$	1MV	(2.57)
$\mathbf{c}_i = \begin{cases} {}^i\mathbf{R}_{i+1}\mathbf{c}_{i+1} + m_i {}^i\hat{\mathbf{s}}_i & i < n \\ m_i {}^i\hat{\mathbf{s}}_i & i = n \end{cases}$	1MV+1VA 0	(2.58)
$\tau_i^{**} = \mathbf{g}_i^* \cdot \mathbf{c}_i$	1VD	(2.59)
$\tau_i^* = \mathrm{tr}({}^0\mathbf{R}_{i,i}\mathbf{D}_i)$	1MM+2A	(2.60)
$\tau_i = \tau_i^* - \tau_i^{**}$	1A	(2.61)

Table 2.5 RECLE algorithm and computation costs

2.2.5 Hybrid

An important advantage of the CLASSLE formulation (and to a lesser extent, RECLE) is that the final expressions for the applied torques are in a form which explicitly identifies the contributions of the inertial, velocity-dependent, and gravitational forces (2.39). This distinction is necessary for some control schemes and trajectory planning algorithms, such as, for instance, those which require the inertial and velocity-dependent torque contribution terms (but not the gravitational term) to be scaled to produce realizable torques [Zalzala & Morris, 1991].

It has also been observed that the joint accelerations can vary at frequencies that are an order of magnitude greater than those of the joint positions and velocities. Izaguirre *et al.* [1992] therefore propose a computational structure comprised of a high-priority synchronous task and low-priority background tasks. The high-priority task evaluates the product $\mathbf{Mq}$ and sums the three contributions to the torque (refer to (2.69)). The low-priority tasks update the inertial coefficients M_{ij} and the velocity-dependent and gravitational torque terms.

Having adopted the above torque decomposition, the main task is to calculate each of the force terms efficiently. Izaguirre *et al.* chose RECNE for the velocity-dependent terms and an efficient recursive procedure for the inertial and gravitational terms. By setting ${}^0\mathbf{p}_0 = \mathbf{0}$ (instead of ${}^0\mathbf{p}_0 = {}^0\mathbf{g}$) and $q_i \equiv 0$, RECNE produces the torques ${}^{i-1}\mathbf{n}_i$ which result from the velocity-dependent forces only. To calculate the inertial and gravitational terms, the concept of *compound links* is used in which compound link i is comprised of links i to n and considered a single rigid body. Conservation of momentum was used to derive backward recursions for the 1st moment of inertia and for the inertia tensor of the compound links, which are given in (2.65) and (2.66). The inertial coefficients and gravitational terms are then calculated directly using (2.67) and (2.68).

$$\begin{array}{lcll} & & \underline{\text{Task}} & \underline{\text{Cost}} \end{array}$$

$${}^{i-1}\mathbf{R}_i \qquad\qquad \text{4M+2T} \tag{2.62}$$

$${}^{j}\mathbf{R}_i = {}^{j}\mathbf{R}_k {}^{k}\mathbf{R}_i \quad 1 < i - j \leq n, \quad k = j + \left\lceil \frac{i-j}{2} \right\rceil \qquad \text{1MM} \tag{2.63}$$

$${}^{j}\mathbf{p}_i = \begin{cases} {}^{i-1}\mathbf{R}_i {}^{i}\hat{\mathbf{p}}_i & j = i - 1 \qquad \text{1MV} \\ {}^{j}\mathbf{p}_k + {}^{j}\mathbf{R}_k {}^{k}\mathbf{p}_i & j < i - 1 \qquad \text{1MV+1VA} \end{cases} \tag{2.64}$$

$${}^{i-1}\mathbf{L}_i = \begin{cases} {}^{i-1}\mathbf{R}_i \left({}^{i}\boldsymbol{\mathcal{L}}_i + {}^{i}\mathbf{L}_{i+1} + M_i {}^{i}\hat{\mathbf{p}}_i\right) & i < n \qquad \text{1MV+1VA} \\ {}^{i-1}\mathbf{R}_i \left({}^{i}\boldsymbol{\mathcal{L}}_i + M_i {}^{i}\hat{\mathbf{p}}_i\right) & i = n \qquad \text{1MV} \end{cases} \tag{2.65}$$

$${}^{i-1}\mathbf{I}_i = \begin{cases} {}^{i-1}\mathbf{R}_i \left({}^{i}\boldsymbol{\mathcal{I}}_i + {}^{i}\mathbf{I}_{i+1} + M_i {}^{i}\tilde{\hat{\mathbf{p}}}_i^{\,2}\right) {}^{i-1}\mathbf{R}_i^{\,T} & \\ \quad - {}^{i-1}\tilde{\mathbf{p}}_i {}^{i-1}\tilde{\mathbf{L}}_i - \left({}^{i-1}\tilde{\mathbf{p}}_i {}^{i-1}\tilde{\mathbf{L}}_i\right)^T & i < n \qquad \text{3MM+3MA} \\ {}^{i-1}\mathbf{R}_i \left({}^{i}\boldsymbol{\mathcal{I}}_i + M_i {}^{i}\tilde{\hat{\mathbf{p}}}_i^{\,2}\right) {}^{i-1}\mathbf{R}_i^{\,T} & \qquad\qquad \text{3MM+2MA} \\ \quad - {}^{i-1}\tilde{\mathbf{p}}_i {}^{i-1}\tilde{\mathbf{L}}_i - \left({}^{i-1}\tilde{\mathbf{p}}_i {}^{i-1}\tilde{\mathbf{L}}_i\right)^T & i = n \end{cases} \tag{2.66}$$

$$M_{ij} = \begin{array}{l} \mathbf{e}_z^{\,T} {}^{i-1}\mathbf{R}_{j-1} {}^{j-1}\mathbf{I}_j \mathbf{e}_z \\ \quad + \left(\mathbf{e}_z \times {}^{i-1}\mathbf{p}_{j-1}\right)^T {}^{i-1}\mathbf{R}_{j-1} \left(\mathbf{e}_z \times {}^{j-1}\mathbf{L}_j\right) \end{array} \qquad \begin{cases} \text{3MV+2VD} & \\ \quad \text{+2VC+A} & i < j \\ \text{1MV+1VD} & i = j \end{cases} \tag{2.67}$$

$$g_i = -{}^{0}\mathbf{g}^{T} {}^{0}\mathbf{R}_{i-1} \left(\mathbf{e}_z \times {}^{i-1}\mathbf{L}_i\right) \qquad \begin{cases} \text{1VC+1VD} & i = 1 \\ \text{1MV+1VC} & i > 1 \\ \quad \text{+1VD} & \end{cases} \tag{2.68}$$

RECNE calculates ${}^{i-1}\mathbf{n}_i$

$$\tau_i = \sum_{j=1}^{n} M_{ij} q_j + g_i + {}^{i-1}\mathbf{n}_i \quad \mathbf{e}_z \qquad n\text{M} + (n+1)\text{A} + \text{1VD} \tag{2.69}$$

Table 2.6 HYBRID algorithm and computation costs

In the expressions in Table 2.6, the constant masses, 1st moments, and inertia tensors of the links with respect to Σ_i are denoted by m_i, ${}^i\mathcal{L}_i$, and ${}^i\mathcal{I}_i$ respectively. The corresponding quantities for the compound link i with respect to Σ_j are M_i, ${}^j\mathbf{L}_i$, and ${}^j\mathbf{I}_i$ respectively.

The primary purpose of this algorithm is to calculate the desired torques using the decomposition previously discussed. Such a decomposition is desirable because different torque contributions are made explicit. This decomposition is one of the advantages of the CLASSLE algorithm and is achieved here by combining RECNE with an efficient recursion for the 1st moments and inertia tensors of the compound links. For these reasons, we refer to this algorithm as HYBRID.

The HYBRID algorithm is unique among those presented here because it is almost completely separable into two concurrent sub-algorithms, RECNE and the one given in Table 2.6. The task graph of the next section illustrates this property.

2.3 Theoretical Analysis of Parallelism

The descriptions of the inverse dynamics algorithms given in the previous section provide little insight into their properties with respect to parallel computation. In this section, the parallel computation structure of each algorithm will be presented and compared in a manner which is independent of the computer architecture on which the algorithm may be executed. The value of such a comparison is that its conclusions will reflect the fundamental properties of the parallelism in the algorithms and, therefore, will remain valid independent of implementation. The disadvantage of the comparison is that the analysis on which it is based does not include important aspects of the hardware implementation, such as communication costs, which can have a dramatic effect on the actual performance. Such considerations will be made in the subsequent sections

To compare the parallelism inherent in the algorithms, each algorithm is represented by a directed, acyclic, task graph in which each node represents a task and unidirectional edges represent precedence constraints The direction of the edges represent the direction of the flow of information during computation This is the most common way of representing the parallel structure of an algorithm Another method uses edges to represent all tasks, both computation and communication, and nodes to represent synchronization points [BARHEN, 1987]

The main objective of performance analysis is to determine the execution time of the algorithm Each node in the graph is assigned a value representing the computation cost of its task, either in terms of the number of operations required, or as the corresponding execution time Note that communication costs between processors can be accounted for by assigning weights to the edges between nodes as will be done in §2 4 The execution time of a sequence of tasks is determined by summing the weights of the nodes and edges along the corresponding path in the graph Since the inverse dynamics problem is solved periodically within a control algorithm, we assume that all of the inputs to the task graph are available at the same time The finishing time of the algorithm is therefore the *critical path length*, the length of the longest path through the graph When communication costs are not included, the longest path through the graph is referred to as the *computational* critical path The task graphs illustrate the maximum parallelism at a particular level and, hence, the computational critical path length represents the *fastest* possible execution time for an algorithm

The task graphs for the inverse dynamics algorithms of the previous section have certain properties which greatly simplify the determination of the computational critical paths Essential to the analysis presented here are the requirements that all tasks be always executed, the cost of each task be a known constant (in terms of the number of floating-point operations), and the graphs be acyclic, that is, they do not contain any loops. These properties imply that the computational critical path length is constant (for a particular n) and can be determined. The fact that there are no conditional tasks (branches) in the algorithms implies that every task in the graph will be executed, which in turn implies that the task graph is *static*. We have also assumed that the cost of each task is constant and not a function of the input to the algorithm.

Some parts of the algorithms can be expressed using loops, however. It is possible to define an entire loop to be a single task since the number of iterations through each loop is always known and, hence, the cost of executing a loop is constant. Because of the choice of tasks given in the previous section, however, each step in a loop is made a distinct task. Furthermore, the loops are unrolled so that the same step becomes a different task each time through the loop. This technique has the desired effect of removing any loops in the task graph representation of the algorithm. It is important from an algorithmic analysis perspective that the task graphs be acyclic. Since we are looking for the longest path through the graph, a cycle with positive weights would create a path of infinite length. This situation occurs because the task graphs can not specify the number of times a loop in the graph should be executed.

In Figures 2.3 to 2.7, the task graphs are presented for five algorithms (but not RECNEb) for a three-link manipulator ($n = 3$). The bold edges between nodes show the critical path without communication costs. Terminal nodes, *i.e.* those with either no predecessor or no successor, are also marked in bold.

The task graph for RECNE (Recursive Newton-Euler algorithm) is perhaps the simplest. Figure 2.3 clearly shows the tasks associated with each link as a vertical block. The recursive components of the algorithm are essentially serial computations and their effects on the critical path length are clearly visible. First the forward recursion for the linear accelerations dominates the computation, followed by the backward recursion for the interbody torques.

As mentioned earlier, RESNE eliminates the need for a backward recursion, and instead uses a forward running sum to calculate the torques. This is shown in Figure 2.4. Despite the extra tasks created by the calculation of the partial velocities, the critical path is identical to the forward half of the RECNE critical path, requiring just one final task instead of a backward recursion.

The RESNE task graph also shows why the summing for ${}^i\boldsymbol{\omega}_i$ is performed in descending order for each link, calculating the partial sums ${}^3\boldsymbol{\omega}_3^{(k)}$ from $k = 3, \ldots, 1$. Each of these tasks depends on the corresponding partial velocity ${}^3\boldsymbol{\omega}_3^k$. All of these, except for ${}^3\boldsymbol{\omega}_3^3$, depend on the partial velocities of the previous link, ${}^2\boldsymbol{\omega}_2^k$. Therefore, ${}^3\boldsymbol{\omega}_3^3$ will be completed first and the sum can begin with ${}^3\boldsymbol{\omega}_3^{(3)}$. The partial velocities ${}^2\boldsymbol{\omega}_2^k$ from the previous link will also be completed in descending order, so that their results will be available to the tasks ${}^3\boldsymbol{\omega}_3^k$ in descending order. Also, since the backward recursion has been replaced by a forward running sum, all information flows approximately down and to the right within the graph. By grouping some of the tasks shown into larger, approximately equal-sized tasks, Hashimoto & Kimura [1989] proposed a systolic pipelined architecture which takes advantage of these regular computation and communication patterns.

One of the main disadvantages of CLASSLE becomes obvious when we consider the corresponding task graph in Figure 2.5. The tasks are highly interconnected with no regular connection structure apparent. This lack of structure can have a dramatic effect on the parallel performance as it leads to an excessive amount of, perhaps non-nearest-neighbour, communication. It is also not clear how the graph generalizes to an n-link system, either in terms of which nodes are added or how they are connected.

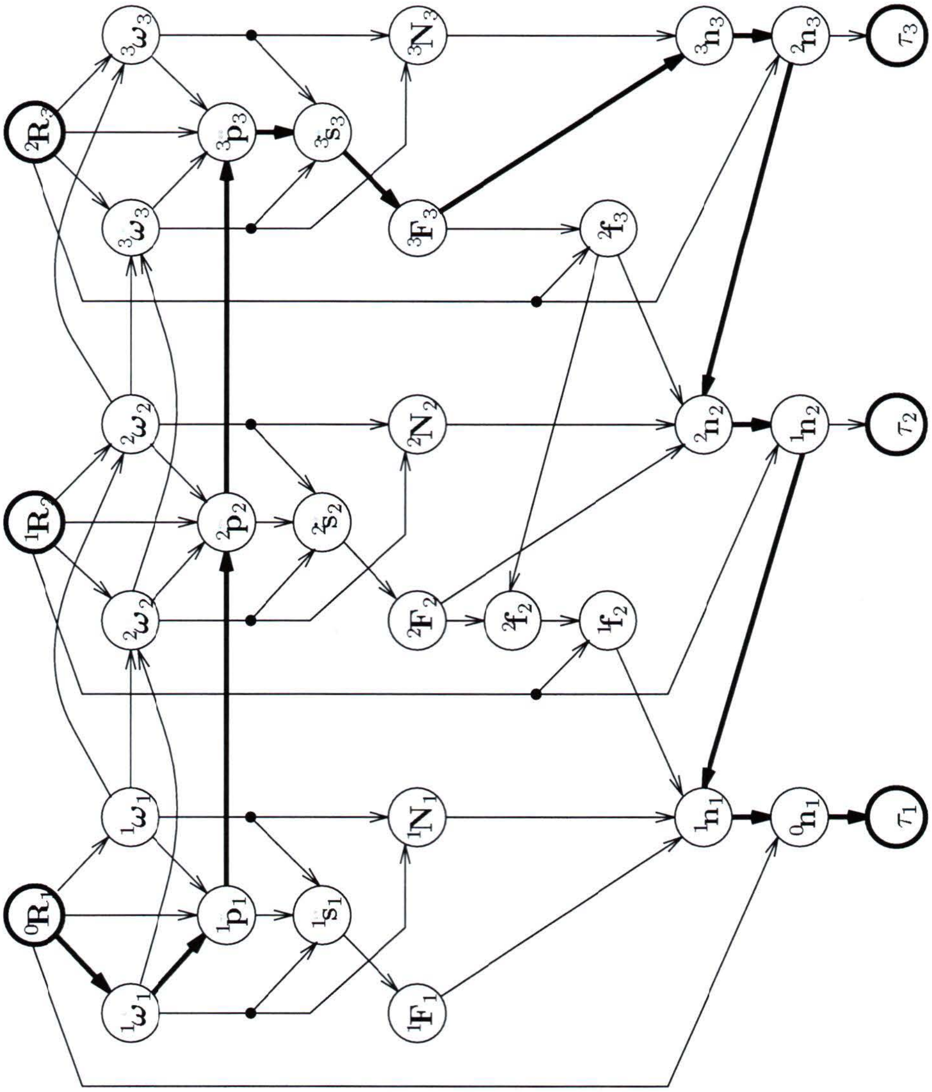

Figure 2.3: RECNE task graph for 3 links.

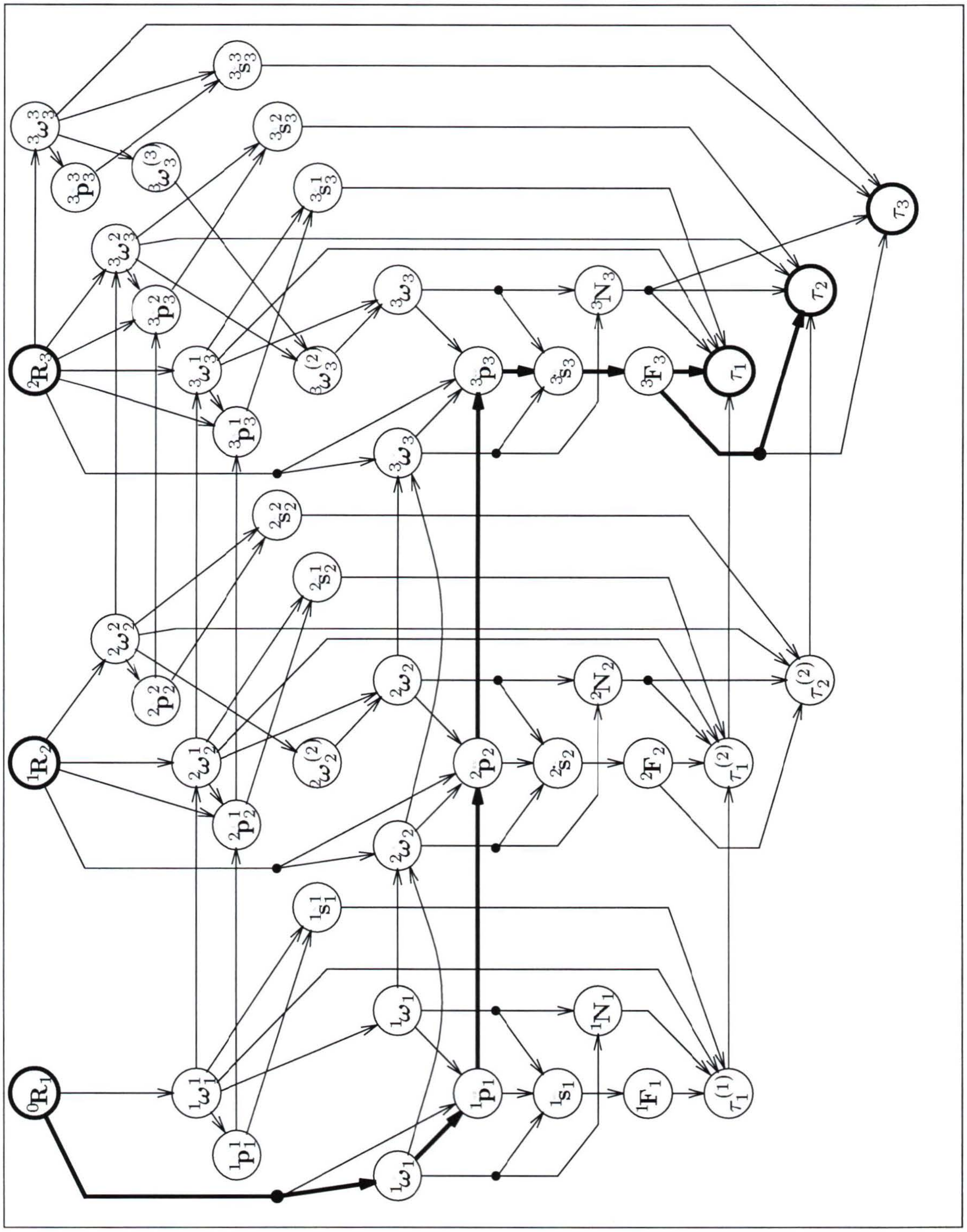

Figure 2.4 **RESNE** task graph for 3 links

In fact, the critical path takes entirely different routes depending on n. Besides the path shown in bold for n=3, the critical path may take the same route for other n or go through the tasks ${}^1\mathbf{T}_n$, ${}^0\mathbf{T}_{n,21}$, C_{21}^1, and τ_1''.

The recursive structure of RECLE is clearly indicated, similarly to RECNE, by the critical path through its task graph (Figure 2.6). For RECLE, however, the quantities being evaluated recursively do not have a physically intuitive meaning. Also, the critical path contains tasks which calculate matrix quantities which are more costly than the vector quantities in the RECNE critical path.

Finally, the task graph of the HYBRID inverse dynamics algorithm is shown in Figure 2.7. The two sub-algorithms, RECNE and a recursive computation using compound links, are completely concurrent, joined only at the very first and last tasks. Note that the critical path goes through the recursive part of the compound link sub-algorithm. This implies that improving the RECNE sub-algorithm (possibly by using RESNE) will not improve the overall performance of the HYBRID algorithm.

The computational critical paths depicted in the task graphs indicate the path through each graph having the greatest cost. From the task graphs for a 3-link system, it is possible to generalize the critical path to the n-link case. This was done for all of the algorithms with the exception of CLASSLE and RECNEb, for which the irregular structure of the task graph makes generalization difficult. Instead, expressions for critical paths within subgraphs must be combined to determine the overall critical path. Table 2.7 lists analytical expressions for the critical path lengths.

For comparison, the computational critical path lengths are plotted in Figure 2.8 for different numbers of bodies in the chain. Note that the cost for CLASSLE is predominantly linear for the range of chain lengths shown. Also, the logarithmic computational complexity of RECNEb results in the shortest computational critical path for $n > 6$. Thus, both RESNE and RECNEb are faster than the more popular

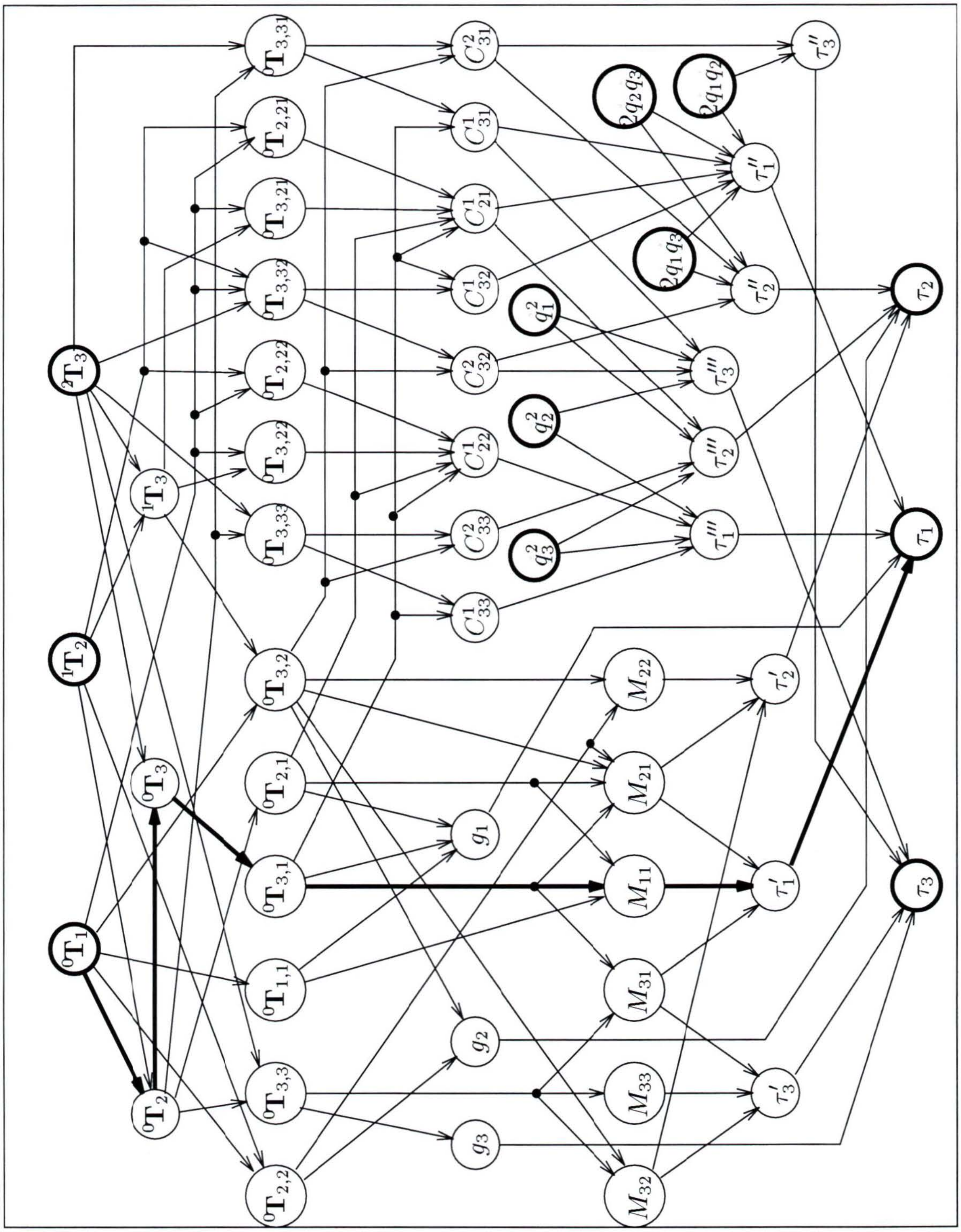

Figure 2.5: CLASSLE task graph for 3 links.

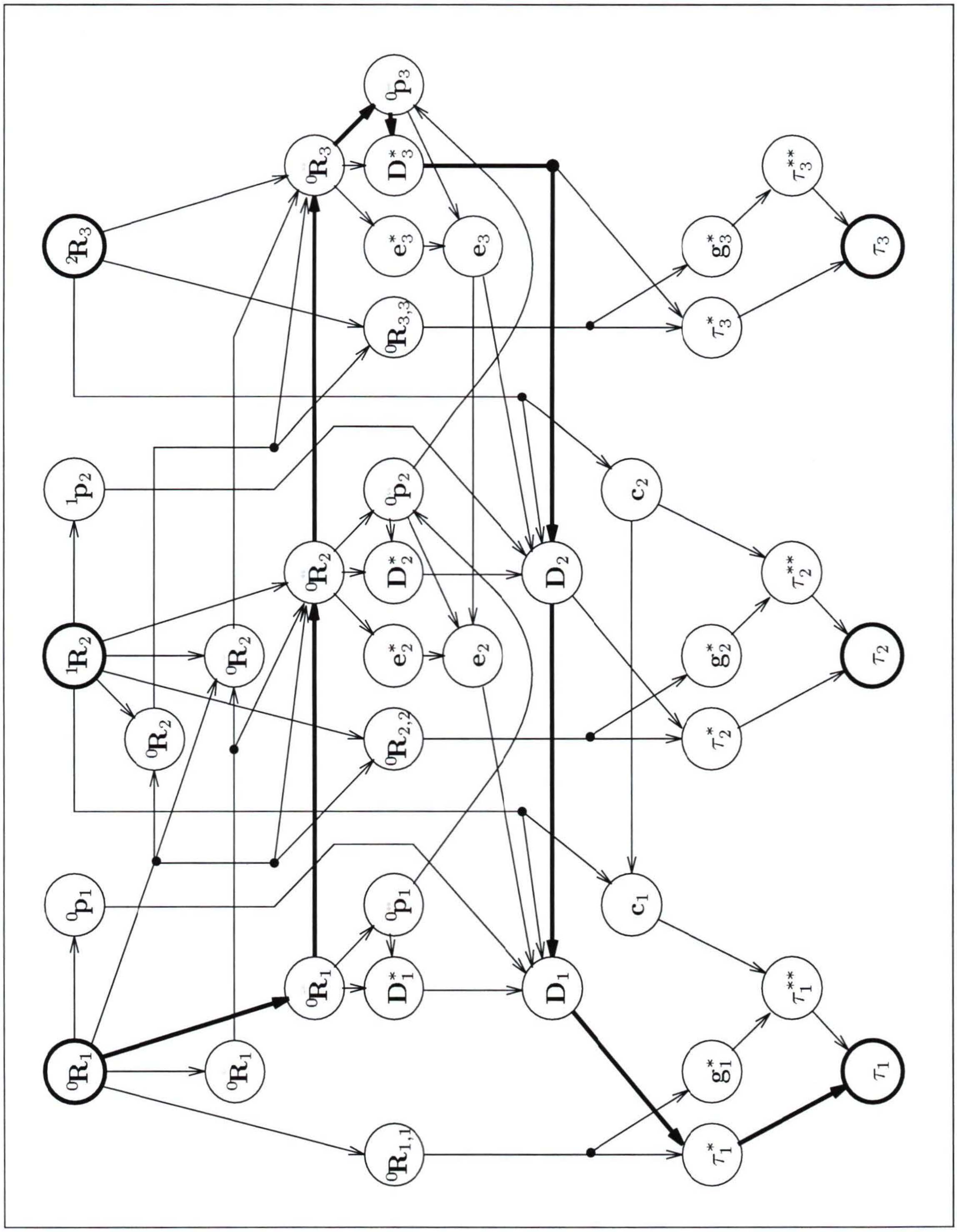

Figure 2.6. RECLE task graph for 3 links.

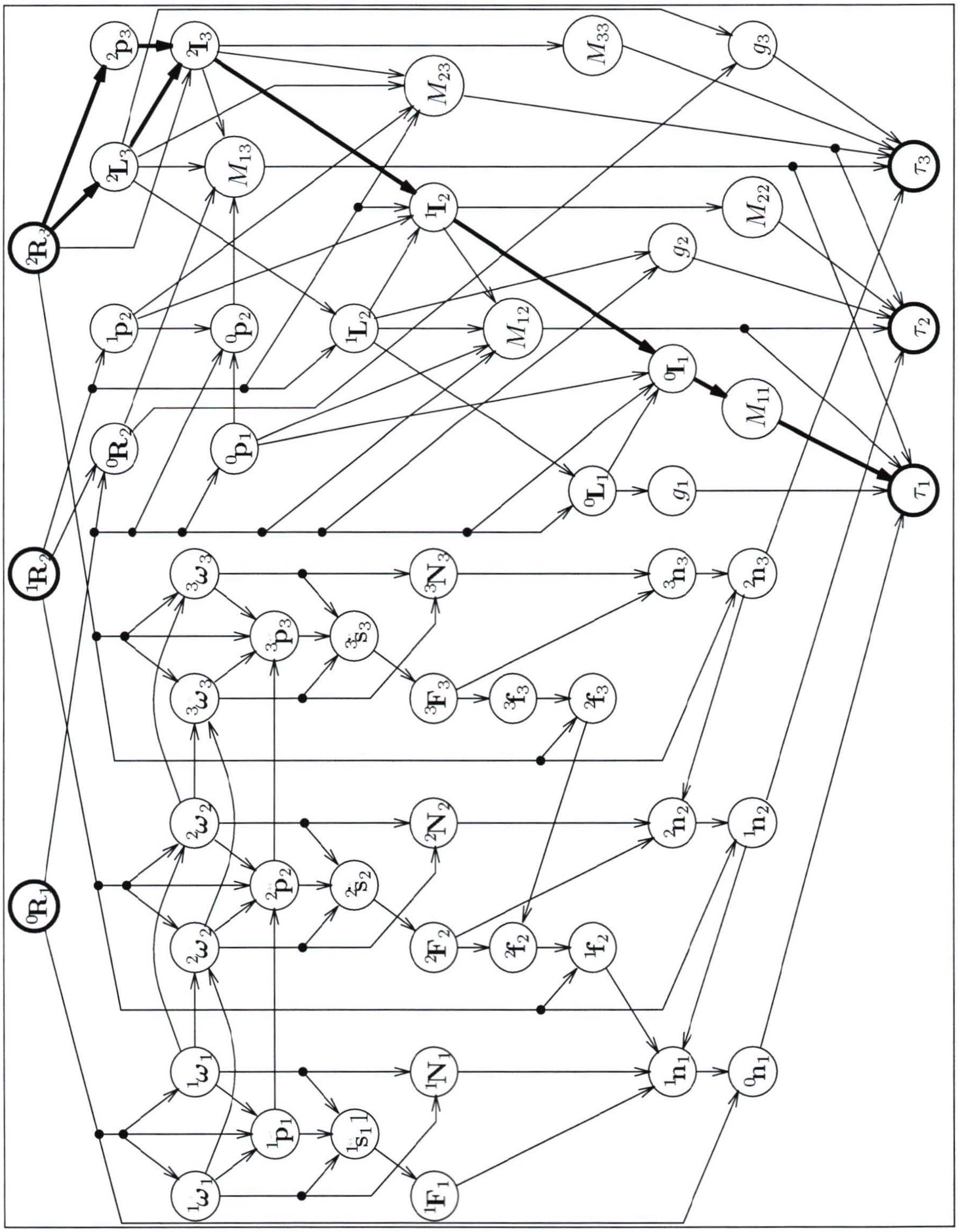

Figure 2.7: HYBRID task graph for 3 links

Algorithm	Critical Path Length	
RECNEb	$27\lceil\lg n\rceil\mathrm{M} + 24\lceil\lg n\rceil\mathrm{A} + 9\lceil\lg(n+1)\rceil\mathrm{A}$ $+ 116\mathrm{M} + 84\mathrm{A} + 2\mathrm{T}$	$n \geq 1$
RESNE	$(27\mathrm{M} + 21\mathrm{A})n + 52\mathrm{M} + 35\mathrm{A} + 2\mathrm{T} + \begin{cases} 0 \\ \mathrm{A} \end{cases}$	$n = 1$ $n > 1$
RECNE	$(48\mathrm{M} + 42\mathrm{A})n + 49\mathrm{M} + 32\mathrm{A} + 2\mathrm{T}$	$n \geq 1$
HYBRID	$(82\mathrm{M} + 82\mathrm{A})n + 37\mathrm{M} + 14\mathrm{A} + 2\mathrm{T}$	$n \geq 1$
RECLE	$(146\mathrm{M} + 117\mathrm{A})n - 24\mathrm{M} - 33\mathrm{A} + 2\mathrm{T}$	$n \geq 1$
CLASSLE	$(129n + 64\lceil\lg n\rceil + 70)\mathrm{M}$ $+ (101n + 48\lceil\lg n\rceil + 49)\mathrm{A} + 2\mathrm{T}$	$n = 1, 2, 3, 5, 9$
	$[n(n-1)/2 + 128n + 64\lceil\lg(n-1)\rceil + 70]\,\mathrm{M}$ $+ [n(n-1)/2 + 100n + 48\lceil\lg(n-1)\rceil + 45]\,\mathrm{A} + 2\mathrm{T}$	other $n \geq 1$

Table 2.7: General computational critical path lengths.

RECNE algorithm

LEE & CHANG [1986] show that the theoretical lower bound for the inverse dynamics of an n-link chain is $O(k_1\lceil n/p\rceil + k_2\lceil\lg p\rceil)$ on p processors. They therefore propose the RECNEb algorithm as a means of achieving that lower bound. Further algorithmic improvements must come from reducing the coefficients of the terms in Table 2.7. HASHIMOTO & KIMURA [1989] give $(33\mathrm{M}+27\mathrm{A})n+105\mathrm{M}+91\mathrm{A}$ for the computational critical path length for RESNE. The lower length shown in Table 2.7 results from eliminating the recursion for ${}^{i}\mathbf{g}$ and from using maximum parallelism. To simplify the critical path length calculation, some researchers (for example, HASHIMOTO & KIMURA [1989] and LEE & CHANG [1986]) sum the worst case execution times for each type of task (such as all of the ${}^{i-1}\mathbf{R}_i$'s). In effect, the tasks are grouped into larger tasks to simplify the task graph. This method does not, however, account for

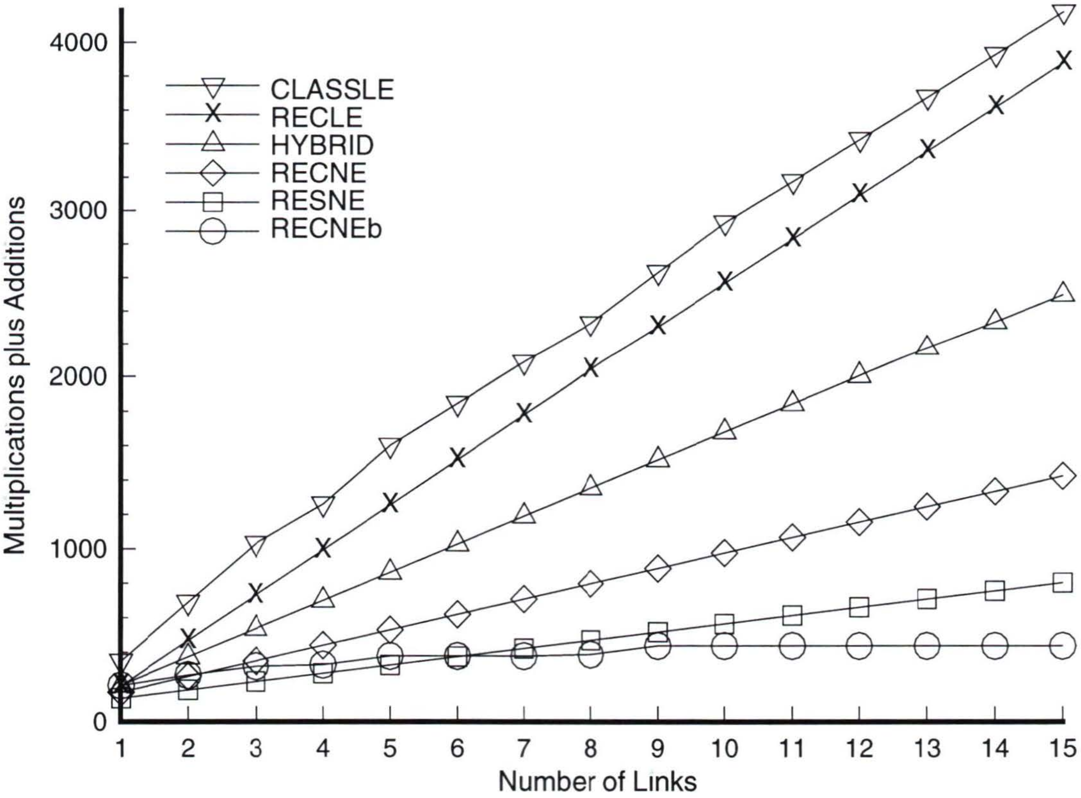

Figure 2.8: Analytical critical path lengths.

the fact that the execution of each type of task can overlap another. To obtain a consistent comparison, we have determined the computational critical path lengths for the common choice of tasks given in the algorithm descriptions of §2.2.

2.4 Performance Modelling

The next step in our analysis of the parallel inverse dynamics algorithms is to include the cost of communications. Indeed, if communication is as significant a part of the execution time as computation, then our previous analysis can no longer accurately predict the performance of the algorithms. The fastest execution of an algorithm on a message-passing network requires that a balance be struck between the amount of computation, which is useful, and the amount of communication, which is necessary. Other issues such as regularity of communications in time and space can significantly affect the performance of the algorithm.

Before we proceed to include the communication costs in the analysis, it is necessary to redefine the performance analysis problem. The analysis so far has measured the execution time of the algorithms by counting the number of floating point operations required along the longest path through the task graph. Accordingly, the natural choice for the unit of cost of communications is the number of floating point numbers which must be sent. Then, a new critical path through the task graphs which includes communications can be determined. With the aforementioned measure for communication costs, however, it becomes difficult to compare an algorithm with much computation and little communication to another with little computation and much communication. Indeed, the true performance measure is the total length of time required for an algorithm's execution. As well, in order to compare the relative significance of computation versus communication, we must be able to express each in the same units. Making this step in the performance analysis, however, reduces the generality of the comparisons because the time required for a given operation necessarily depends on the hardware on which it is executed. This in turn implies that we need to have a model of the hardware being used, a model which can account for all of the time spent by each processor. Therefore, the problem now becomes one of performance modelling and prediction which require accurate descriptions not only of the algorithms but also of the hardware being considered. The next section attempts to provide such a description.

2.4.1 The Hardware Model

The essential elements of the hardware model we will be using have already been encountered in the task graphs if we consider a task graph to represent maximum parallelism, with one processor per task. In our model, each processor executes a single sequential process. Processors must communicate with each other to exchange data, there is no shared memory. All communications are between pairs of processors only, there is no global communications mechanism. A pair of processors must synchronize to initiate communication. Until both processors are ready to communicate, one of them must remain idle and can execute no other tasks.

The actual transputer is capable of much more than what our model suggests. However, to improve performance or to facilitate performance analysis, many features were not utilized. For example, each transputer can execute concurrent processes using a built-in scheduler, but this makes it difficult to predict exactly when a task begins and ends execution. Communication buffers can also be used to allow a processor to continue execution while awaiting synchronization, or to receive a message before the receiving process is ready. They are not included in our model because of the significant overhead incurred when using buffers.

In fact, each transputer has four bit-serial communication links which can communicate independently of the CPU. For our purposes, we wish to know the following: 1) how long the CPU is occupied (blocked) when initiating a send, before the link hardware takes over, 2) how long it takes to send the message, and 3) how long the receiving CPU is occupied. Experiments were conducted in which the receiver waited idle, and the time to send the message was measured by the sender. This determines the amount of time for which the sending CPU is occupied. The receive time was measured in the same way and, for confirmation, ping tests were done to measure the round-trip delay as measured by the sending CPU. The results show that, over the range of message sizes required for the inverse dynamics algorithms (8-128 bytes), the measured send and receive times are equal. That is, both the sender and receiver are blocked for the same amount of time. The time blocked while sending x floating-point numbers (of 8 bytes each) is approximated quite well by $(13x + 7)/2$ microseconds (μs) (for the INMOS T800 transputer chip). Since this time increases linearly with the length of the message, it seems to imply that there is no overlapping of communication with computation. Therefore, for our purposes, we have assumed that the blocked time is the entire time required for transmission, and thus, both processors are blocked beginning at the same time.

To predict the computation time of each task, it was found that counting the number of multiplications, additions, and trigonometric function evaluations was not sufficient. Instead, these three numbers, together with the number of assignments, were necessary and sufficient. To measure the time taken for each of these operations, a set of six different size tasks were executed and timed to provide sample data points. The execution time of each task was assumed to be a linear function of each of the four operation times. A least-squares estimate of these times produced the values given in Table 2.8, where E is the time required for one assignment. Note that the time required for a communications task is comparable to the time required for a computation task, and hence is not negligible.

Operation	Time (μs)
M	1.37
A	0.85
T	42
E	1.55

Table 2.8: Computation times of the basic operations.

Although it would be convenient to use the operation times quoted by the hardware manufacturers, the method described above is more reliable, because it accounts for the programming style used throughout the tasks In the present implementation, all tasks were implemented without array indexing or control statements such as loops so that all computation could be accounted for Temporary variables are used to store the results of expressions contained within parentheses as given in the task descriptions This affects the computational cost only in terms of the number of assignments necessary The final code thus uses only the simplest programming constructs and is suitable for automatic computer generation which is used by IZAGUIRRE *et al* [1992] and VUKOBRATOVIĆ *et al* [1988]

With the above hardware model, the accuracy of the predicted execution times was excellent The code for the RESNE algorithm for a 3-link manipulator was implemented on a fully-connected network of three transputers according to the schedule produced by the scheduler (discussed in the next section) Tests were conducted using arbitrary numerical data for the state and physical parameters of the manipulator since it was found that zeroes within the computation significantly reduced the execution time The predicted total execution time on each of the three processors was 857, 1075, and 1090 μs (including communications), while the actual execution times were 853, 1077, and 1091 μs respectively

2.4.2 Scheduling

The hardware model presented in §2 4 1 allows us to include the effect of communications on an algorithm's performance, assuming a message-passing architecture such as the transputer This model differs in a significant way from the model which the task graphs seem to imply We have already determined that communication can not be overlapped with computation since the CPU is blocked by the entire communication

This also implies that communications can not be executed concurrently with each other. Even though the task graphs show edges fanning out in parallel from each node, the communications to and from each task must be executed sequentially. In a sense, communications can be considered as additional tasks which each processor must perform. Thus, even if we consider the maximum parallelism illustrated in the task graphs, in which each computational task is allotted its own processor, each processor must still execute multiple tasks because of the communications required. We have already determined that the communications tasks are roughly the same size as the computation tasks, and so are not negligible. The question then arises: In which order should the communications tasks be performed to minimize the algorithm's overall execution time?

This question can be generalized when we consider the parallel computer architectures typically used in robotics applications. Usually, maximum parallelism is not available, or even desired, so that the number of processors available is much less than the number of tasks. Also, the processors in a particular architecture have their own interconnection topology which may affect the cost of communications between tasks. Thus, to implement an algorithm on a given architecture we must first map the task graph to the processor graph, assigning each task to a processor. Second, the tasks assigned to each processor must be ordered to satisfy the precedence constraints of the task graph and to minimize the total execution time. Both of these problems together are called the *scheduling* problem. The inverse dynamics algorithms described in this chapter have the property that they can be *statically* scheduled, that is, the schedule is created once, off-line, before the algorithm is executed.

One of the first descriptions of a scheduling algorithm developed specifically for a robot inverse dynamics algorithm is presented in Luh & Lin [1982]. In their implementation, the processors are connected in a chain with shared memory between each pair of processors used for communications. Communication costs are therefore not included. First, they fix the mapping for the tasks so that all quantities pertaining to a link are calculated on the corresponding processor. Then, they show that the naive method of executing each task as soon as it is ready leads to sub-optimal schedules. Instead, they use a *variable branch-and-bound* algorithm to schedule RECNE for the Stanford arm onto six microprocessors, one per link.

The scheduling algorithm of Luh & Lin begins by creating an arbitrary schedule which satisfies the precedence constraints and the mapping constraint. At the time each task completes (and at the start of scheduling), the scheduler must choose the next task for execution. Each of the possible choices forms a branch in a decision tree. An arbitrary choice of task to schedule (branch to follow) is made at each decision point (node in the tree) until a feasible schedule is obtained. This technique is called a *depth-first* search since no other branches are explored until the search has proceeded all the way to the goal, in this case, a feasible schedule. The execution time, including idle time, of the first schedule obtained becomes an upper bound on the execution time of the optimal schedule. Then, the scheduler backtracks to the last preceding decision point and chooses a different task to be executed at that time. If the minimum possible execution time for this branch is greater than the current upper bound, then that branch can be pruned immediately. The minimum execution time for a branch is determined by adding the current time at the decision point to the total computation time (not including idle time) remaining on that branch. These forward and backward searches are alternated until the entire decision tree has been searched. Thus, the scheduler is guaranteed to find an optimal schedule since the tree is searched exhaustively, branches are only pruned if they can not possibly lead to an optimal schedule.

Several heuristics have been combined with the branch-and-bound scheduling technique to reduce the amount of searching. The heuristic provides a measure of the viability of each branch in the search tree and is used to prune the unlikely candidates for an optimal schedule. A depth first/implicit heuristic search is described by KASAHARA & NARITA [1985] which is based on a critical path/most immediate successors first heuristic. VUKOBRATOVIĆ *et al.* [1988] used a largest processing time first heuristic.

A more complex scheduling algorithm was presented by LEE & CHEN [1990] which mapped task graphs onto processor interconnection graphs. Heuristics based on the *level* of a task (the length of the longest path starting at that task) and the amount of communications required by that task were used to prioritize the list of ready tasks (those tasks whose predecessors have all completed). Further improvements were made by repeatedly generating schedules and using *simulated annealing* to approach the optimal schedule. To evaluate the success of their algorithms, Lee & Chen compared the schedules produced for simple problems to the optimal solutions found by an exhaustive search.

Although Luh & Lin were able to obtain optimal schedules, they also fixed the task mapping, did not consider communication costs, and performed an exhaustive search. In general, finding the optimal schedule is an NP-complete problem (there is no known polynomial-time algorithm), so an exhaustive search is not practical for any reasonably sized problems. In the next section we describe a scheduling algorithm which considers the full scheduling problem, including communication costs, and yet is simpler to implement than the algorithm of Lee & Chen. This scheduler was implemented and used for the performance comparisons of §2.5.1.

The ETF Scheduler

The Earliest Task First (ETF) scheduling algorithm, described by HWANG *et al* [1989], prioritizes the ready tasks according to the earliest time at which each task can execute, including communication delays Since the location of a task determines which communications are necessary, and hence affects a task's start time, the earliest start time is determined for all possible pairs of ready tasks and available processors Then, the task with the earliest possible start time is scheduled to execute on the corresponding processor

It should be noted that no backtracking is performed to improve a schedule Thus, ETF is a *greedy* algorithm which bases all decisions only on currently available information The objection raised by Luh and Lin, that executing tasks as soon as they are ready leads to sub-optimal schedules, shows that greedy algorithms do not always lead to optimal solutions It is easy to show, however, that once the mapping constraint used by Luh and Lin is relaxed, the ETF algorithm finds the optimal solution for the example they gave

In order to apply the ETF scheduler to the inverse dynamics algorithms, we first need to provide the scheduler with a software description of the task graphs which is expressive enough to capture their complexity and yet can easily generalize the graphs to arbitrary values of n For these reasons the task graphs are expressed as Prolog predicates where each predicate states a rule Together, the rules for a task graph specify the nodes and edges which exist for a given n Each `node` predicate also specifies the cost of a task in terms of matrix-vector operations, and the size of the task's output for determining communication costs These costs are evaluated as times in microseconds by using the hardware model of §2 4 1 and a processor interconnection graph, which is also expressed in Prolog Prolog's recursive nature makes it easy to determine properties of the task graphs such as the critical path and its length and experiments were performed to verify the analytical expressions for the critical path lengths presented in §2.3. Finally, the scheduler is also implemented in Prolog to ease the interface to the task graph descriptions.

Some modifications to the ETF scheduler were necessary to comply with the hardware model described earlier. Although communications delays are included when the scheduler determines the earliest execution time of a ready task, ETF assumes that communications can be performed concurrently with each other and with computation. The modified ETF scheduler determines which communications are necessary (if the two tasks are on different processors), and schedules those communications tasks along with the computation tasks. The send and receive task pairs are scheduled to execute at the same time on their respective processors to eliminate deadlocks and to help minimize idle time. The hardware model also requires that the earliest start time calculation be changed. For each available processor, the modified ETF algorithm sums the communications delays and adds the latest finishing time of a task's predecessors to determine the earliest time at which that task can begin execution. Although it may be possible to schedule some of the communications tasks earlier, and therefore not delay the start of the task, all of the delays are included in the sum because it is not known whether they will occur until the tasks are scheduled. Once the necessary communications tasks have been scheduled, the actual start time of the task may be earlier than predicted. However, the predicted earliest start time is still used to prioritize the ready tasks as is done in the original ETF algorithm.

It is certain that further improvements can be made to the ETF scheduling algorithm, such as adding the heuristics suggested by Lee & Chen [1990]. One possibility which has not been considered in any of the schedulers mentioned is to duplicate tasks. Rather than assigning a task to one processor and communicating its results, the scheduler may opt to assign that task to multiple processors to reduce the communication delays. This technique would have the greatest effect for tasks which require little computation but have much to communicate. Obviously, detecting such circumstances would demand a considerably more sophisticated scheduling algorithm. Eventually, however, the improvements gained from additional heuristics are no longer worth the effort needed to modify the scheduling algorithm. It is hoped, therefore, that the ETF algorithm with the modifications described here has achieved an acceptable balance between simplicity and performance.

2.5 Performance Analysis

This section compares the efficiency of the six inverse dynamics algorithms of §2.2 with respect to the hardware model presented earlier. The capability of the modified ETF scheduler is also illustrated by comparing theoretical performance bounds for the algorithms with the performance of the resulting schedules.

2.5.1 Scheduled Performance Results

Since the accuracy of the transputer hardware model has been established, we can use the scheduler for performance prediction instead of actually implementing the code. The scheduler can not, however, determine the optimal schedules and, therefore, the *best* (sub-optimal) schedules must be used to compare the algorithms. Note that such a comparison is not strictly a function of the properties of the algorithms themselves, but is necessarily affected by the optimality of the scheduler. Figure 2.9 shows the execution times as a function of n for the best schedules produced for each algorithm. The corresponding number of processors used to obtain these schedules is not optimal, since the schedules are not optimal, but is the best estimate using the modified ETF scheduler. Performance comparisons of the algorithms based on this graph can only be

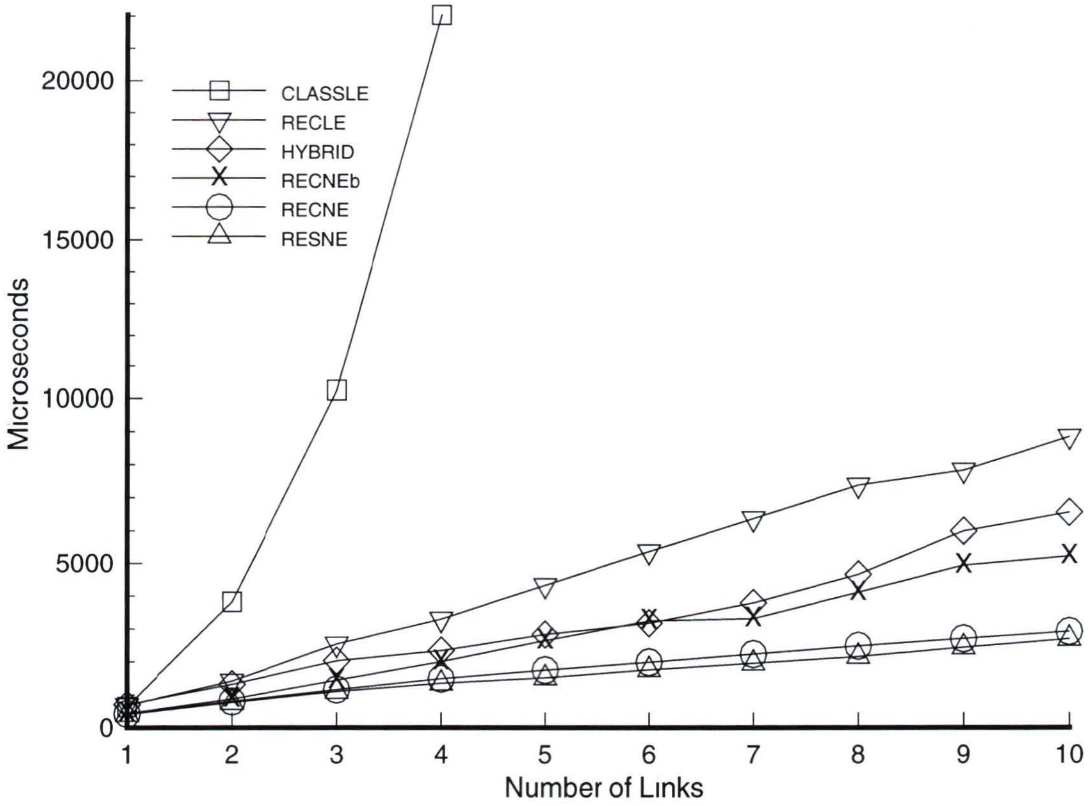

Figure 2.9: Best schedules for the inverse dynamics algorithms.

qualitative, since we do not have a useful measure of the optimality of the schedules. Note also that these performance results are a function of the hardware through the hardware model. Further improvements can be achieved using hardware specialized to each algorithm or level of parallelism which may affect the relative performance of the algorithms.

At least on a qualitative basis, however, the performance results confirm the analytical comparison of the task graphs presented in §2.3, with one notable exception. Recall from that section that the logarithmic complexity of the **RECNEb** critical path length made **RECNEb** the fastest algorithm when communication costs were not included. Figure 2.9 shows that communication costs have made RECNEb *more* costly than both RECNE and RESNE. The disadvantages of the recursive doubling technique are that it creates more tasks, and therefore more precedence constraints and more communications, to calculate similar quantities. These additional costs are more than enough to offset the increased parallelism offered by the recursive doubling structure. It should be noted that only RECNE and RECLE require a number of tasks which varies linearly with n, all of the other algorithms are quadratic, except CLASSLE, which is cubic. This may be a factor in explaining why RESNE is much closer in speed to RECNE, and why CLASSLE is much worse than RECLE, than what was predicted by the critical path lengths.

As mentioned earlier, it is not widely known that RESNE is more efficient than RECNE in parallel implementations. The analyses performed in this chapter provide solid evidence of the efficiency of RESNE for inverse dynamics calculations.

Many other researchers have published experimental execution times for the inverse dynamics problem. The algorithm chosen most often is RECNE which has been symbolically simplified and implemented using highly optimized code. The first parallel implementation of RECNE was reported by KASAHARA & NARITA [1985] which achieved an execution time of 5.37 ms for the 6 link Stanford arm (one prismatic joint) on a 7 processor shared-memory architecture. XIAO & CHENG [1991] assigned the trigonometric function evaluations to one transputer and all other computations to another transputer to achieve 0.98 ms for the 6 link Puma 560. Also for the Puma, RAJAGOPALAN *et al.* [1990] reported 2.8 ms on 3 transputers, and ZOMAYA & MORRIS [1990] reported $\approx 2ms$ (judging from a graph) on 4 transputers. We can compare these execution times to $2.007ms$ for 6 links on 5 transputers as predicted by the modified ETF scheduler. Finally, FIJANY & BEJCZY [1991] used special hardware to exploit parallelism at the matrix-vector level and achieved an execution time of

0.187ms for 6 links.

Using RESNE, Hashimoto *et al.* [1990] calculated the inverse dynamics for 3 links on 3 transputers in 0.464ms (giving a 0.66ms sampling period for the computed torque control technique). The execution time predicted by the modified ETF scheduler is 1.099ms for 3 links on 3 transputers. Zomaya [1992] executed CLASSLE for the Stanford arm on 8 transputers in 4.14 ms. The RECLE algorithm was implemented by Zalzala & Morris [1991] and executed in 2.46 ms for a 6 link problem on 4 transputers (compare with 5.365ms for 6 links on 7 transputers). All of these execution times are consistent with the current results. Although only Xiao & Cheng [1991] state explicitly the programming optimizations they employed, it is probable that other researchers (such as Hashimoto *et al.* [1990]) used similar techniques. Their results highlight the performance improvements which can be made by symbolic simplification and custom programming.

The six figures (2.11–2.16) show theoretical and scheduled execution times for each of the six inverse dynamics algorithms respectively. The lowest execution times obtained with the current scheduler are labelled Best. This term will be used throughout to denote the lowest, though sub-optimal, execution times predicted by the modified ETF scheduler for any possible number of processors (including maximum parallelism). In many parallel implementations of the inverse dynamics algorithms, the number of processors is chosen to be equal to the number of links. Although this provides a good rule of thumb, it does not always give the optimal performance. The results corresponding to the aforementioned rule are denoted by Macro and, as observed from the figures, are often not much worse than the Best results. Figure 2.10 shows the best number of processors as determined by the modified ETF scheduling algorithm as a function of the number of links n. Except for CLASSLE, the difference between the Best number of processors and the Macro number was usually not

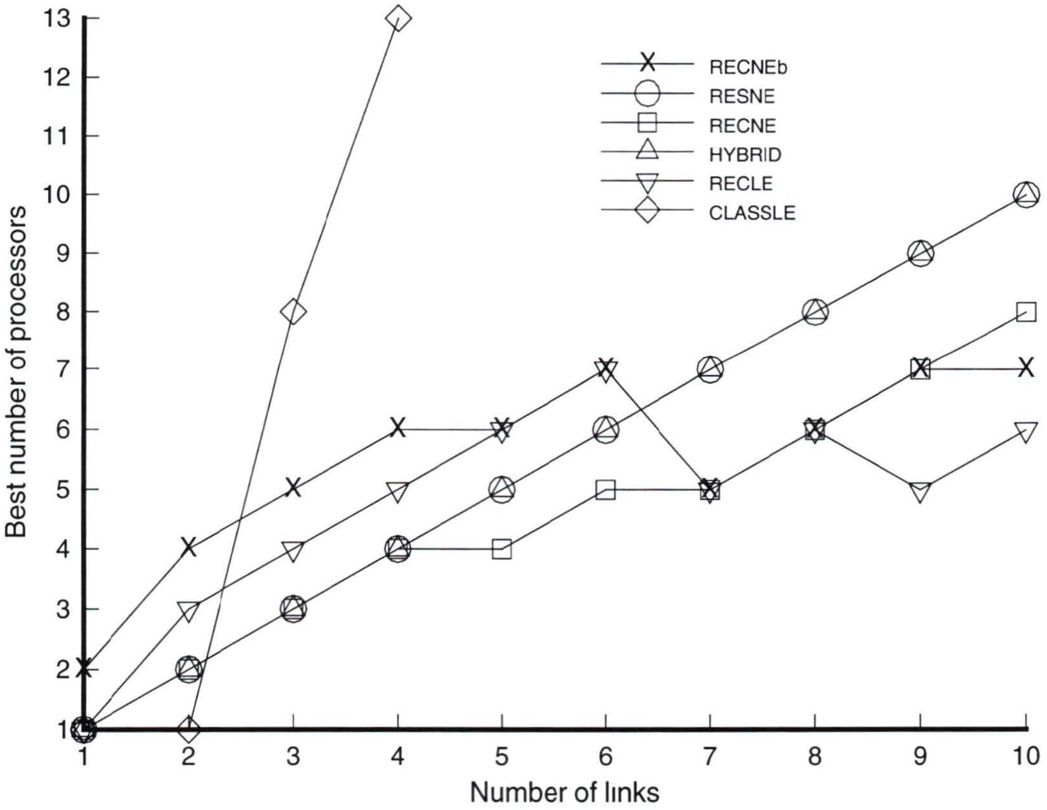

Figure 2.10. Best number of processors.

more than two. In fact, the differences may be sooner attributed to deficiencies in the scheduler than any properties of the algorithms. If enough processors are provided for maximum parallelism and each task is forced onto a distinct processor, then the scheduler predicts the total execution times labelled Maxpar. Finally, as a lower bound, the computational critical path length derived in §2.3 is also plotted. In Figure 2.14 for CLASSLE, not all of the curves continue to 10 links because we were unable to obtain the results due to insufficient memory on a Sun workstation.

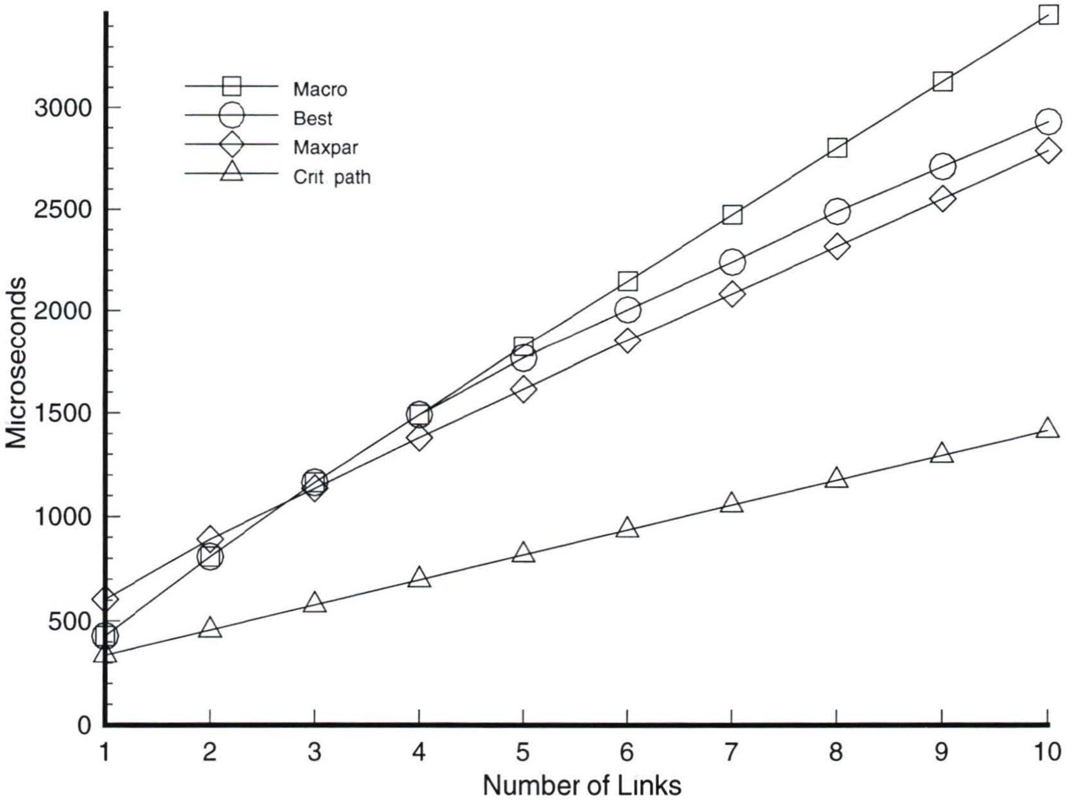

Figure 2.11. Theoretical and scheduled execution times for RECNE.

2.5.2 Performance Bounds

The previous section illustrated the kind of performance to be expected in an actual parallel implementation in which communication costs are included. However, these performance results are a function of the scheduler. To compare the inverse dynamics algorithms independently of the scheduler, while including communication costs, it is necessary to determine theoretical bounds on the execution time of each algorithm. Lower bounds on the execution time of the algorithms permit us to determine the potentially most efficient algorithm. Upper bounds are useful in real-time applications for which worst case performance is important.

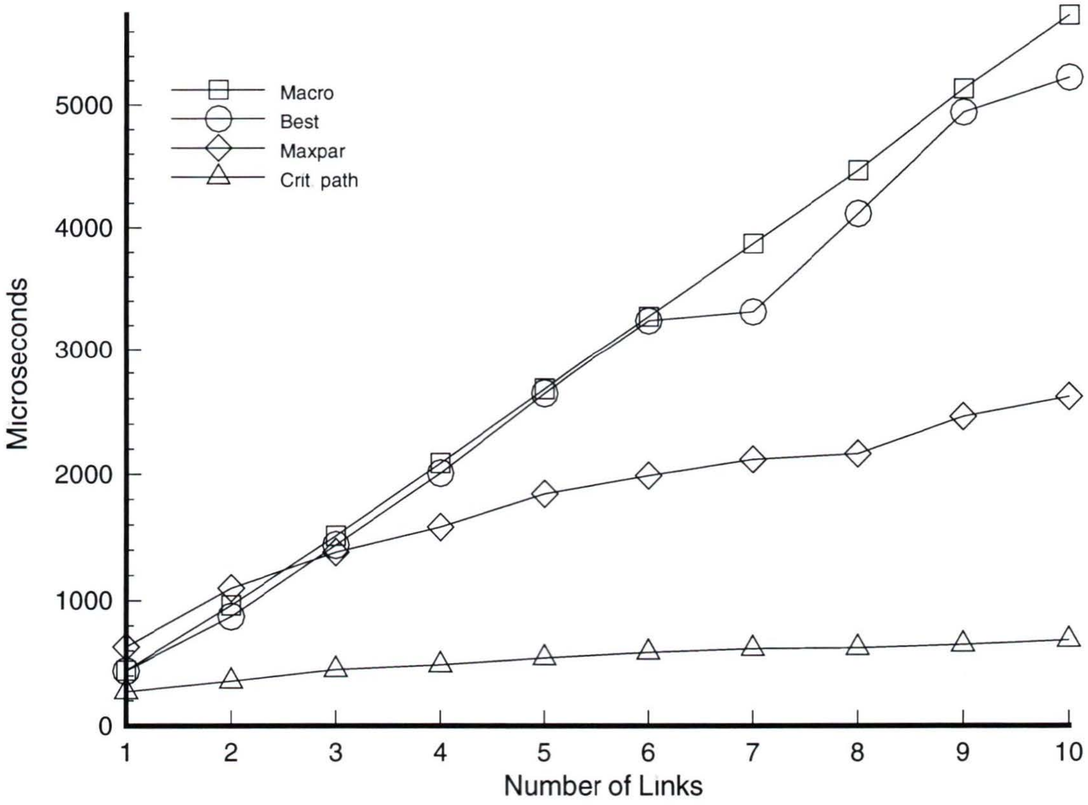

Figure 2 12 Theoretical and scheduled execution times for **RECNEb**

Since the communication tasks are roughly the same size as the computation tasks, the computational critical path length provides only a very weak lower bound, as can be seen from the figures, less than half of the best scheduled execution time Whether the optimal schedule is much closer to the lower bound is doubtful

Obtaining a meaningful upper bound on the execution time of the algorithms is also difficult To include the cost of communications in a theoretical upper bound, while maintaining the hardware model described earlier, it is necessary to derive a bound independent of the ordering of a task's communications The only bound which appears to do this is the sum of all of the computation times (the single processor

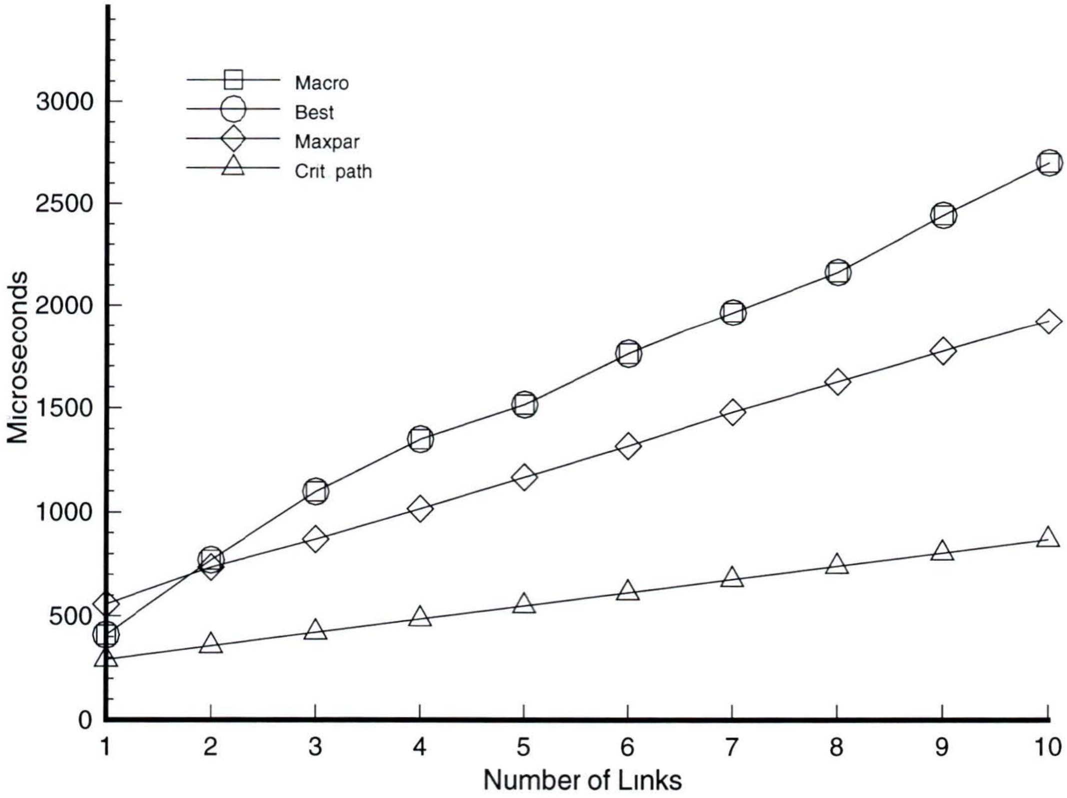

Figure 2.13 Theoretical and scheduled execution times for RESNE.

execution time) plus the sum of all of the communication times which exist in the maximum parallelism case. This bound is equal to the execution time of a schedule which executes all possible tasks (including unnecessary communication tasks) on a single processor. By comparison with the scheduled execution times, however, this bound is very weak, and therefore is not plotted. Rather than summing all possible communications costs, the bound on the amount of communication proposed by HWANG *et al.* [1989] considers the longest path through the task graph with all task costs set to zero. However, this longest path assumes that communications to and from each task can be performed in parallel, which violates the hardware model

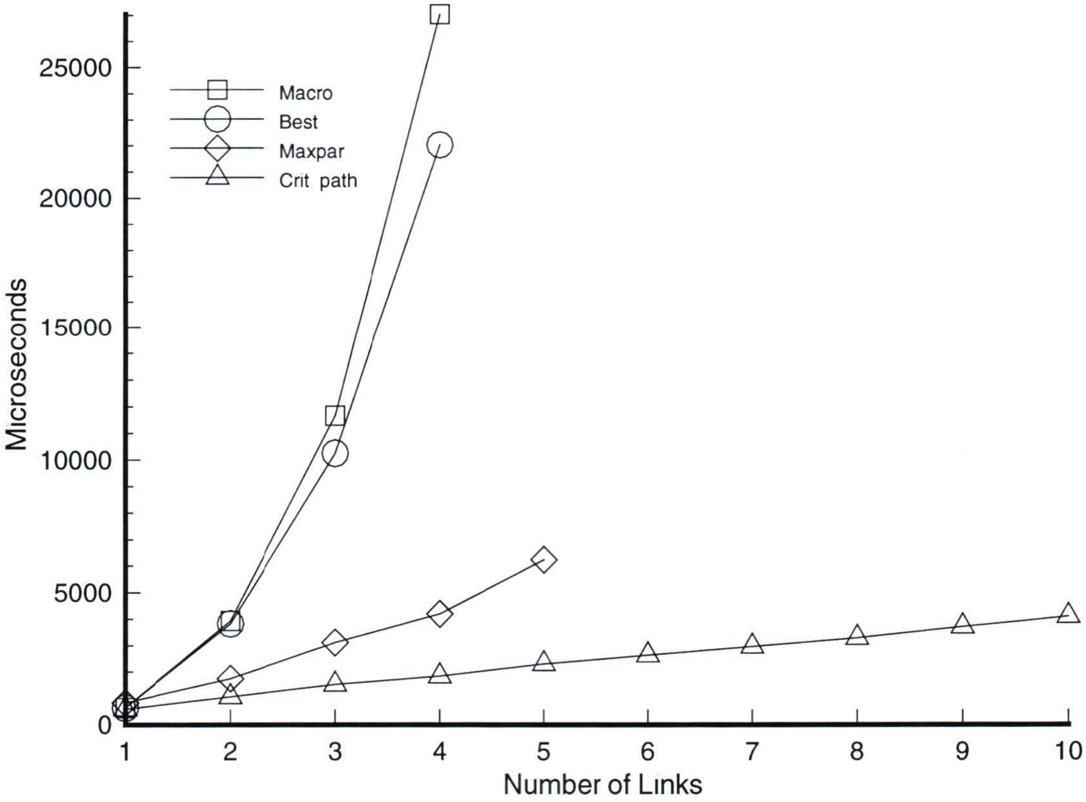

Figure 2.14: Theoretical and scheduled execution times for CLASSLE.

used here. It is worth noting that the maximum parallelism case alone is no aid in deriving an upper bound because although the *number* of communications tasks is maximized, they are executed in parallel, as are the computations. Thus, the total execution time is not maximized.

2.5.3 Optimality of the Scheduler

Since there are no useful performance bounds available, parallel performance comparison of the inverse dynamics algorithms remains dependent on the schedules used

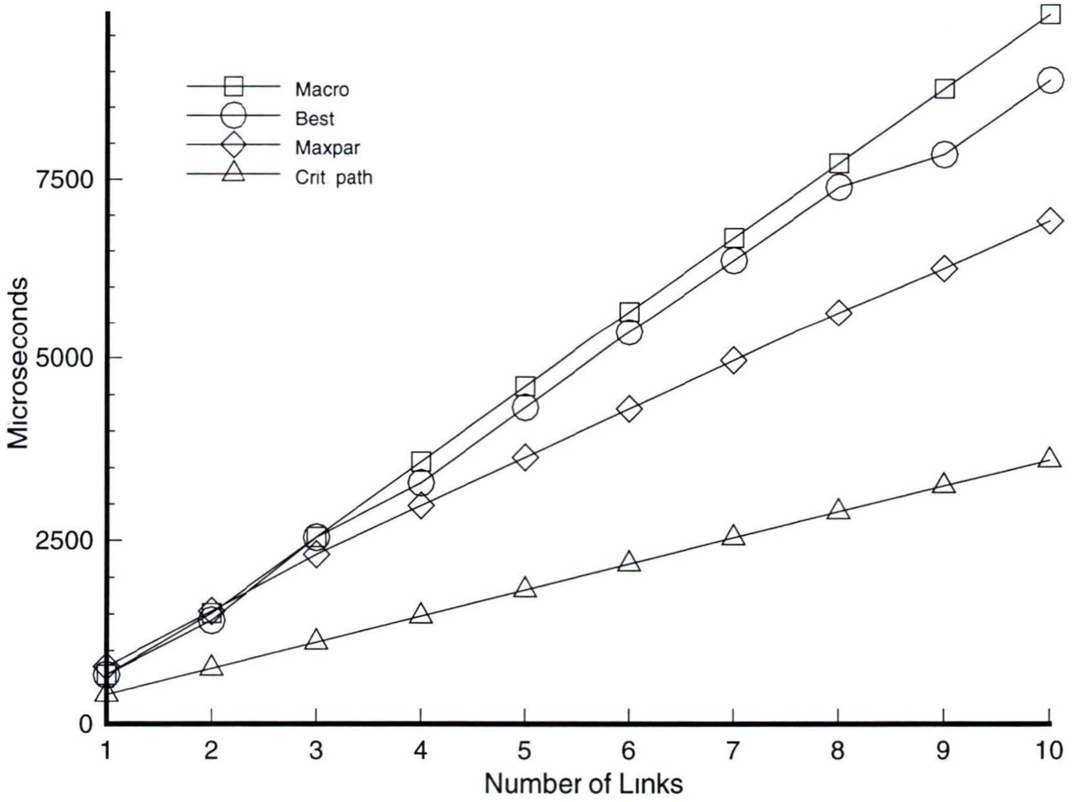

Figure 2.15: Theoretical and scheduled execution times for RECLE.

to implement them. It is therefore important to choose an efficient scheduling algorithm. Optimal schedules provide the best possible performance for an algorithm and allow the relative efficiencies of the algorithms to be observed directly, without including inefficiencies in the schedule. Since we do not have a useful lower bound on the execution times of the algorithms, it is difficult to evaluate the optimality of the modified ETF scheduling algorithm. Nevertheless, some progress can be made in that direction.

Each graph (Figures 2.11–2.16) shows the *best* schedule produced by the scheduler and the maximum parallelism schedule obtained by forcing each task onto a separate

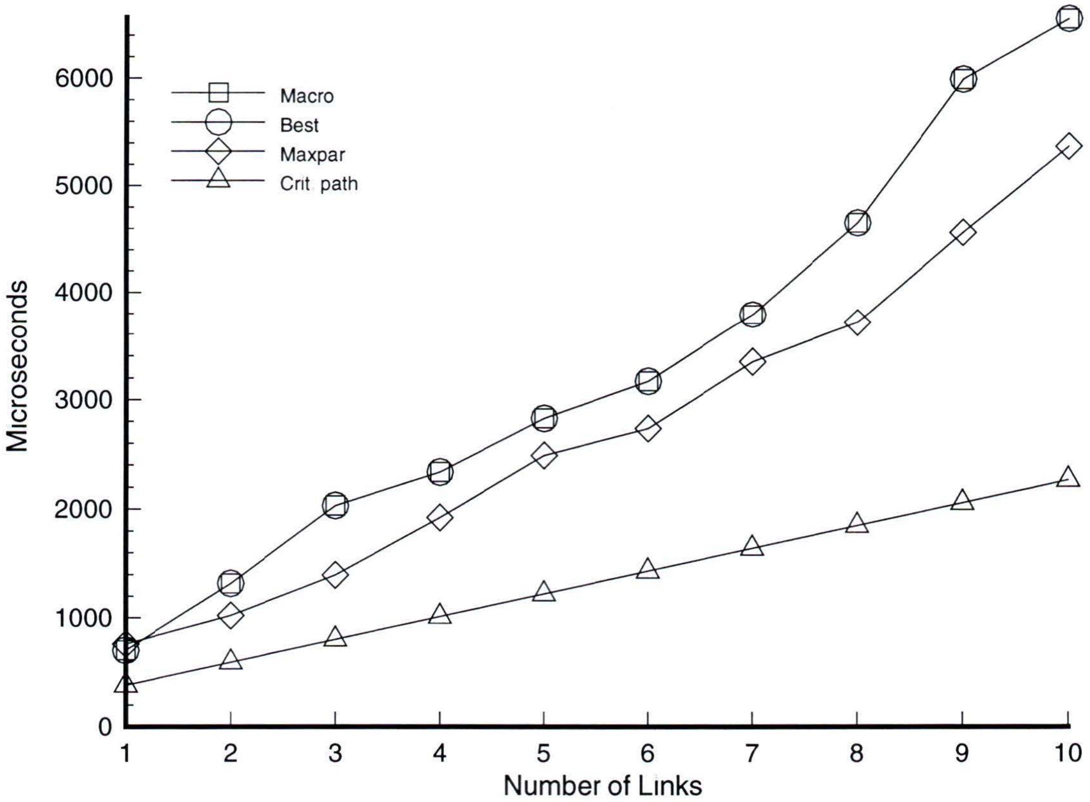

Figure 2 16 Theoretical and scheduled execution times for HYBRID

processor In each case, however, the best schedule is worse than the maximum parallelism schedule, illustrating the fact that the scheduler was never able to attain the performance of the maximum parallelism schedule even if sufficient processors were available When using the scheduler, we usually observed that the total execution time decreased until a certain number of processors were available, after which the execution time did not change because the scheduler chose not to use the additional processors This in turn means that the scheduler was never able to attain the maximum parallelism performance on its own In some cases, the scheduler used the additional processors but predicted a greater execution time than the previous minimum. Both of these behaviours are indications that the scheduling algorithm can be improved by incorporating a backtracking mechanism to retain the currently best schedule while broadening the search for the optimal schedule. A worthwhile extension of this work would therefore be to investigate the scheduler proposed by LEE & CHEN [1990].

Since the maximum parallelism schedules require the maximum number of communications tasks, it is reasonable to assume that they are not optimal. In fact, the maximum parallelism performance can be used as an heuristic to decide how to improve performance. In KWAN *et al.* [1990], the maximum parallelism performance is used when searching for the optimal number of processors for a given algorithm (an exhaustive search as we have done is not always practical). In particular, Kwan *et al.* begin with upper and lower bounds on the optimal number of processors and use a bisection search. A schedule is created for the number of processors midway between the bounds. If the scheduled execution time is greater than the maximum parallelism execution time, then performance can be improved by adding more processors to the current value. This current number of processors then replaces the lower bound and a new schedule is created. Otherwise, the current number of processors becomes the new upper bound and the process is repeated. Once the upper and lower bounds are equal, it may still be possible to improve performance by reducing the *granularity*, or the size of the tasks, thereby increasing the potential parallelism present and reducing the critical path length.

2.6 Summary of the Inverse Dynamics Algorithms

This chapter provides evidence for the relative merit of each of the inverse dynamics algorithms in a real multicomputer implementation. The analytical performance comparison based on the computational critical path lengths is the first detailed comparison of the theoretical parallel efficiencies of the algorithms. It establishes a common basis for comparison which is independent of any parallel computer hardware characteristics. Further analysis includes communication costs and shows that the scheduler can accurately predict the total execution times. With the exception of **RECNEb**, the predicted execution times qualitatively agree with the analytical computational critical path length comparison, in the sense that **RESNE** is the most efficient inverse dynamics algorithm for our application. The **RECNEb** algorithm is a clear example of the significance of communication costs and the limitations of performance comparisons which ignore these costs.

Further conclusions, however, are subject to our confidence in the optimality of the scheduler. For example, the optimal number of processors obviously depends on the optimal schedule, as would an analysis of the parallel speedup and efficiency (speedup per number of processors) of the algorithms. It would also be important to consider more realistic processor interconnection topologies. All of the results presented in this chapter have been for schedules executing on a fully-connected topology. Since each transputer has only four serial communications links, a fully-connected topology is only possible for up to five processors (actually only four, since one processor must connect to the host). Other hardware may allow more local connections, but there will still be a physical limit. Varying topologies introduce an additional factor which complicates the scheduling process. Investigation of this effect, and of more sophisticated scheduling algorithms, would be natural extensions to the present work.

Chapter 3

A Parallel Simulation Dynamics Algorithm

The proposal and investigation of new robot simulation dynamics algorithms remains an active area of research [FIJANY & BEJCZY, 1989; WONG & LAWRENCE, 1992]. Since robot simulation dynamics algorithms are more complex than inverse dynamics algorithms, a detailed parallel performance analysis of the type conducted in the last chapter is not practical. The performance of these algorithms can only be gauged either through prediction based on theoretical models or, preferably, through analysis of an actual parallel implementation. In particular, this chapter considers a novel algorithm of SHARF [1990] designed specifically for parallel implementation. An implementation and performance analysis of the algorithm is conducted to validate preliminary performance models based on a serial implementation. The implementation is also important as a demonstration of the viability of the algorithm.

The simulation dynamics algorithm described in this chapter was specifically designed to exploit parallelism at a level unavailable to other parallel simulation dynamics algorithms. In particular, an intuitive method of parallelizing robot dynamics algorithms is to assign the computations pertaining to each body to the corresponding processor in a chain topology. This is referred to as the *macroparallel* level of computation. The clear advantage of this level of parallelism is that the complex scheduling process is eliminated. Section 3.1 describes the macroparallel algorithm and the properties which make it suitable for such an implementation.

As mentioned in the introduction to this thesis, the performance of a parallel algorithm may be highly dependent on the chosen computer architecture. Section 3.2 discusses the architectural considerations which were made during the algorithm design and its parallel implementation. The subsequent section illustrates how the choice of communication methods can significantly affect the performance of the algorithm on a network of transputers.

The performance of the macroparallel simulation dynamics algorithm is not analyzed in the same detail as the inverse dynamics algorithms of the previous chapter. However, the same concepts apply and are used in the analysis. For example, the algorithm's simple choice of tasks allows us to determine the computational critical path length simply by measuring the execution times of tasks in the serial implementation. The analysis is then extended to include the cost of communications. Performance models are presented in §3.5 which use information obtained from the serial implementation and the actual parallel implementation, as well as theoretical properties of the algorithm. Finally, the models are discussed and compared to the measured parallel performance.

3.1 Macroparallel Algorithm

This section describes the macroparallel simulation dynamics algorithm and highlights those steps which are relevant to the parallel performance. In our presentation, we follow the notation used in SHARF [1990] which is somewhat different from that used in the previous chapter. The complete derivation of the algorithm can also be found in the same reference.

3.1.1 Solution for Constraint Forces

In the simulation dynamics problem, the motion equations derived using the Newton-Euler formulation for an open chain of rigid bodies contain two sets of unknowns: the accelerations of the bodies, and the interbody constraint forces. In most simulation dynamics algorithms, the constraint forces are eliminated in the formulation to obtain a reduced, independent set of equations for the accelerations. These equations form a system of ordinary differential equations which can be integrated to obtain the positions and velocities of the bodies — the state of the system at the next time step. In the present formulation, however, the constraint forces are explicitly determined by solving a linear system of equations of the form

$$\mathbf{A}\boldsymbol{f}_{\square} = \mathbf{b} \tag{3.1}$$

where $\boldsymbol{f}_{\square}$ represents the assembly of generalized constraint forces $\boldsymbol{f}_{n,\square}$, starting with the base body $\mathcal{B}_0$ to the tip body $\mathcal{B}_N$.

$$\boldsymbol{f}_{\square} = \mathrm{col}\{\boldsymbol{f}_{0,\square}, \boldsymbol{f}_{1,\square}, \ldots, \boldsymbol{f}_{N,\square}\} \tag{3.2}$$

In the remainder of this chapter, the subscript n denotes the quantity corresponding to body $\mathcal{B}_n$ and thus will range from 0 to N. The matrix $\mathbf{A}$ in (3.1) is symmetric positive-definite and has a *block tridiagonal* structure given by

$$\mathbf{A} = \begin{bmatrix} \ddots & & & & \mathbf{0} \\ & & & & \\ & -\mathbf{L}_n & \mathbf{D}_n & -\mathbf{U}_n & \\ & & & & \\ \mathbf{0} & & & & \ddots \end{bmatrix} \tag{3.3}$$

where

$$\begin{aligned} \mathbf{L}_n &= \mathcal{Q}_n^T \mathcal{T}_{n,n-1} \mathcal{M}_{n-1}^{-1} \mathcal{Q}_{n-1} \\ \mathbf{D}_n &= \mathcal{Q}_n^T \left(\mathcal{M}_n^{-1} + \mathcal{T}_{n,n-1} \mathcal{M}_{n-1}^{-1} \mathcal{T}_{n,n-1}^T \right) \mathcal{Q}_n \\ \mathbf{U}_n &= \mathcal{Q}_n^T \mathcal{M}_n^{-1} \mathcal{T}_{n+1,n}^T \mathcal{Q}_{n+1} = \mathbf{L}_{n+1}^T \end{aligned} \tag{3.4}$$

In (3.4), $\mathcal{M}_n$ is a *constant* 6×6 generalized mass matrix, $\mathcal{Q}_n$ is the constraint projection matrix of dimension $6 \times p_n$, where p_n denotes the number of constrained degrees of freedom, and $\mathcal{T}_{n,n-1}$ is a generalized transformation matrix given by

$$\mathcal{T}_{n,n-1} = \begin{bmatrix} {}^{n}\mathbf{R}_{n-1} & -{}^{n}\mathbf{R}_{n-1}\tilde{\mathbf{r}}_{n-1,n} \\ \mathbf{0} & {}^{n}\mathbf{R}_{n-1} \end{bmatrix} \tag{3.5}$$

Following the convention used in SHARF [1990], the body-fixed coordinate frames are located at the proximal ends of the links. Hence, the position of coordinate frame n with respect to coordinate frame $n-1$, given by $\mathbf{r}_{n-1,n}$, is expressed in coordinate frame $n-1$ and is constant. Note that the block tridiagonal structure of $\mathbf{A}$ corresponds to the physical structure of the open chain, constraint forces exist only between consecutive bodies in the chain.

The right-hand side of (3.1) is defined as

$$\mathbf{b} = \mathrm{col}\{\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_N\} \tag{3.6}$$

where

$$\begin{aligned}\mathbf{b}_n &= \mathcal{Q}_n^T \Big[\mathcal{T}_{n,n-1}\mathcal{M}_{n-1}^{-1}\left(\mathcal{P}_{n-1}\boldsymbol{f}_{n-1,\mathrm{c}} - \mathcal{T}_{n,n-1}^T\mathcal{P}_n\boldsymbol{f}_{n,\mathrm{c}} + \boldsymbol{f}_{n-1,\mathrm{ext}} + \boldsymbol{f}_{n-1,\mathrm{I}}\right) \\ &\quad + \mathcal{M}_n^{-1}\left(\mathcal{T}_{n+1,n}^T\mathcal{P}_{n+1}\boldsymbol{f}_{n+1,\mathrm{c}} - \mathcal{P}_n\boldsymbol{f}_{n,\mathrm{c}} - \boldsymbol{f}_{n,\mathrm{ext}} - \boldsymbol{f}_{n\mathrm{I}}\right) \\ &\quad + \mathcal{T}_{n,n-1}\boldsymbol{v}_{n-1} + \mathcal{P}_n\boldsymbol{v}_{n\gamma}\Big]\end{aligned} \tag{3.7}$$

In the above, we have the $6 \times (6 - p_n)$ projection matrices $\mathcal{P}_n$, which project the control forces $\boldsymbol{f}_{n,\mathrm{c}}$ onto the body-fixed orthogonal axes. Also, $\boldsymbol{f}_{n,\mathrm{ext}}$ denote the external forces acting on $\mathcal{B}_n$, and $\boldsymbol{f}_{n\mathrm{I}}$ denotes the nonlinear inertial forces. Finally, $\boldsymbol{v}_n$ are the *generalized* absolute velocities, and $\boldsymbol{v}_{n\gamma}$ are the *free* joint rates corresponding to the joint degrees of freedom. For dynamics simulation, the control and external forces are usually supplied as inputs. To simplify the parallel performance analysis, we assume them to be dependent on the state of $\mathcal{B}_n$ only. Since the system state is distributed throughout the entire chain of processors, this assumption aids a macroparallel implementation by reducing the amount of communication necessary. Each of the inertial forces $\boldsymbol{f}_{n\mathrm{I}}$ is a nonlinear function of the generalized absolute velocity $\boldsymbol{v}_n$ given by

$$\boldsymbol{f}_{n\mathrm{I}} = \tilde{\boldsymbol{v}}_n^T\mathcal{M}_n\boldsymbol{v}_n \tag{3.8}$$

These velocities are obtained through a forward recursion from the base to the tip given by

$$\boldsymbol{v}_n = \mathcal{T}_{n,n-1}\boldsymbol{v}_{n-1} + \mathcal{P}_n\boldsymbol{v}_{n\gamma} \tag{3.9}$$

We note that $\mathcal{P}_n$, $\mathcal{Q}_n$ and $\mathcal{T}_{n,n-1}$ depend only on the configuration of $\mathcal{B}_n$, which is local state information. Therefore, the blocks of $\mathbf{A}$ as defined by (3.4) are independent of each other for $n = 0, \ldots, N$. For example, $\mathbf{D}_1$ does not depend on $\mathbf{D}_0$ or $\mathbf{D}_2$ or any other $\mathbf{D}_n$ and similarly for $\mathbf{L}_n$ and $\mathbf{U}_n$. These blocks, as well as the blocks of $\mathbf{b}$ (3.7), can therefore be calculated simultaneously.

The special properties of the system matrix **A** motivated Sharf to consider *iterative* methods to solve for $\boldsymbol{f}_{n,\square}$ In particular, the sparseness of **A** makes these methods computationally efficient Moreover, because the matrix **A** is block tridiagonal, there is a variety of well-established iterative schemes that can be used to solve the system (3 1) [HAGEMAN & YOUNG, 1981] With regard to parallelism, it is through employment of iterative methods that a macroparallel solution for the $\boldsymbol{f}_{n,\square}$ is feasible

A thorough investigation of several iterative techniques has been conducted by SHARF [1990] and it was concluded that the polynomially preconditioned conjugate gradient methods are computationally most efficient for solving the linear system (3 1) The method chosen, denoted by J-PPCG(2), is a composite iteration that involves an inner and an outer loop The former comprises two steps of the Jacobi iteration and essentially produces the preconditioning matrix **M** For this particular scheme, **M** is the approximate polynomial inverse of **A** truncated to two terms (see [SHARF, 1990] for details) This preconditioner is used in the conjugate gradient iteration of the outer loop Table 3 1 lists the parallel algorithm for J-PPCG(2) The notation $(\mathbf{u}, \mathbf{v})$ indicates the standard inner product of the vectors **u** and **v**

3.1.2 Motion Equations for $\mathcal{B}_n$

Once the constraint forces are known, the accelerations of the bodies are calculated using the dynamics equations for each body in the chain These are

$$\boldsymbol{v}_n = \boldsymbol{\mathcal{M}}_n^{-1} (\boldsymbol{f}_{n\mathrm{T}} + \boldsymbol{f}_{n\mathrm{I}}) \qquad (3\ 10)$$

```
Initialize: $i = 1$
    Do in Parallel $n = 0, \ldots, N$
        $\mathbf{r}_n^{(0)} = \mathbf{b}_n + \mathbf{L}_n \boldsymbol{f}_{n-1,\square}^{(0)} - \mathbf{D}_n \boldsymbol{f}_{n,\square}^{(0)} + \mathbf{U}_n \boldsymbol{f}_{n+1,\square}^{(0)}$
        $\mathbf{M}_n \mathbf{z}_n^{(0)} = \mathbf{r}_n^{(0)}$
        $\mathbf{p}_n^{(0)} = \mathbf{z}_n^{(0)}$
        $a_n^{(0)} = \left(\mathbf{z}_n^{(0)}, \mathbf{r}_n^{(0)}\right)$
Do until convergence (outer loop):
    Do in Parallel $n = 0, \ldots, N$
        $\mathbf{w}_n^{(i-1)} = -\mathbf{L}_n \mathbf{p}_{n-1}^{(i-1)} + \mathbf{D}_n \mathbf{p}_n^{(i-1)} - \mathbf{U}_n \mathbf{p}_{n+1}^{(i-1)}$
        $d_n^{(i-1)} = \left(\mathbf{p}_n^{(i-1)}, \mathbf{w}_n^{(i-1)}\right)$
    $\alpha = \frac{\sum_{n=0}^{N} a_n^{(i-1)}}{\sum_{n=0}^{N} d_n^{(i-1)}}$
    Do in Parallel $n = 0, \ldots, N$
        $\boldsymbol{f}_{n,\square}^{(i)} = \boldsymbol{f}_{n,\square}^{(i-1)} + \alpha \mathbf{p}_n^{(i-1)}$
        Local convergence check:
        $\boldsymbol{\Delta}_n^{(i)} = \boldsymbol{f}_{n,\square}^{(i)} - \boldsymbol{f}_{n,\square}^{(i-1)}$
        if $\frac{(\boldsymbol{\Delta}_n^{(i)}, \boldsymbol{\Delta}_n^{(i)})^{1/2}}{(\boldsymbol{f}_{n,\square}^{(i)}, \boldsymbol{f}_{n,\square}^{(i)})^{1/2}} < \varepsilon_n$, set $\boldsymbol{f}_{n,\square} = \boldsymbol{f}_{n,\square}^{(i)}$ and terminate iteration for $\boldsymbol{f}_{n,\square}$.
        $\mathbf{r}_n^{(i)} = \mathbf{r}_n^{(i-1)} - \alpha \mathbf{w}_n^{(i-1)}$
        Solve $\mathbf{M}_n \mathbf{z}_n^{(i)} = \mathbf{r}_n^{(i)}$ for $\mathbf{z}_n^{(i)}$ using the inner-loop iteration.
        $a_n^{(i)} = \left(\mathbf{z}_n^{(i)}, \mathbf{r}_n^{(i)}\right)$
    $\beta = \frac{\sum_{n=0}^{N} a_n^{(i)}}{\sum_{n=0}^{N} a_n^{(i-1)}}$
    Do in Parallel $n = 0, \ldots, N$
        $\mathbf{p}_n^{(i)} = \mathbf{z}_n^{(i)} + \beta \mathbf{p}_n^{(i-1)}$
    $i = i + 1$
```

Table 3.1: Parallel J-PPCG(2) Algorithm for Step III (ii).

where $\boldsymbol{f}_{n\mathrm{T}}$ denotes the total external generalized force acting on $\mathcal{B}_n$, and is determined by

$$\begin{aligned} \boldsymbol{f}_{n\mathrm{T}} &= \boldsymbol{\mathcal{T}}^T_{n+1,n}\left(-\boldsymbol{\mathcal{P}}_{n+1}\boldsymbol{f}_{n+1,\mathrm{c}} - \boldsymbol{\mathcal{Q}}_{n+1}\boldsymbol{f}_{n+1,\square}\right) \\ &\quad +\boldsymbol{\mathcal{P}}_n\boldsymbol{f}_{n,\mathrm{c}} + \boldsymbol{\mathcal{Q}}_n\boldsymbol{f}_{n,\square} + \boldsymbol{f}_{n,\mathrm{ext}} \end{aligned} \tag{3.11}$$

To complete the procedure, we need to specify the independent joint accelerations $\boldsymbol{v}_{n\gamma}$, as they are the variables which are integrated to determine the trajectory of the chain. They can be obtained by differentiating the recursive velocity kinematics equation (3.9) and solving to get

$$\boldsymbol{v}_{n\gamma} = \left(\boldsymbol{\mathcal{P}}^T_n\boldsymbol{\mathcal{P}}_n\right)^{-1}\boldsymbol{\mathcal{P}}^T_n\left(\boldsymbol{v}_n - \boldsymbol{\mathcal{T}}_{n,n-1}\boldsymbol{v}_{n-1} - \boldsymbol{\mathcal{T}}_{n,n-1}\boldsymbol{v}_{n-1} - \boldsymbol{\mathcal{P}}_n\boldsymbol{v}_{n\gamma}\right) \tag{3.12}$$

It can now be seen that, with the constraint forces known, the dynamics equations (3.10) in conjunction with (3.11) can be evaluated for all $\boldsymbol{v}_n$ in parallel. Using these in Equation (3.12), we can also calculate the joint accelerations $\boldsymbol{v}_{n\gamma}$ for $n = 0, \ldots, N$ in parallel.

3.1.3 Summary of the Macroparallel Algorithm

We can now summarize the macroparallel solution algorithm for simulation dynamics of a rigid-body chain. Since the algorithm is executed at every time step during the integration process, the state of the system (*i.e.*, joint displacements and velocities) is assumed to be known and the desired output is joint accelerations. The algorithm is divided into five principal computation stages, corresponding to the logical elements of the procedure. As was mentioned earlier, we dedicate one processor to each body in the chain. Then, the computation of each stage can be carried out as listed in Table 3.2.

```
I Initialization
    Do in Parallel $n = 0, \ldots, N$
        Update $\mathcal{T}_{n,n-1}$, $\mathcal{P}_n$ and $\mathcal{Q}_n$, $\boldsymbol{f}_{n,\mathrm{c}}$, $\boldsymbol{f}_{n,\mathrm{ext}}$
II Direct Velocity Kinematics
    Do Recursively $n = 1$ to $N$
        Solve for $\boldsymbol{v}_n$ from (3.9)
III Solution for Constraint Forces
    (i) Do in Parallel $n = 0, \ldots, N$
            Evaluate $\mathbf{L}_n$, $\mathbf{D}_n$ and $\mathrm{b}_n$ from (3.4) and (3.7)
    (ii) Do in Parallel $n = 0, \ldots, N$
            Calculate $\boldsymbol{f}_{n,\Box}$ iteratively with J-PPCG(2) method
IV Evaluation of Motion Equations
    Do in Parallel $n = 0, \ldots, N$
        Solve for $\boldsymbol{v}_n$ from (3.10) together with (3.11)
V Inverse Acceleration Kinematics
    Do in Parallel $n = 0, \ldots, N$
        Calculate $\boldsymbol{v}_{n\gamma}$ using (3.12)
```

Table 3.2 Macroparallel algorithm

Let us comment on the parts of the algorithm which are not macroparallelizable. As indicated above, Step II of the algorithm involves evaluation of the absolute velocities $\boldsymbol{v}_n$ given the joint rates $\boldsymbol{v}_{n\gamma}$. This is done according to the velocity kinematics relations (3.9) and is a recursive computation because of the recursive data dependency. Therefore, this calculation cannot be executed in parallel for $n = 0, \ldots, N$. In addition to this, we observe that most of the J-PPCG(2) iterative procedure is macroparallelizable except for the computation of parameters α and β. These involve summations of the form $\sum_{n=0}^{N} d_n$ where d_n are appropriate scalars that are independent for $n = 0, \ldots, N$ and therefore, can be evaluated in parallel. However, the part of the calculation whereby d_n are summed over n is not macroparallelizable. A preliminary investigation conducted by SHARF [1990] on a serial architecture indicated that these non-macroparallelizable parts of the algorithm constitute a relatively minor part of the whole computation.

We also note that the convergence criterion implemented in the parallel iterative algorithm is based on a *local relative* error. It represents the norm of the difference between two successive constraint force iterates for each body measured as a fraction of the norm of the recent iterate for *that* body. This criterion is different from the standard one used for checking convergence—the global error—and was proposed by SHARF [1990] in order to make the convergence check a macroparallel computation. To achieve this, it was necessary to eliminate the step of summing the norms of the iterates over all bodies, a calculation similar to that carried out in evaluating α and β. In that case, this step cannot be avoided, while as shown in SHARF [1990], the local relative error provides a valid test for convergence of the iteration.

Finally, we observe that the number of iterations required to achieve convergence grows linearly as the size of the system, that is, N, increases. As is known [AXELSSON, 1985], the use of preconditioning improves the theoretical upper bound on the rate of convergence of the classical conjugate gradient iteration. Furthermore, since the iterative solution for constraint forces is carried out at every step in the simulation, the solutions at the previous time steps are used to make the initial guess for the iteration at the current time step. As a result, the number of iterations taken to converge is on average considerably better than the aforementioned theoretical bound [SHARF, 1990]. Since each iteration includes the tasks of summing for α and β, we can see that the total execution time will increase proportional to N^2. For the timing results presented in the following sections, the number of iterations was set to the average over time steps observed in simulations of a generic multi-body maneuver for a chain of 1 to 10 bodies.

3.2 Architectural Considerations

When designing a parallel algorithm, consideration must always be given to the effects that the computer system will have on the performance. In fact, for parallel implementations, the computer architecture is as much a part of the design as the algorithm is. This section discusses the design of the parallel simulation dynamics algorithm in the context of a message-passing architecture.

As described earlier, the algorithm exploits macroparallelism in the multibody dynamics problem and can be classified as coarse-grain. Each processor executes essentially the same instructions on different data, with the exception of those associated with the end bodies. We also noted that some portions of the algorithm are not macroparallelizable, in particular, the computation of α and β which is done at every iteration in the solution for constraint forces. These characteristics make a loosely synchronized multiple-instruction-stream, multiple-data-stream (MIMD) architecture an appropriate choice for implementation. (Here, the term loosely synchronized means that synchronization occurs only between neighbouring processors when communication is required.) The algorithm does not require any communication beyond a processor's nearest (two) neighbours. Therefore, we have chosen a chain topology for the transputer network to match the physical topology of the multibody system. On the other hand, the non-macroparallelizable parts of the algorithm may be more efficiently executed on a network with a richer interconnection topology or a shared memory and hence, the present choice of parallel processing system may not be optimal for the algorithm. The transputer network was employed for the present implementation, however, because of its accessibility and popularity.

Now let us consider an example of how the processor chain topology affects the design of the algorithm. There are many occasions throughout the algorithm when each processor must exchange information with both of its neighbours, for example, the current values of the constraint forces. Since each processor can only perform one task at a time, we must decide the order in which to perform the two send and two receive tasks. To prevent deadlock, we must specify the order for each processor such that no two neighbouring processors are both attempting to send or receive at the same time. One option is to have each processor receive from its inward neighbour (lower node number) before sending to its outward neighbour, and then repeat in the other direction. However, this results in a serial communication pattern as the communications propagate down the chain and back, creating precedence constraints which introduce idle time. A better solution is to have every even-numbered node send outwards while every odd-numbered node receives from its inward neighbour. The even-numbered nodes then send inwards, receive from outwards, and receive from inwards. Although such a simple problem can easily be dealt with during the programming stage of development, similar problems may require modifications to the algorithm itself.

The issue of how the convergence check is accomplished provides a more substantial example of how the computer architecture can affect the parallel algorithm. In fact, the choice of a local convergence check in §3.1 was motivated by consideration of the architecture during algorithm design. However, the implementation of this check was not discussed in SHARF [1990]. The following paragraphs describe a method for accomplishing the local convergence check which has almost no effect on the performance. The solution is given here in some detail, both as an illustration of the considerations which must be made, and to describe how it could be implemented. Because the experimental measurements presented in this chapter were obtained for fixed numbers of iterations, in order to match the experiments on the serial implementation, the convergence check was not needed. However, it is a very important part of a *complete* simulation program which comprises not only the solution for the accelerations, but also their integration for the trajectory of the system.

Implementation Solution for Local Convergence Check

The advantage of using a local relative error is that each processor can determine whether its own iterations have converged, without communicating with the other processors. Assuming that some other processors may need to continue their own iterations, a converged processor can signal its neighbours that it has converged and that they should use the last values received in all further calculations. Thus, the converged processor may now continue with the next step of the computation, taking advantage of its early convergence.

There is an implementation problem, however, in making the convergence check a macroparallel computation. The problem is created by the processor interconnection topology, which we have specified to be a chain, and arises when a processor converges. For the other processors to continue their own iterations after a processor has converged, the summing for α and β must continue Although the last quantities communicated by the converged processor can be remembered and used by its neighbours, there is no way to perform the sums for α and β without involving the converged processor in communications along the chain Recall from the transputer hardware model of §2 4 1 that the fastest communications methods require that each transputer execute only a single process Therefore, the converged processor must either proceed to the next step of the computation or continue passing information along the chain If the converged processor is held back from further computation, it must determine when all processors have converged before it can proceed Establishing the convergence of all of the processors, however, is a global decision which defeats the purpose of the local convergence criterion Such a *boolean* global convergence check requires the same communications (though slightly shorter messages) as a numerical global convergence check (of the absolute error), thus incurring almost the same communication costs

The simplest solution to this problem is a compromise between the efficiency of the local convergence check and the necessity of global communications Rather than wait until all processors have converged, as required by a global convergence check, a converged processor need only wait until all processors between it and the nearest end of the chain have converged After this time, the converged processor will no longer be needed for communications along the chain and can continue to the next computation Thus, in general, a processor will have to wait for some, but not all, of the other processors to converge, only the two end processors will never have to wait

This solution can be implemented with little additional communications overhead in the following way Each of the summing processes for α and β begins at the two ends of the chain and accumulates the sum at a node in the middle of the chain (This has been incorporated in the present implementation) The sum is then distributed simultaneously in both directions. The convergence state of the processors can be communicated by appending a single number to both of the inward bound and both of the outward bound messages. Each number names the node which has converged and for which all other nodes farther from the middle of the chain have converged. Thus, these numbers are initially -1 and $N+1$. These two nodes form the edge of the set of processors no longer participating in the iterations. On the inward passes, a node normally forwards the same number it receives. If it determines, however, that it is on the edge of the set of remaining processors, the node must do one of two things. If it has not converged, it must remember the partial sums and constraint forces last communicated by its converged neighbour toward the end. Otherwise, if it has converged, it forwards its own number and proceeds to the next step in the computation after the iteration for the constraint forces. A node in the interior of the chain which has converged but does not receive its neighbour's number must continue to participate in the summing for α and β. The central node which performs the final sum then distributes the sum and the final two node numbers in both directions along the chain. These node numbers are used by the remaining nodes during the next sum to determine the new central node. Thus, the node which performs the final sum will change dynamically to remain in the middle of the iterating processors. Experimental evidence presented in SHARF [1990] indicates that, for a fixed-base manipulator, the nodes will likely converge from the base to the tip.

The method just described introduces no additional communication costs other than slightly longer messages. Since the simulation dynamics algorithm requires relatively short messages, communication costs are incurred as idle time as much as transmission time. Idle time occurs when a processor must wait to synchronize for communication. Clearly, the most idle time is created by the non-macroparallelizable parts of the algorithm which require processors to wait as information is propagated along the chain. In addition to reducing the amount of non-macroparallelizable computation, it is also desirable to reduce the amount of synchronization between neighbouring processors. Therefore, for this algorithm, the *number* of messages affects the communication costs at least as much as the length of the messages. The next section illustrates the costs of various aspects of communication on a network of transputers.

3.3 Communications Overhead

The experiments described in this chapter were all conducted on a network of T800 transputers connected to a UNIX host and running under the Trollius operating system supplement [TROLLIUS, 1990]. A Trollius kernel runs on each transputer and provides access to the host node's operating system and file system, often by using commands identical to UNIX. Trollius also acts as a supplement to UNIX on the host node, providing commands to boot the network, load and execute programs, and monitor the execution and state of transputer processes. Besides host access, other Trollius libraries provide communications functions which allow C or FORTRAN programs running on a transputer to pass messages to other processes on the same or different nodes, or to interact with the kernel.

There are four communications levels within Trollius which allow the user to trade off speed *vs.* functionality. They are, in order of decreasing functionality, the transport, network, datalink, and physical levels. The *physical* level offers the fastest communication speed, and uses *virtual circuits* in which only nearest-neighbour communications are possible. A virtual circuit prevents other processes, including system processes, from accessing the communication link. The *datalink* level provides optional message buffering on both the sending and receiving nodes. Messages can also be labelled as different *events* and *types*, thereby allowing the receiving process to synchronize only with certain messages. The *network* level supports virtual circuits among non-neighbouring nodes and adds routing and long message packetization. In addition to this, the *transport* level ensures that both the sending and receiving processes have synchronized before the message is sent. This prevents the sender from filling the message buffers and communications links with too many messages.

To obtain the best performance, it is often necessary to fine-tune the code to better suit a particular application or implementation. All of the communications levels utilize the same data structure for specifying the necessary attributes of a message, such as the event, type, length, destination, and memory location of the message. The lower communications levels simply ignore the entries in the data structure not needed by their reduced functionalities. This makes it easy to successively improve performance by changing the name of the communication routine in the calling statement. Also included in the structure is a 32-byte "data pouch" which allows short numeric messages to be passed along with longer messages. During the development phase, it was discovered that all of the communications levels, including the physical level, send the data pouch whether it is used or not. This represents a significant overhead when the actual message of interest is only eight bytes, the length of one double precision number. This feature was therefore removed, essentially reducing the physical communication level to the primitive communications functions provided by Occam, the transputer's native language. Since the simulation dynamics algorithm has a simple nearest-neighbour communication pattern and does not require the capabilities of the higher communication levels, we are able to take advantage of the most efficient mode of communication provided by Occam.

3.4 Experimental Measurements

For the purpose of parallel performance analysis and modelling, an algorithm is often subdivided into components which are either parallelizable or strictly serial This means that a given portion can either keep *all* processors busy or only one Clearly, it is desired to minimize the serial part in order to maximize the potential speedup We have seen in §3 2, however, that the summing for α and β begins at both ends of the chain at once and therefore is not a strictly serial computation Nevertheless, we will use the term "serial" to refer to all non-macroparallelizable parts of the algorithm This section discusses how the execution times associated with the serial components of the macroparallel algorithm for simulation dynamics were determined This data is subsequently employed by the performance modelling in the next section

As described in §3 1, the serial parts of the macroparallel algorithm are the computation of α, β, and $\boldsymbol{v}_n$, $n = 0, \ldots, N$ Measuring their execution times in the parallel implementation was not straightforward Initially, the program execution was traced by recording the times at which all communications began and ended Although this method can provide a wealth of information such as processor utilization, load balancing, and idle times, it was found that too much overhead was required to generate this information To find a timing method which did not add significant overhead, the number of iterations were set to be constant as the length of the chain increased Thus, the total execution time of the macroparallel parts of the algorithm should also remain constant However, even measuring the times at the beginning and ending of the non-macroparallelizable tasks was found to add an increasing (though small) amount to the total execution time as the length of the chain increased In both cases the Trollius trace facility was used, and it is thought that this was the source of the additional overhead

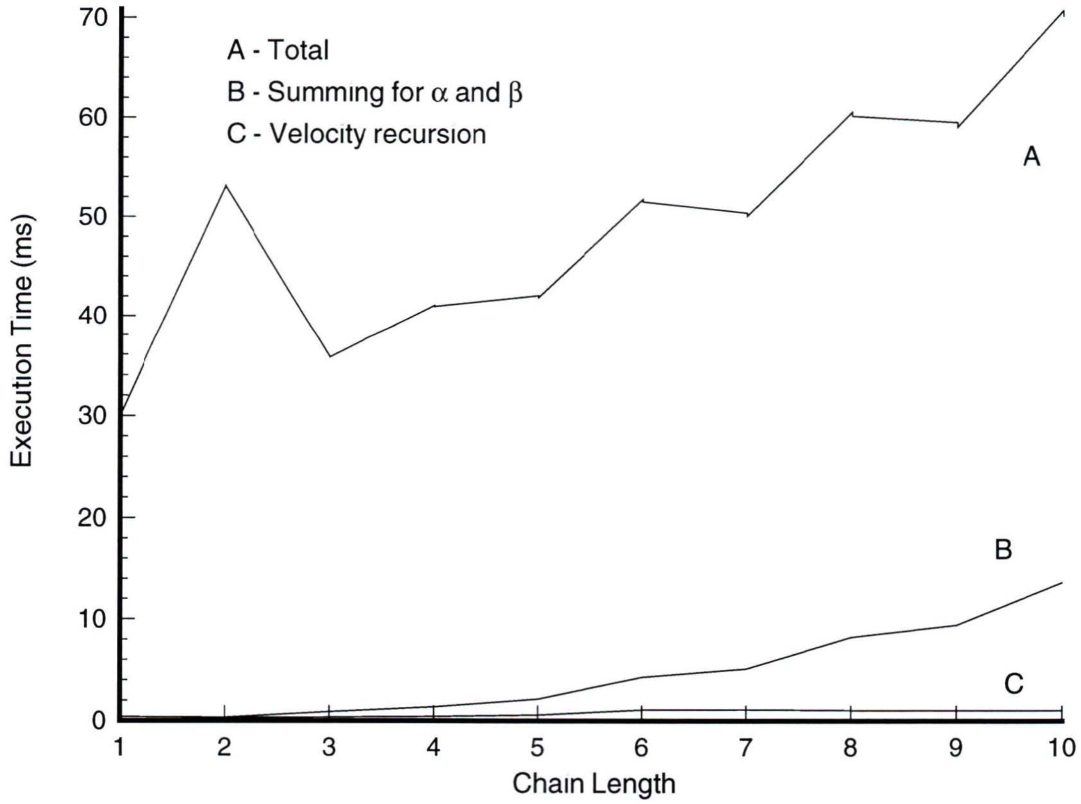

Figure 3.1: Execution times of the complete algorithm and for the non-macroparallelizable parts.

Because of these complications, a different approach to measuring the execution times of the non-macroparallelizable tasks is taken. Instead of directly timing them, the source code for each of these tasks is deleted in turn and the resulting execution time for the algorithm is measured. Thus, the difference between the total execution times with and without each task represents the execution time of each serial component of the algorithm, including computation, communication, and idle times. Figure 3.1 shows the total execution time using the Occam level of communications and the execution times attributed to the velocity recursion and the summing required for α

and β, as a function of the size of the system

There are two aspects of Figure 3 1 which should be commented on The first is the unusually high execution time required for $N = 2$ This behaviour only appears when communication modes without buffers are used so it is supposed that, for $N = 2$, the performance gain from parallelism is offset by the additional communications overhead The second unusual aspect is that, for $N \leq 10$, the time required to perform the velocity recursion peaks at N=7 Since there is no overhead incurred by the timing method, this phenomenon is probably caused by changes in the amount of idle time between tasks In some cases, better synchronization between the processors may reduce the total idle time, thereby accomplishing more work in less time

A better alternative for obtaining the execution time of the serial parts of the procedure would be to manage the trace information storage from within the algorithm, avoiding the Trollius trace facility completely This was not done because it was felt that the indirect timing method was sufficiently reliable

3.5 Performance Models

Accurate performance models are important not only for predicting the behaviour of an algorithm beyond the observed test cases, but also for aiding the design of new algorithms by exposing the limitations of the current one Using the timing measurements obtained in the previous section, as well as serial execution times, this section presents performance models which attempt to predict the execution time of the macroparallel algorithm as a function of N

Let us denote the actual execution time for an N-body chain on a single transputer by T_1 and that on N transputers by T_N (Note that T_1 is a function of N and represents the corresponding uniprocessor execution time for whatever value of N is being discussed.) Next, we consider the experimental speedup of an N transputer implementation over a single transputer given by

$$S_N = \frac{T_1}{T_N} \tag{3.13}$$

For comparison, we describe two theoretical models for predicting the speedup of parallel algorithms. The first model, based on Amdahl's law [Dongarra *et al.*, 1991], attempts to predict the parallel execution time using the single processor execution time. It requires the number of operations that can be executed in parallel on N processors as a fraction of the total operation count. We denote this fraction by f_1. The predicted parallel execution time is then

$$T_{A_N} = \frac{f_1 T_1}{N} + (1 - f_1)T_1 \tag{3.14}$$

and the corresponding predicted speedup is

$$S_A = \frac{N}{f_1 + (1 - f_1)N} \tag{3.15}$$

To determine S_A, we estimated f_1 by timing the serial parts of the algorithm on one transputer. We denote those serial parts by $T_{\mathrm{II},1}$ and $T_{(\alpha,\beta),1}$, the times required on one transputer for the velocity recursion (Step II) and for the summing for α and β respectively.

Gustafson [1988] proposed to estimate the single processor execution time using the execution time of an N-body problem on a chain of N processors. In contrast to the original premise of Amdahl's law, the problem size does not remain fixed but is allowed to change to fit the available number of processors. Another distinction of Gustafson's model is that the fraction f_N is measured as the time spent in evaluating the parallel parts of the algorithm in the *actual* parallel implementation. The predicted uniprocessor execution time is then

$$T_{G_1} = (1 - f_N + N f_N)T_N \tag{3.16}$$

resulting in a speedup of

$$S_G = N - (N-1)(1 - f_N). \tag{3.17}$$

We determined f_N according to

$$1 - f_N = \frac{T_{(\alpha,\beta),N} + T_{\|,N}}{T_N} \tag{3.18}$$

where $T_{(\alpha,\beta),N}$ and $T_{\|,N}$ are the execution times in the parallel implementation of the non-macroparallelizable parts of the algorithm (averaged over $n = 0, \ldots, N$). These were measured as was described in §3.4 and are represented by lines B and C in Figure 3.1.

Sharf [1990] predicted the macroparallel algorithm's execution speed on a parallel processing system using the following expression:

$$T_P = 1.3\frac{[T_1 - T_{(\alpha,\beta),1} - T_{\|,1}]}{N} + T_{(\alpha,\beta),1} + T_{\|,1} \tag{3.19}$$

This model allocates 30% of the parallel computation time to the associated overhead and adds to this the time for the serial parts of the algorithm. The estimate of 30% overhead was based on practical experience with a similar multi-processor implementation [Kasahara *et al.*, 1987; Kasahara, 1989].

It is interesting to combine Gustafson's and Sharf's models by adding a 30% overhead to the parallel fraction of Gustafson's model. Figure 3.2 shows the experimental speedup and the speedup predicted by Amdahl's, Gustafson's, Sharf's, and the combined models.

It is worthwhile to observe that the above performance models all relate the single processor execution time T_1 and the parallel execution time T_N. Indeed, one can show

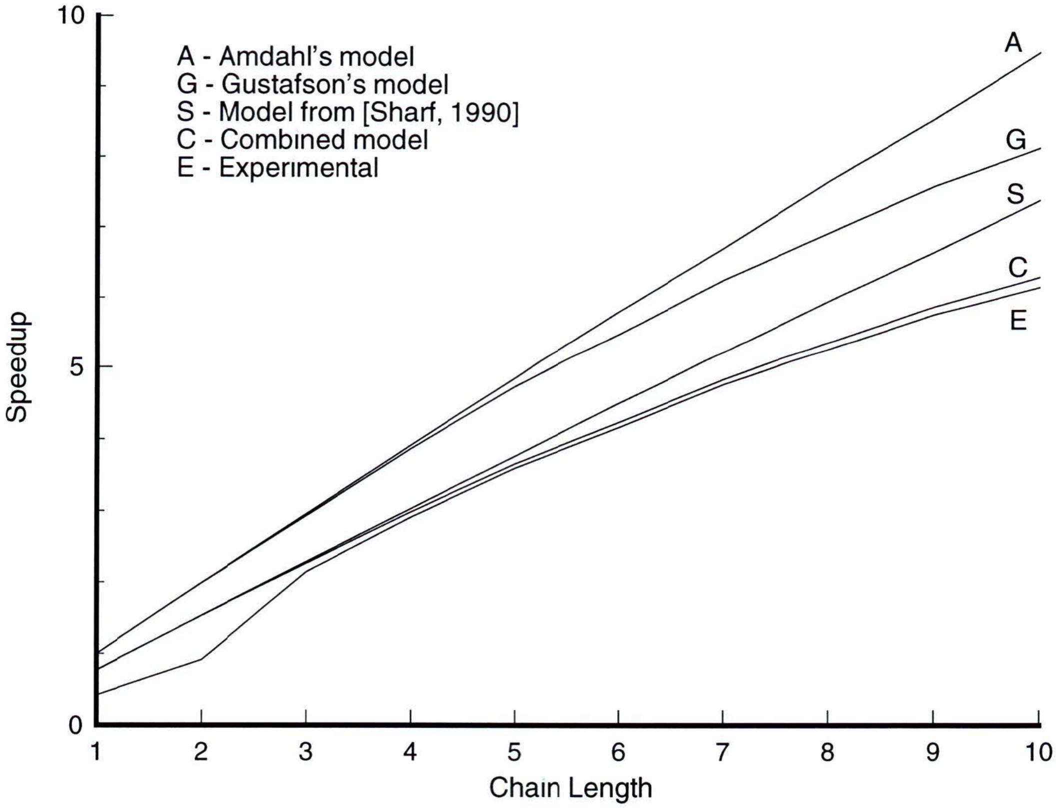

Figure 3.2. Experimental and theoretical speedups.

that (3.18) implies (3.19) with zero overhead (the coefficient in (3.19) equal to 1). This is also true of Amdahl's model if we express f_1 as

$$f_1 = \frac{T_1 - T_{(\alpha,\beta),1} - T_{\parallel,1}}{T_1} \qquad (3.20)$$

The salient feature of Gustafson's model, however, is that the execution times of the serial parts of the algorithm are measured in the parallel implementation rather than the single processor implementation. Amdahl's model predicts the maximum speedup possible for a given algorithm assuming no communications overhead. Gustafson's model also measures the fraction of execution time spent on the serial parts of the algorithm This fraction varies mostly with the number of processors, not the size of the problem, and so can not be accurately determined from measurements of the serial implementation Sharf's model includes an estimate of the overhead incurred in the parallel implementation of the *parallel* parts of the algorithm Thus, the combined model attempts to account for the varying serial fraction and the parallel overhead

The difference between Amdahl's and Gustafson's speedup models illustrates the following aspect of the parallel implementation The single processor execution times of the non-macroparallelizable parts form only a small fraction of the total execution time This is reflected in Figure 3 2 by the near ideal speedup (of N) predicted by Amdahl's law However, these "serial" parts of the algorithm comprise a significant portion of the total parallel execution time as shown in Figure 3 1 For example, summing N scalars distributed over N processors requires considerably more time than it does on 1 processor Although it can be said that the computation has been effectively distributed across N processors, since only a small fraction of the number of operations is not macroparallelizable, it is clear that a more precise model is required to account for the significance of the non-macroparallelizable parts of the algorithm

Including the overhead of the macroparallelizable parts of the algorithm more closely reflects the observed speedup, as shown by Sharf's model in Figure 3 2, but the speedup still increases linearly Only Gustafson's and the combined models, which include the effect of a varying serial fraction, predict speedups which increase less than linearly as observed experimentally

Perhaps the most important measure of the macroparallel algorithm's usefulness is its *absolute* speedup It is defined as improvement in performance over the best serial algorithm, the recursive algorithm for simulation dynamics

$$S_{\mathsf{P/R}} = \frac{T_{\mathsf{R}}}{T_N} \tag{3 21}$$

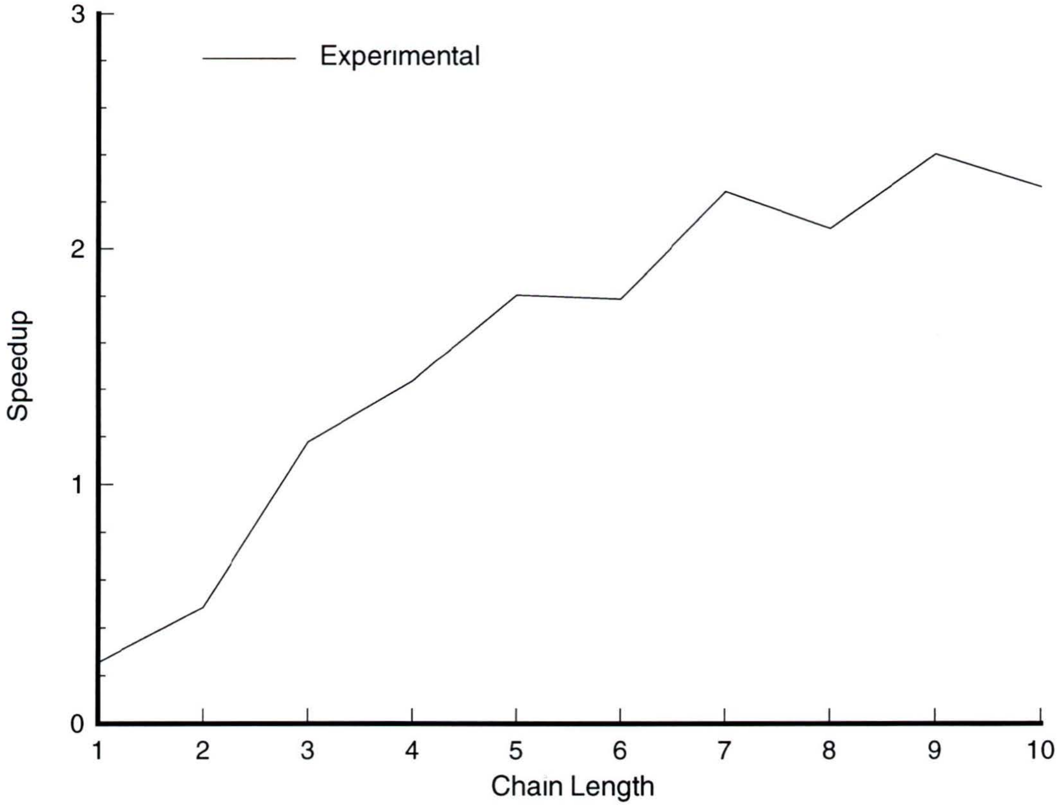

Figure 3.3 Absolute speedup

The plot of $S_{\mathsf{P/R}}$ is given in Figure 3.3 The execution times of the recursive algorithm were measured on a workstation and thus were adjusted for differences in CPU speed Although the absolute speedup is worse than the relative speedup, there is still a significant gain over the best serial algorithm

This chapter has thus demonstrated that the macroparallel algorithm is a viable solution to the simulation dynamics problem The discussion throughout this chapter has shown that there are many issues that arise in the design and analysis of a parallel algorithm In fact, there are so many implementation considerations which must be made during the design process that it becomes necessary to use a detailed hardware model and an accurate accounting of the computation and communication costs to predict the parallel performance of an algorithm. For algorithms for which this is not practical, such as the macroparallel algorithm, it is essential that the algorithm be implemented on a parallel computer system for there to be any certainty of its relative merits. This chapter has also conveyed the experience gained during the implementation of the macroparallel algorithm. Hopefully, this experience will aid further development of parallel algorithms for simulation dynamics of robotic systems.

Chapter 4

Conclusions

The goal of this thesis was to gain an understanding of the factors which affect the parallel performance of parallel dynamics algorithms in robotics. This was accomplished by modelling the performance of the algorithms and comparing predictions of their performance with actual implementation results. The algorithms were implemented on a message-passing architecture composed of transputers. It was shown that the architecture of the computer system being used is an important consideration not only in analyzing performance results, but also during the design of the algorithms.

Chapter 2 provided an extensive comparison of the parallel performance properties of six robot inverse dynamics algorithms. Such a comparison becomes a necessity as more new algorithms are proposed in the literature. The algorithms were described in a common notation for easy comparison, and were also presented visually as task graphs. The graphs illustrated the maximum parallelism of each algorithm and highlighted their important computational properties. With the aid of the task graphs, analytical expressions were derived which expressed the minimum possible execution time of each algorithm in terms of floating-point operations. Although similar expressions were given in the literature, the expressions presented here are the first derived from a common choice of tasks and level of parallelism. Therefore, they provide the first meaningful theoretical comparison of several parallel algorithms for the inverse dynamics problem.

For the parallel computer system chosen, a transputer network, it is known that communication costs have a significant effect on the performance of the inverse dynamics algorithms. There is no published parallel performance comparison of the inverse dynamics algorithms which includes communication costs. This thesis provides such a comparison. After verifying a hardware model of the transputer, the analysis uses a scheduling algorithm which incorporates the hardware model to search for the optimal assignment of tasks to processors. Although the predicted performances are only directly valid for a transputer network, the relative performance comparison is more generally applicable to other message-passing architectures. An important original result is that although the logarithmic parallelism of the **RECNEb** algorithm gives it the fastest execution time in terms of floating-point operations, it also requires a large number of precedence constraints, and hence communications, making the **RECNEb** algorithm a mediocre performer. Also, the Resolved Newton-Euler algorithm was shown to be consistently the best performer. The analysis of Chapter 2 makes the reasons for its success clear. It is important that such experimental evidence, combined with performance analysis, be presented, since the slower Recursive Newton-Euler algorithm continues to be the popular choice.

The purpose of Chapter 3 was to provide a proof-of-concept for a recently proposed simulation dynamics algorithm. Since it is not practical to undertake the kind of in-depth analysis applied to the inverse dynamics algorithms, simpler performance models were proposed and evaluated. The execution times of the non-macroparallelizable parts of the algorithm were measured (both in the single and multi-processor implementations) in an attempt to model the overall performance of the parallel implementation It was found that although these non-macroparallelizable parts were insignificant in a single processor implementation, they had a substantial effect on the parallel performance This illustrates the difficulty of predicting parallel performance based on serial performance, and the need for better models of parallel performance Detailed discussions of the architectural considerations in algorithm design and the effects of communication overhead on performance are given to relate the experience gained Hopefully, this experience will be useful for future algorithm development The parallel implementation on a transputer network also showed that the macroparallel algorithm achieves an absolute speedup over the best serial algorithm

Bibliography

Andersson, R L (1989) Computer architectures for robot control A comparison and a new processor delivering 20 real MFLOPS In *Proceedings of the 1989 IEEE International Conference on Robotics and Automation*, pp 1162–1167

Axelsson, O (1985) A survey of preconditioned iterative methods for linear systems of algebraic equations *BIT*, 25 166–187

Barhen, J (1987) Hypercube ensembles An architecture for intelligent robots In J H Graham (Ed), *Computer Architectures for Robotics and Automation*, chapter 8, pp 195–236 Gordon Breach Science Publishers, New York

Binder, E E & Herzog, J H (1986) Distributed computer architecture and fast parallel algorithms in real-time robot control *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-16(4) 543–549

Dongarra, J J , Duff, I S , Sorensen, D C , & van der Vorst, H A (1991) *Solving Linear Systems on Vector and Shared Memory Computers* SIAM, Philadelphia

Featherstone, R (1987) *Robot Dynamics Algorithms* Kluwer Academic Publishers, Boston

Fijany, A & Bejczy, A K (1989) A class of parallel algorithms for computation of the manipulator inertia matrix *IEEE Transactions on Robotics and Automation*, 5(5) 600–615

Fijany, A & Bejczy, A K (1991) Parallel computation of manipulator inverse dynamics *Journal of Robotic Systems*, 8(5) 599–635

Gustafson, J L (1988) Reevaluating Amdahl's law *Communications of the ACM*, 31(5) 532–533

Hageman, L A & Young, D M (1981) *Applied Iterative Methods* Computer Science and Applied Mathematics Academic Press, Toronto

Hashimoto, K & Kimura, H (1989) A new parallel algorithm for inverse dynamics *International Journal of Robotics Research*, 8(1) 63–76

Hashimoto, K , Ohashi, K , & Kimura, H (1990) An implementation of a parallel algorithm for real-time model-based control on a network of microprocessors *International Journal of Robotics Research*, 9(6) 37–47

Hollerbach, J M (1980) A recursive Lagrangian formulation of manipulator dynamics and a comparative study of dynamics formulation complexity *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-10(11) 730–736

Hwang, J -J , Chow, Y -C , Anger, F D , & Lee, C -Y (1989) Scheduling precedence graphs in systems with interprocessor communication times *SIAM Journal of Computing*, 18(2) 244–257

Izaguirre, A , Hashimoto, M , Paul, R P , & Hayward, V (1992) A new computational structure for real-time dynamics *International Journal of Robotics Research*, 11(4) 346–361

Kasahara, H (1989) Private communication with I Sharf

Kasahara, H , Fujii, H , & Iwata, M (1987) Parallel processing of robot motion simulation In *Proceedings of the IFAC 10th Triennial World Congress*, pp 329–340

Kasahara, H & Narita, S (1985) Parallel processing of robot-arm control computation on a multimicroprocessor system *IEEE Journal of Robotics and Automation*, RA-1(2) 104–113

Kokaji, S (1986) Collision-free control of a manipulator with a controller composed of sixty-four microprocessors *IEEE Control Systems Magazine*, pp 9–14

Kwan, A W , Bic, L , & Gajski, D D (1990) Improving parallel program performance using critical path analysis In D Gelernter, A Nicolau, & D Padua (Eds), *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, chapter 18, pp 358–373 The MIT Press

Lathrop, R H (1985) Parallelism in manipulator dynamics *International Journal of Robotics Research*, 4(2) 80–102

Lee, C S G & Chang, P R (1986) Efficient parallel algorithm for robot inverse dynamics computation *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-16(4) 532–542

Lee, C. S. G. & Chang, P. R. (1988). Efficient parallel algorithms for robot forward dynamics computation. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-18(2):238–251.

Lee, C. S. G. & Chen, C. L. (1990). Efficient mapping algorithms for scheduling robot inverse dynamics computation on a multiprocessor system. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(3):582–595.

Li, C.-J. & Sankar, T. S. (1992). Fast inverse dynamics computation in real-time robot control. *Mechanism and Machine Theory*, 27(6):741–750.

Liao, F.-Y. & Chern, M.-Y. (1985). Robot manipulator dynamics computation on a VLSI array processor. In *Proc. First Int. Conf. Supercomp. Sys.*, pp. 116–125.

Luh, J. Y. S. & Lin, C. S. (1982). Scheduling of parallel computation for a computer-controlled mechanical manipulator. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-12(2):214–234.

McMillan, S., Orin, D. E., & Sadayappan, P. (1991). Real-time robot dynamic simulation on a vector/parallel supercomputer. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pp. 1836–1841.

Nigam, R. & Lee, C. S. G. (1985). A multiprocessor-based controller for the control of mechanical manipulators. *IEEE Journal of Robotics and Automation*, RA-1(4):173–182.

Paul, R. P. (1981). *Robot Manipulators: Mathematics, Programming, and Control*. The MIT Press, Cambridge, Massachusetts.

Rahman, M. & Meyer, D. G. (1987). A cost-efficient high-performance bit-serial architecture for robot inverse dynamics computation. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-17(6):1050–1058.

Rajagopalan, R., Cheng, R. M. H., & Poon, S. (1990). Parallel computation of inverse dynamics of robots employing transputers. In *ISMM Conf. on Parallel Processing and Distributed Computing*.

Sadayappan, P., Ling, Y.-L. C., Olson, K. W., & Orin, D. E. (1989). A restructurable VLSI robotics vector processor architecture for real-time control. *IEEE Transactions on Robotics and Automation*, 5(5):583.

Sharf, I. (1990). *Parallel Simulation Dynamics for Open Multibody Chains*. PhD thesis, Institute for Aerospace Studies, University of Toronto.

Trollius (1990) *Trollius Library Reference in C* The Ohio State University and Cornell Research Foundation

Vukobratović, M & Kirćanski, N (1985) *Real-time Dynamics of Manipulation Robots* volume 4 of *Scientific Fundamentals of Robotics* Springer-Verlag, New York

Vukobratović, M , Kirćanski, N , & Li, S G (1988) An approach to parallel processing of dynamic robot models *International Journal of Robotics Research*, 7(2) 64–71

Wong, D & Lawrence, P D (1992) Parallel computation of forward dynamics for serial chain multibody systems using orthogonal complements Technical Report CICSR-TR92-009, Centre for Integrated Computer Systems Research, University of British Columbia

Xiao, J W & Cheng, R M H (1991) Efficient parallel computation of robot inverse dynamics In *2nd National Applied Mechanisms and Robotics Conf* , pp VIIB 3-1–VIIB 3-5

Yoshikawa, T (1990) *Foundations of Robotics Analysis and Control* The MIT Press, Cambridge, Massachusetts

Zalzala, A M S & Morris, A S (1991) A distributed pipelined architecture of robot dynamics with VLSI implementation *International Journal of Robotics and Automation*, 6(3) 117–128

Zhang, H & Paul, R P (1986) A robot force and motion server In *Proc ACM/IEEE-CS Fall Joint Computer Conf* , pp 178–184

Zheng, Y -F & Hemami, H (1986) Computation of multibody system dynamics by a multiprocessor scheme *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-16(1) 102–110

Zomaya, A Y (1992) Highly efficient transputer arrays for the computation of robot dynamics *Concurrency Practice and Experience*, 4(2) 185–205

Zomaya, A Y & Morris, A S (1990) Modeling and simulation of robot dynamics using transputer-based architectures *Simulation*, pp 269–278

VITA

Surname Pond Given Names Christopher Burke

Place of Birth Mississauga, Ontario Date of Birth August 3, 1967

Educational Institutions Attended

University of Victoria	1991–1993
University of Waterloo	1986–1991

Degrees Awarded

B.Math. (Honours Co-op)	University of Waterloo	1991

Honours and Awards

Descartes Fellowship	1988–1991
Descartes Entrance Award	1986

Publications

B. Pond, I. Sharf. A Parallel Algorithm for Dynamics Simulation of Multibody Chains—Implementation on a Transputer System. *Proceedings of the 26th Annual Simulation Symposium*, Washington, D.C., 1993.

PARTIAL COPYRIGHT LICENSE

Title of Thesis

Parallel Algorithms for Dynamics of Robotic Manipulators

Author [redacted]

Christopher Burke Pond

Aug. 31, 1993

Date