2-1-1987

Efficient Parallel Algorithm for Robot Forward Dynamics Computation

C. S. G. Lee
Purdue University

P. R. Chang
Purdue University

Efficient Parallel Algorithm for Robot Forward Dynamics Computation

C. S. G. Lee
P. R. Chang

TR-EE 87-1
February 1987

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Efficient Parallel Algorithms For Robot Forward Dynamics Computation

C. S. G. Lee and P. R. Chang

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

TR-EE 87-1

February 1987

ABSTRACT

Computing the robot forward dynamics is important for real-time computer simulation of robot arm motion. Two efficient parallel algorithms for computing the forward dynamics for real-time simulation were developed to be implemented on an SIMD computer with n processors, where n is the number of degrees-of-freedom of the manipulator. The first parallel algorithm, based on the Composite Rigid-Body method, generates the inertia matrix using the parallel Newton-Euler algorithm, the parallel linear recurrence algorithm, and the row-sweep algorithm, and then inverts the inertia matrix to obtain the joint acceleration vector desired at time t. The time complexity of this parallel algorithm is of the order $O(n^2)$ with $O(n)$ processors. Further reduction of the order of time complexity can be achieved by implementing the Cholesky's factorization procedure on array processors. The second parallel algorithm, based on the conjugate gradient method, computes the joint accelerations with a time complexity of $O(n)$ for multiplication operation and $O(n \log n)$ for addition operation. The proposed parallel computation results are compared with the existing methods.

This work was supported in part by the National Science Foundation Engineering Research Center Grant CDR-8500022. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the funding agency.

1. Introduction

Manipulator dynamics plays a major role in the analysis, design, and synthesis of control law for the manipulator, as well as computer simulation of robot arm motion. Recent research focuses more on the former problem than the latter problem. However, real-time computer simulation of robot arm motion with manipulator dynamics taken into consideration offers an effective way of testing and verifying proposed control strategies without the expense and mechanical problems of working with the actual manipulator. This paper focuses on real-time computer simulation of robot arm motion and proposes efficient parallel algorithms for computing the joint acceleration vector of the manipulator which can be integrated to obtain the time history of the robot motion.

The simulation problem may be formulated as the forward (or direct) dynamics problem which can be stated as: Given an input force/torque vector $\tau(t)$ and a vector of external forces/torques exerted on the last link of the manipulator $\mathbf{k}(t)$, compute the joint acceleration vector $\ddot{\mathbf{q}}(t)$, based on an appropriate manipulator dynamic model, from values of $\tau(t)$, $\mathbf{k}(t)$, the joint position $\mathbf{q}(t)$, and the joint velocity $\dot{\mathbf{q}}(t)$. The resultant $\ddot{\mathbf{q}}(t)$ is then integrated to give new values of $\mathbf{q}(t)$ and $\dot{\mathbf{q}}(t)$; and the process is repeated for the next input force/torque vector.

Computationally, the dynamic equations of motion as derived from the Lagrange-Euler formulation are very inefficient and result in the order of $O(n^4)$ arithmetic operations [1] for computing the joint torques, where n is the number of degrees-of-freedom of the manipulator. The Newton-Euler formulation [2] was utilized as an alternative to deriving more efficient equations of motion for computing the joint torques. Because of the recursive structure in the Newton-Euler equations of motion, the number of arithmetic operations for computing the joint torque is linearly proportional to the number of degrees-of-freedom of the manipulator. Furthermore, Lee and Chang [3] have shown that by reformulating the Newton-Euler equations of motion in a linear homogeneous recurrence form and utilizing the "recursive doubling" [4,5] technique to compute the joint torques, the computation has been shown to achieve the time lower bound of $O(\lceil \log_2 n \rceil)$. In addition to being fast and efficient in computing the joint torques, the Newton-Euler equations of motion have also been utilized by Walker and Orin [6] to compute the joint acceleration vector for computer simulation of robot arm motion. This paper focuses on extending their work by taking advantages of parallel algorithms running on a single-instruction-multiple-data-stream (SIMD) computer.

The dynamic equations of motion of a manipulator can be written as

$$\mathbf{H}(\mathbf{q})\ddot{\mathbf{q}}(t) + \mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}}(t) + \mathbf{G}(\mathbf{q}) = \tau(t). \tag{1}$$

They can be rewritten as

$$\mathbf{H}(\mathbf{q})\ddot{\mathbf{q}}(t) = \tau(t) - \mathbf{b} \tag{2.a}$$

$$\mathbf{b} = \mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}}(t) + \mathbf{G}(\mathbf{q}) \tag{2.b}$$

where $\mathbf{H}(\mathbf{q})$ is an $n \times n$ symmetric inertia matrix, $\mathbf{b}$ is the bias torque vector due to gravity $\mathbf{G}(\mathbf{q})$ and velocity terms $\mathbf{C}(\mathbf{q},\dot{\mathbf{q}})$, and $\tau(t)$ is a generalized applied force/torque vector. Utilizing the Newton-Euler equations of motion, Walker and Orin [6] considered four methods for providing solutions to the forward dynamics problem. The first three methods involve different ways of computing the symmetric $n \times n$ inertia matrix, $\mathbf{H}(\mathbf{q})$, which is then inverted to yield $\ddot{\mathbf{q}}(t)$ directly. The fourth method is an iterative procedure based on the conjugate-gradient technique to estimate the joint acceleration vector $\ddot{\mathbf{q}}(t)$ in less than n iterations. The advantage of the conjugate-gradient method is that the computation of $\mathbf{H}(\mathbf{q})$ can be avoided and at the order of $O(n^2)$ is theoretically the most efficient. But when $n = 6$ (or $n \leq 12$), methods 1-3 are more efficient because of their smaller coefficients on the complexity polynomial. For n=6, method 3 and method 4 have, respectively, 1629 scalar multiplications and 1255 scalar additions and 3435 scalar multiplications 2532 scalar additions. The computational complexity of these four methods is tabulated in Table 1 for comparison.

A different approach is proposed by Featherstone [7] who introduced a spatial notation to provide a pleasingly uniform combined representation of rotational and translational quantities. Based on the so-called articulated-body method, the joint acceleration vector $\ddot{\mathbf{q}}(t)$ can be computed in $O(n)$ steps. The evaluation is performed in two stages: First, homogeneous articulated-body inertias are calculated for each link using a fixed-step iteration that starts at the end-effector and works toward the base; second, the joint accelerations are calculated in another fixed-step iteration, this time working from the base toward the end-effector. Although the computational complexity of the method is proportional to $O(n)$, the coefficient of n is quite large. Thus, for $n = 6$, there are 2250 scalar multiplications and 1816 scalar additions. The computational complexity of this method is also tabulated in Table 1 for comparison. From Table 1, one can realize that the forward dynamics problem is more computational intensive than the inverse dynamics problem. Thus, for real-time simulation of robot arm motion, a further substantial improvement in the computational efficiency of forward dynamics computation is required.

Our present approach to the forward dynamics problem is to implement existing forward dynamics methods on parallel-computer systems to achieve the real-time requirements. Due to its recursive nature and the low order in computational complexity, Featherstone's articulated-body method was first considered for parallel computer implementation. Unfortunately, the recursive equations in Featherstone's method have a nonlinear recurrence (equation (38) in [7]) and Kung [8] showed that the parallel evaluation of a nonlinear recurrence cannot be faster than the obvious sequential algorithm by any parallel algorithm using any number of processors. In other words, the nonlinear recurrence in Featherstone's method cannot be parallelized.

Excluding Featherstone's method, one may consider the parallelization of the four methods proposed by Walker and Orin. Among these four methods, method 4 is theoretically the most efficient while method 3 is the most efficient for a reasonable n (i.e. for most industrial robot, $n \leq 12$). Thus, this paper focuses on parallelizing these two methods. For method 3 (also called the Composite Rigid-Body method), our proposed parallel algorithm reduces the computational complexity from $O(n^3)$ to $O(n^2)$, using $O(n)$ number of processors, and from $O(n^2)$ to $O(n)$, using $O(n^2)$ number of processors. For the $O(n)$ number of processors case, the parallel Newton-Euler algorithm and the row-sweeping algorithm are, respectively, used to compute the bias vector $\mathbf{b}$ and the matrix $\mathbf{H}(\mathbf{q})$ at $O(n)$ time complexity, then the set-ordering technique and the parallel Cholesky factorization and the column-sweeping algorithm are proposed to solve the linear system of equations $\mathbf{H}(\mathbf{q})\ddot{\mathbf{q}}(t) = \tau - \mathbf{b}$ at $O(n^2)$ time complexity. The bottleneck of the above parallel computation is the inversion of the inertia matrix $\mathbf{H}(\mathbf{q})$ to obtain $\ddot{\mathbf{q}}(t)$ in Eq. (2.a). So, an array processor-based VLSI architecture with $O(n^2)$ processors can be used to solve the inversion problem at $O(n)$ time complexity [9]. Furthermore, it should be noted that the coefficients of the complexity polynomials on both methods are quite small. There are $O(\lceil (n^2 - 1)/6 \rceil)$ scalar multiplications and $O(\lceil (n^2 - 1)/2 \rceil)$ scalar additions for using $O(n)$ processors and $O(7n + 9 \lceil (n - 1)/2 \rceil)$ multiplications and $O(8n + 5 \lceil (n - 1)/2 \rceil)$ scalar additions for using $O(n^2)$ processors.

For the conjugate-gradient method (method 4), the finite-step iterative procedure can be parallelized to achieve a faster computation. The proposed parallel conjugate-gradient method requires $O(1)$ scalar multiplications and $O(\lceil \log_2 n \rceil)$ scalar additions per iteration, giving $O(n)$ scalar multiplications and $O(n \lceil \log_2 n \rceil)$ scalar additions for n iterations. The computational complexity of the proposed parallel composite rigid-body method and the parallel conjugate-gradient method is tabulated in Table 1 for comparison.

2. Parallel Composite Rigid-Body Method

In this section, efficient parallel algorithms based on the Composite Rigid-Body method (See Appendix A) will be discussed. The method involves first obtaining the bias vector $\mathbf{b}$ from the parallel Newton-Euler computation, then the computation of the matrix $\mathbf{H}(\mathbf{q})$ is based on the equations in Appendix A which requires the parallel linear recurrence algorithms and the row-sweeping algorithm. Finally, a parallel linear system solver is proposed to solve for the $\ddot{\mathbf{q}}(t)$ in Eq. (2.a). The parallel Newton-Euler computation, the parallel linear recurrence algorithms, the row-sweeping algorithm, and the parallel Cholesky factorization are discussed in the following subsections.

2.1. Parallel Newton-Euler Computation

The bias vector **b** in Eq. (2.a) can be computed from the Newton-Euler equations of motion by setting the joint acceleration vector $\ddot{\mathbf{q}}(t) = \mathbf{0}$. The efficient parallel algorithm proposed by Lee and Chang [3] can be used to compute the Newton-Euler equations of motion to achieve the time lower bound of $O(\lceil \log_2 n \rceil)$. This parallel Newton-Euler algorithm can be denoted by a subroutine $NE^{(P)}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}, \tau)$ where $\mathbf{q}$, $\dot{\mathbf{q}}$, and $\ddot{\mathbf{q}}$ are, respectively, the input joint position, velocity, and acceleration, and τ is the resultant joint torque vector which is the desired bias vector **b**.

2.2. Parallel Linear Recurrence Algorithms

From the equations in Appendix A, one finds that Eqs. (A.1) and (A.3) are in homogeneous linear recurrence form while Eq. (A.2) for computing $\mathbf{c}_j$ is in an inhomogeneous linear recurrence form. These linear recurrence problems can be solved by the "recursive doubling" technique [4,5]. In general, the first-order linear recurrence problem can be stated as: Given $x(0) \neq$ identity and $a(i)$, $b(i)$, $0 \leq i \leq n$, and the recursive equation $x(i) = a(i) * x(i-1) + b(i)$, where $*$ and $+$ may be scalar (or matrix) multiplication and scalar (or vector or matrix) addition, respectively, find $x(1), x(2), \ldots, x(n)$. If $a(i)$ and $b(i)$ are both not identities, then this is the first-order *inhomogeneous* linear recurrence problem. If $a(i)$ or $b(i)$ is identity, then it becomes the first-order *homogeneous* linear recurrence problem. A parallel solution, called "recursive doubling" [4,5], is especially suited for solving the linear recurrence problems in SIMD computers. The homogeneous linear recurrence problem can be solved by the first-order homogeneous linear recurrence algorithm (FOHRA) [3], while the inhomogeneous linear recurrence problem can be solved by the first-order inhomogeneous recurrence algorithm (FOIHRA) which is stated here for convenience:

Algorithm FOIHRA. (First-Order Inhomogeneous Recurrence Algorithm) Given $a(i)$, $b(i)$, $0 \leq i \leq n$, this algorithm computes the first-order inhomogeneous linear recurrence equation using the recursive doubling technique.

Step 1. [*Initialization*] Given $a(i)$, $b(i)$, $0 \leq i \leq n$, let $X^{(k)}(i)$, $Y^{(k)}(i)$ be the ith sequences at the kth level, and let $X^{(0)}(i) = a(i)$, $Y^{(0)}(i) = b(i)$, for $0 \leq i \leq n$, and $s = \lceil \log_2(n+1) \rceil$.

Step 2. [*Compute x(i) parallelly*]
FOR k = 1 step 1 until s, DO

$$X^{(k)}(i) = \begin{cases} X^{(k-1)}(i-2^{k-1}) * X^{(k-1)}(i), & \text{if } 2^{k-1} \leq i \leq n \\ X^{(k-1)}(i), & \text{if } 0 \leq i < 2^{k-1} \end{cases} \quad (3)$$

```
$$Y^{(k)}(i) = \begin{cases} X^{(k-1)}(i) * Y^{(k-1)}(i-2^{k-1}) + Y^{(k-1)}(i), \text{ if } 2^{k-1} \le i \le n \\ Y^{(k-1)}(i), \text{ if } 0 \le i < 2^{k-1} \end{cases} \quad (4)$$
    End_DO
    Set $x(i) = Y^{(s)}(i),\ 1 \le i \le n.$
End FOIHRA
```

End FOIHRA

The "*" in Step 2 denotes an associative operator. Both FOHRA and FOIHRA algorithms have a computational complexity of $O(\log_2 n)$ which is the time lower bound of the linear recurrence problem. Equations (A.1) and (A.3) in Appendix A can be solved by the subroutine FOHRA in [3], while Eq. (A.2) by the subroutine FOIHRA.

2.3. Row-Sweeping Algorithm

Equation (A.7) in Appendix A can be conveniently expressed as a set of linear recurrence equations which can be efficiently computed by a technique called "row-sweeping" [10]. The row-sweeping algorithm is a parallel solution for solving the upper triangular linear recurrence equation system on an SIMD computer. The problem of solving a set of linear recurrence equations can be stated as: Given a_{ij}, $1 \le i \le (j-1)$, $1 \le j \le n$, and $x_{jj} = x_j^0$, $1 \le j \le n$, find x_{ij}, $1 \le i \le (j-1)$, $1 \le j \le n$, on an SIMD machine of n processors, based on the equation

$$x_{ij} = x_{(i+1),j} + a_{ij}\ . \tag{5}$$

Equation (5) can be conveniently rewritten in a matrix form,

$$\begin{bmatrix} x_{12} & x_{13} & \cdots & x_{1,n} \\ 0 & x_{23} & \cdots & x_{2,n} \\ 0 & 0 & \cdots & x_{3,n} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & x_{(n-1),n} \end{bmatrix} = \begin{bmatrix} x_{22} & x_{23} & \cdots & x_{2,n} \\ 0 & x_{33} & \cdots & x_{3,n} \\ 0 & 0 & \cdots & x_{3,n} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & x_{n,n} \end{bmatrix} + \begin{bmatrix} a_{12} & a_{13} & \cdots & a_{1,n} \\ 0 & a_{23} & \cdots & a_{2,n} \\ 0 & 0 & \cdots & a_{3,n} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & a_{(n-1),n} \end{bmatrix} \tag{6}$$

For the ith row, there are $(n-i)$ undetermined variables, i.e., $x_{i,i+1}$, $x_{i,i+2}$, ..., $x_{i,n}$ which can be evaluated from the resultant $(i+1)$th row variables, i.e. $x_{(i+1),(i+2)}, x_{(i+1),(i+3)},\ldots,x_{(i+1),n}$ and the given constants $a_{i,i+1},\ldots,a_{i,n}$ and $x_{(i+1),(i+1)} = x_{i+1}^0$. The computation starts from the bottom row and "sweeps" to the upper row. In each sweeping, a specified row is evaluated; thus, the technique is called the row-sweeping algorithm. If one assigns the $(j-1)$th processor to deal with the computation of the variables, x_{1j}, $x_{2j},\ldots,x_{(j-1),j}$ in the $(j-1)$th column, where $2 \le j \le n$, then the problem can be solved in $(n-1)$ steps. Based on the above concept, the row-sweeping algorithm may be stated as follows:

Algorithm Row-Sweep. (Row-Sweeping Algorithm) Given $a_{ij}, 1 \leq i \leq (j-1), 1 \leq j \leq n$, and $x_j^0, 1 \leq j \leq n$, this algorithm computes $x_{ij}, 1 \leq i < j, 1 \leq j \leq n$, based on equation (5).

```
Step 1. [Initialization] Let $X^{(i)}(j)$ be the result of the $j$th equation at the $(n-i)$th
        iteration and $X^{(j)}(j) = x_j^0$, where $1 \leq j \leq n$.
Step 2. [Parallel Compute $X^{(i)}(j), i < j$, in backward]
        FOR $i = (n-1)$ step -1 until 1, DO
          The $(j-1)$th processor computes $X^{(i)}(j)$, $2 \leq j \leq n$, according to Eq. (7):
        End_DO
Step 3. [Output the results] Let $x_{ij} \leftarrow X^{(i)}(j)$, $1 \leq i \leq j$, $1 \leq j \leq n$ and return.
```

$$X^{(i)}(j) = \begin{cases} X^{(i+1)}(j) + a_{ij}, & \text{if } (i+1) \leq j \leq n \\ X^{(i+1)}(j), & \text{if } 2 \leq j \leq i \end{cases} \tag{7}$$

End Row-Sweep

The row-sweeping algorithm is used to solve Eq. (A.7), that is,

$$\mathbf{f}_{i,j} = \mathbf{f}_{i+j,j} \tag{8.a}$$

$$\mathbf{n}_{i,j} = \mathbf{n}_{(i+1),j} + \mathbf{p}_i^* \times \mathbf{f}_{(i+1),j} \tag{8.b}$$

where $1 \leq i \leq (j-1)$, $1 \leq j \leq n$, and

$$\mathbf{f}_{jj} = \mathbf{F}_j \tag{9.a}$$

$$\mathbf{n}_{jj} = \mathbf{N}_j + \mathbf{c}_j \times \mathbf{F}_j ; 1 \leq j \leq n \tag{9.b}$$

where $\mathbf{F}_j$, $\mathbf{N}_j$, $\mathbf{c}_j$, are given parameters. From Eqs. (8.a) and (9.a), one finds that $\mathbf{f}_{i,j} = \mathbf{f}_{i+1,j} = \mathbf{F}_j$, and $a_{ij} = \mathbf{p}_i^* \times \mathbf{F}_j$ may be evaluated for $1 \leq i \leq (j-1)$, $1 \leq j \leq n$. Using these results, Eq. (8.b) becomes

$$\mathbf{n}_{i,j} = \mathbf{n}_{(i+1),j} + a_{ij}, \quad 1 \leq i \leq (j-1),\ 1 \leq j \leq n \tag{10}$$

and, $\mathbf{n}_{j,j} = \mathbf{N}_j + \mathbf{c}_j \times \mathbf{F}_j$, $1 \leq j \leq n$ may be evaluated before solving Eq. (10). Equation (10) is an upper triangular linear recurrence equation system and can be solved by the row-sweeping algorithm in $(n-1)$ steps.

2.4. Parallelized Linear System Solver

The above parallel linear recurrence algorithms and the row-sweeping algorithm are used to efficiently compute the equations in Appendix A to obtain the inertia matrix $\mathbf{H}(\mathbf{q})$. Thus, given the input force/torque vector $\tau(t)$ and the bias vector $\mathbf{b}$ computed from the parallel Newton-Euler computation, Eq. (2.a) becomes a set of linear system of equations in the form of $\mathbf{H}(\mathbf{q})\ddot{\mathbf{q}}(t) = \mathbf{y}$, where $\mathbf{y} = \tau - \mathbf{b}$. This set of linear system of equations can be efficiently solved by the Cholesky factorization [11].

This method solves the equations by factorizing the $n \times n$ symmetric matrix $\mathbf{H}(\mathbf{q})$ into $\mathbf{L}\mathbf{L}^T$, where $\mathbf{L}$ is a lower triangular matrix and the superscript "T" denotes matrix transpose, then it solves the subsystems in two steps: First, the equation $\mathbf{L}\mathbf{x} = \mathbf{y}$ is solved by back substitution; then, the resultant subsystem $\mathbf{L}^T \ddot{\mathbf{q}}(t) = \mathbf{x}$ is solved by another back substitution.

Since we are interested in the inversion of an $n \times n$ symmetric matrix $\mathbf{H}(\mathbf{q})$, a parallel computation of the Cholesky factorization has been developed and can be divided into two parts: First is the parallel computation of the off-diagonal lower elements l_{ki}, where

$$l_{ki} = \begin{cases} (h_{ki} - \sum_{j=1}^{i-1} h_{ij} h_{kj})/h_{ii}, \text{ if } i < k \leq n,\ 2 \leq i \leq n \\ h_{k1}/h_{11} \text{ (first}-\text{column), if } i=1,\ 1 < k \leq n \end{cases} \tag{11}$$

and the second is the parallel computation of the diagonal elements l_{kk}, that is

$$l_{kk} = \begin{cases} (h_{kk} - \sum_{j=1}^{k-1} h_{kj}^2)^{1/2}, \text{ if } 2 \leq k \leq n \\ (h_{11})^{1/2}, \text{ if } k=1 \end{cases} \tag{12}$$

where h_{ij} and l_{ij} are the (i,j) component of the matrices $\mathbf{H}$ and $\mathbf{L}$, respectively.

Basically, the elements l_{ki}, $i \geq 2$, in Eq. (11), can be obtained in three steps: The parallel computation of the product terms $(-h_{ij}\, h_{kj})$ for $1 \leq j \leq (i-1)$, $i \leq k \leq n$, $2 \leq i \leq n$; the summation of the resultant product terms; and then the computation of l_{ki}. Similarly, the element l_{kk}, $k \geq 2$, in Eq. (12), can be obtained in the same procedure. In evaluating a specified l_{ki}, $i \geq 2$, there are $(i-1)$ necessary product terms. So, the total number of necessary product terms is $\sum_{k=i}^{n} \sum_{i=2}^{n} (i-1) = (n^3 - n)/6$. Because the evaluations of these product terms are identical, it is easy to show that the computational complexity of the parallel computation using n processors is $\lceil (n^2-1)/6 \rceil$ scalar multiplications. For convenience, we let $h_{ijk} = -h_{ij}\, h_{kj}$ and the next goal is how to parallelly compute the summation $\hat{l}_{ki} = h_{ki} + \sum_{i=1}^{j-1} h_{ijk}$, $i \leq k \leq n$, $2 \leq i \leq n$. Obviously, the computations for $\hat{l}_{ki_1}$ and $\hat{l}_{ki_2}$, $i_1 \neq i_2$, or in different columns are not identical. So, there is no easy way to compute $\hat{l}_{ki}$ parallelly. More arrangements on the parallel algorithm are necessary. A parallel algorithm based on the set-ordering technique is proposed to solve the summation problem efficiently and is described below:

The parameters used in the set-ordering procedure are:

1) $\hat{l}_{ki} = h_{ki} + \sum_{j=1}^{i-1} h_{ijk}$ where $h_{ijk} = -\ h_{ij}\, h_{kj}$, for $1 \leq j \leq i-1$, $1 \leq k \leq n$,

$2 \leq i \leq n$.

2) $NA(S_l) \triangleq$ number of additions needed to evaluate $\sum_{i=1}^{l} a_i = l-1$, where S_l is a set consisting of a collection of terms $a_1, a_2, \ldots, a_l$.

3) $a_{ki}^{(C_{ki},0)} = \sum_{j=0}^{C_{ki}} a_{ki}^{(j)}$, where C_{ki} is the counter for indicating the length of summation in S_{ki}.

Procedure Set-Ordering $(\mathbf{H}, \hat{l}_{ki})$. Given h_{ki} for $i \leq k \leq n$, $1 \leq i \leq n$, and $h_{ijk} = -h_{ij} h_{kj}$ for $1 \leq j \leq i-1$, $i \leq k \leq n$, $2 \leq i \leq n$, where h_{ki} is the (k,i) component of the inertia matrix $\mathbf{H}(\mathbf{q})$, this procedure computes $\hat{l}_{ki}$ based on the set-ordering technique.

Step 1. [*Initialization*]

(i) Set $a_{ki}^{(0)} \leftarrow h_{ki}$, $a_{ki}^{(j)} \leftarrow h_{ijk}$ for $1 \leq j \leq i-1$, $i \leq k \leq n$, $2 \leq i \leq n$.

(ii) Set $S_{ki} = \{a_{ki}^{(i-1)}, a_{ki}^{(i-2)}, \ldots, a_{ki}^{(0)}\}$ for $i \leq k \leq n$, $2 \leq i \leq n$

(iii) Set $C_{ki} = 0$, $a_{ki}^{(0,0)} = a_{ki}^{(0)}$ for $i \leq k \leq n$, $2 \leq i \leq n$

(iv) Set $N = n(n-1)/2$

Step 2. [*Set Ordering*] Order S_{ki} in a descending order according to $NA(S_{ki})$, and let the sets $S_{k,i}^{(l)}$, $1 \leq l \leq N$, correspond to the ordered set sequence $S_{k,i}$.

Step 3. [*Compute the n (or N) Highest Ordered Set $S_{ki}^{(l)}$ Parallelly*] If $N \geq n$ (or $N < n$), the computation of the set $S_{ki}^{(l)}$ can be evaluated by the lth processor, $1 \leq l \leq n$ (or N),

(i) $S_{ki}^{(l)} \leftarrow S_{ki}^{(l)} - \{a_{ki}^{(C_{ki}+1)}, a_{ki}^{(C_{ki},0)}\}$.

(ii) $a_{ki}^{(C_{ki}+1,0)} \leftarrow a_{ki}^{(C_{ki}+1)} + a_{ki}^{(C_{ki},0)}$

(iii) $S_{ki}^{(l)} \leftarrow S_{ki}^{(l)} \cup \{a_{ki}^{(C_{ki}+1,0)}\}$.

(iv) $C_{ki} \leftarrow C_{ki} + 1$.

(v) $NA(S_{ki}) = NA(S_{ki}^{(l)}) = NA(S_{ki})-1$

Step 4. [*De-Ordering*] $S_{ki} = S_{ki}^{(l)}$, $1 \leq l \leq N$.

Step 5. [*Check for Termination*]

(i) Let N_1 be the number of the current S_{ki}, whose $NA(S_{ki}) = 0$.

(ii) The sets S_{ki} whose $NA(S_{ki}) \neq 0$ will be considered in the next iteration; otherwise, go to *Step* 6.

(iii) $N = N - N_1$.

(iv) If $N > 0$, Go to *Step* 2; otherwise continue

Step 6. [*Output Result*]
Output $a_{ki}^{(C_k,0)} = \sum_{j=0}^{i-1} a_{ki}^{(j)}$, $i \le k \le n$, $1 \le i \le n$, and terminate.

End Set-Ordering

The time complexity of the set-ordering procedure is $(n(n-1))/2$ scalar adds using n processors. An example illustrating the use of set-ordering procedure is given in Appendix C. Based on the above discussion and procedure, a parallelized version of the Cholesky factorization is summarized below:

Procedure Parallel-Cholesky-Factorization. This procedure is used to compute the lower triangular matrix $\mathbf{L}$ of a given $n{\times}n$ symmetric matrix $\mathbf{H(q)}$ $(\mathbf{H(q)} \equiv \mathbf{LL}^T)$.

Step 1. [*Compute* h_{ijk}] Compute

$$h_{ijk} = -\, h_{ij}h_{kj}\ ;\ 1 \le j \le i-1,\ \ i \le k \le n,\ 2 \le i \le n$$

parallelly using n processors.

Step 2. [*Compute* $\hat{l}_{ki}$] Call the Procedure Set-Ordering $(\mathbf{H}, \hat{l}_{ki})$ and obtain $\hat{l}_{ki}$, $i \le k \le n$, $2 \le i \le n$

Step 3. [*Compute Diagonal Elements h_{kk} Using n Processors*]

$$l_{11} \leftarrow \sqrt{h_{11}}\ ,\ \ l_{kk} \leftarrow \sqrt{\hat{l}_{kk}} \text{ for } 2 \le k \le n$$

Step 4. [*Compute Off-Diagonal Elements of* $\mathbf{L}$]

$$l_{k1} \leftarrow h_{k1}/h_{11} \quad \text{for} \quad 1 \le k \le n$$
$$l_{ki} \leftarrow \hat{l}_{ki}/h_{ii} \quad \text{for} \quad 2 \le i < k,\ 2 \le k \le n$$

Step 5. [*Output and Termination*]
Output l_{ki} for $i \le k \le n$, $1 \le i \le n$ and return.

End Parallel-Cholesky-Factorization.

The computational complexity of the Parallel-Cholesky-Factorization is analyzed below:

(a) The parallel evaluation of h_{ijk} in *Step 1* takes $(\lceil (n^2-1)/6 \rceil)$ scalar mults.

(b) The parallel evaluation of $\hat{l}_{ki}$ by the Set-Ordering method takes $\lceil (n^2-1)/2 \rceil$ scalar adds.

(c) The parallel evaluation of the diagonal elements l_{kk} and the off-diagonal elements l_{ki} in Step 3 and Step 4 takes one square root and $\lceil (n-1)/2 \rceil$ scalar mults respectively.

After performing the Parallel-Cholesky-Factorization procedure, a lower triangular matrix $\mathbf{L}$ is obtained. The linear system equation $\mathbf{H}(\mathbf{q})\ddot{\mathbf{q}}(t) = \mathbf{y}$ could be solved by the following two subsystems, that is, $\mathbf{Lx} = \mathbf{y}$ and $\mathbf{L}^T\ddot{\mathbf{q}}(t) = \mathbf{x}$. Fortunately, an efficient parallel algorithm exists (called the column-sweeping algorithm [10]) that can solve the upper (or lower) linear system in $2n - 1$ scalar multiplications and $n - 1$ additions. Hence, the total computational complexity of the parallel computation for solving the linear system $\mathbf{H}(\mathbf{q})\ddot{\mathbf{q}}(t) = \mathbf{y}$ is $\lceil (n^2-1)/6 \rceil + 4n + \lceil (n-1)/2 \rceil - 2$ scalar multiplications, $\lceil (n^2-1)/2 \rceil + 2n - 2$ scalar additions, and 1 square root.

2.5. Computing the Joint Acceleration Vector

The basic idea of the Composite Rigid-Body method is to find the elements of the upper right triangular matrix of $\mathbf{H}(\mathbf{q})$. Three important parameters, the composite mass M_j, the location of the composite center of mass $\mathbf{c}_j$, and the composite inertia $\mathbf{E}_j$, from links j through n, may be computed recursively. Next, the force/torque at joint j is propagated backward to obtain the force/torque at joint $(j-1), \cdots, 1$. The (i,j) component h_{ij} of the $\mathbf{H}(\mathbf{q})$ are then found by projecting the resultant joint force/torque onto the joint i axis of motion, where $1 \leq i \leq (j-1)$, that is, the column of the upper triangular matrix of $\mathbf{H}(\mathbf{q})$. The procedure is repeated n times to obtain all the elements of the upper triangular matrix of $\mathbf{H}(\mathbf{q})$. The procedure can be parallelized by applying the parallel algorithms discussed above and the joint acceleration vector $\ddot{\mathbf{q}}(t)$ can be solved by the parallel linear system solver.

Prior to evaluating the equations, some necessary parameters are given or evaluated in advance.

(a) The 3×3 rotation matrices ${}^{i-1}\mathbf{R}_i$, $i=1,2, \cdots, n$, which indicate the orientation of link i coordinates referenced to link $(i-1)$ coordinates, need to be evaluated in advance.

(b) ${}^i\mathbf{p}_i^*$ denotes the origin of link i coordinate frame from the origin of link $(i-1)$ coordinate frame, expressed with respect to link i coordinates. ${}^i\mathbf{s}_i$ denotes the location of the center of the mass of link i from the origin of link i coordinate frame, expressed with respect to link i coordinates. ${}^i\mathbf{J}_i$ denotes the inertia matrix of link i about its center of mass, expressed with respect to link i coordinates. Note that ${}^i\mathbf{p}_i^*$, ${}^i\mathbf{s}_i$, and ${}^i\mathbf{J}_i$ must be given in advance and are constants when referred to their own link coordinates.

(c) Let $\omega_0 = \dot{\omega}_0 = \mathbf{0}$, $\ddot{\mathbf{p}}_0 = [g_x, g_y, g_z]^T$ and $|\mathbf{g}| = 9.869621 m/s^2$. τ denotes the torques (forces) of each joint. $\mathbf{q}, \dot{\mathbf{q}}$ denote, respectively, the given joint positions and velocities.

(d) The parallel Newton-Euler computation is used to generate the bias vector $\mathbf{b}$. The position-dependent parameters $\mathbf{z}_i$, $\mathbf{p}_i^*$, $\mathbf{s}_i$ and $\mathbf{J}_i$ are used repeatedly in the Newton-Euler computation and other computations in the Composite Rigid-Body method. In order to avoid these redundant evaluations, these essential parameters are calculated in the initial step. A new parallel Newton-Euler subroutine which is similar to the parallel Newton-Euler subroutine discussed in Section 2.1 except the parameters $\mathbf{z}_i$, $\mathbf{p}_i^*$, $\mathbf{s}_i$, $\mathbf{J}_i$ are evaluated before the calculation starts has been developed, i.e., $NE1^{(P)}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}, \mathbf{z}_i, \mathbf{p}_i^*, \mathbf{s}_i, \mathbf{J}_i, \tau)$, where $\mathbf{q}$, $\dot{\mathbf{q}}$, $\ddot{\mathbf{q}}$, $\mathbf{z}_i$, $\mathbf{p}_i$, $\mathbf{s}_i$, $\mathbf{J}_i$ are known input vectors and τ is the resultant output. With this new parallel Newton-Euler subroutine, the computation of the composite-rigid-body method can be summarized in the following algorithm.

Procedure PCRBM (Parallel Composite-Rigid-Body Method). Given τ^*, $\mathbf{q}$, $\dot{\mathbf{q}}$, m_i, ${}^i\mathbf{p}_i$, ${}^i\mathbf{s}_i$, ${}^i\mathbf{J}_i$, and ${}^{i-1}\mathbf{R}_i$, for $1 \le i \le n$, this procedure computes the joint acceleration vector $\ddot{\mathbf{q}}(t)$ parallelly.

Step 1. Parallel compute

$$ {}^0\mathbf{R}_i = {}^0\mathbf{R}_{i-1}\,{}^{i-1}\mathbf{R}_i\,, \quad 1 \le i \le n \tag{13}$$

by calling the subroutine FOHRA.

Step 2. Parallel compute $\mathbf{z}_i$, $\mathbf{p}_i^*$, $\mathbf{s}_i$, and $\mathbf{J}_i$, according to

$$\mathbf{z}_i = {}^0\mathbf{R}_i\mathbf{z}_0\,,\ \mathbf{p}_i^* = {}^0\mathbf{R}_i\,{}^i\mathbf{p}_i^*\,,\ \mathbf{s}_i = {}^0\mathbf{R}_i\,{}^i\mathbf{s}_i \ \text{,and} \tag{14}$$

$$\mathbf{J}_i = {}^0\mathbf{R}_i\,{}^i\mathbf{J}_i\,{}^i\mathbf{R}_0 = {}^0\mathbf{R}_i\,{}^i\mathbf{J}_i({}^0\mathbf{R}_i)^T$$

where $1 \le i \le n$ and $\mathbf{z}_0 = (0\,,\,0\,,\,1)^T$

Step 3. Initialize $M_n = m_n$ and compute

$$M_j = M_{j+1} + m_j\ ,\ 1 \le j \le (n-1), \tag{15}$$

by calling the subroutine FOHRA.

Step 4. (i) Initialize

$$\mathbf{c}_n = \mathbf{s}_n + \mathbf{p}_n^* \tag{16}$$

(ii) Parallel compute

$$a_j = M_{j+1}/M_j, \tag{17}$$

$$\mathbf{b}_j = \frac{1}{M_j}(m_j(\mathbf{s}_j + \mathbf{p}_j^*) + M_{j+1}\mathbf{p}_j^*) \quad 1 \le j \le n-1,$$

(iii) Parallel compute

$$\mathbf{c}_j = a_j\mathbf{c}_{j+1} + \mathbf{b}_j, \quad 1 \le j \le n-1, \tag{18}$$

by calling the subroutine FOIHRA.

Step 5. (i) Initialize $\mathbf{E}_n = \mathbf{J}_n$

(ii) Parallel compute

$$\mathbf{b}_j = M_{j+1}[(\mathbf{c}_{j+1} + \mathbf{p}_j^* - \mathbf{c}_j)^T (\mathbf{c}_{j+1} + \mathbf{p}_j^* - \mathbf{c}_j)\mathbf{I}_{3\times3} - (\mathbf{c}_{j+1} + \mathbf{p}_j^* - \mathbf{c}_j)(\mathbf{c}_{j+1} + \mathbf{p}_j^* - \mathbf{c}_j)^T] + \mathbf{J}_j + m_j[(\mathbf{s}_j + \mathbf{p}_j^* - \mathbf{c}_j)^T (\mathbf{s}_j + \mathbf{p}_j^* - \mathbf{c}_j)\mathbf{I}_{3\times3} - (\mathbf{s}_j + \mathbf{p}_j^* - \mathbf{c}_j)(\mathbf{s}_j + \mathbf{p}_j^* - \mathbf{c}_j)^T] \quad (19)$$

(iii) Parallel compute

$$\mathbf{E}_j = \mathbf{E}_{j+1} + \mathbf{b}_j \ , \ 1 \le j \le n-1 \quad (20)$$

by calling the subroutine FOHRA.

Step 6. Parallel compute

$$\mathbf{F}_j = \mathbf{z}_{j-1} \times (M_j \mathbf{c}_j) \ ; \ \mathbf{N}_j = \mathbf{E}_j \mathbf{z}_{j-1} \ , \ 1 \le j \le n \quad (21)$$

Step 7. (i) Initialize

$$\mathbf{f}_{ij} = \mathbf{F}_j \ , \ 1 \le i \le j \ , 1 \le j \le n \quad (22)$$

(ii) Parallel compute

$$\mathbf{n}_{jj} = \mathbf{N}_j + \mathbf{c}_j \times \mathbf{F}_j \ , \ 1 \le j \le n \quad (23)$$

(iii) Parallel compute

$$\mathbf{b}_{ij} = \mathbf{p}_i^* \times \mathbf{F}_j \ , \ 1 \le i \le j-1 \ , 1 \le j \le n, \quad (24)$$

(iv) Parallel compute using the row-sweeping algorithm

$$\mathbf{n}_{ij} = \mathbf{n}_{(i+1),j} + \mathbf{b}_{ij} \ , \ 1 \le i \le j-1 \ , 1 \le j \le n, \quad (25)$$

Step 8. Parallel compute h_{ij}

$$h_{ij} = \begin{cases} \mathbf{z}_{i-1}^T \ \mathbf{n}_{ij} \ , \text{ if joint } i \text{ is rotational} \\ \mathbf{z}_{i-1}^T \ \mathbf{f}_{ij} \ , \text{ if joint } i \text{ is translational} \end{cases} \quad (26)$$

where $1 \le i \le j$, $1 \le j \le n$.

Step 9. Parallel compute the bias vector **b**

$$\mathbf{b} = \mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}}(t) + \mathbf{G}(\mathbf{q}) \quad (27)$$

by calling the subroutine $NE1^{(P)}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}} = \mathbf{0}, \mathbf{z}_i, \mathbf{p}_i^*, \mathbf{s}_i, \mathbf{J}_i, \tau)$ and let $\mathbf{b} = \tau$. Next, parallel compute $\mathbf{y} = \tau^* - \mathbf{b}$.

Step 10. Solve the system equation

$$\mathbf{H}(\mathbf{q})\ddot{\mathbf{q}}(t) = \mathbf{y} = \tau^{*} - \mathbf{b} \tag{28}$$

by the Parallel-Cholesky-Factorization algorithm.

Step 11. Termination and output the result $\ddot{\mathbf{q}}(t)$.

End PCRBM

Previously undefined terms, expressed in the base coordinates, are given as follows: m_j is the mass of link j, $\mathbf{f}_j$ is the force exerted on link j by link $j-1$, $\mathbf{n}_j$ is the moment exerted on link j by link $j-1$, M_j is the total mass of links j through n, $\mathbf{c}_j$ is the location of the center of mass of the composite rigid-body of links j through n with respect to the origin of link $j-1$ coordinates, $\mathbf{E}_j$ is the moment of inertia matrix of the composite system of links j through n, $\mathbf{F}_j$ is the total force exerted on the composite system of links j through n, $\mathbf{N}_j$ is the total moment exerted on the composite system of links j through n, $\mathbf{n}_{ij}$ is the moment exerted on joint i due to the motion of the composite system of links j through n, $\mathbf{f}_{ij}$ is the force exerted on joint i due to the motion of the composite system of links j through n.

The evaluation of the total computational complexity of the Parallel-Composite-Rigid-Body method can be derived as follows:

(a) The parallel evaluation of Eq. (13) which is a recursive matrix product form indicates $(27\lceil\log_2 n\rceil - 19)$ scalar multiplications and $(18\lceil\log_2 n\rceil - 14)$ scalar additions.

(b) Eq. (15) is a recursive scalar addition form and requires $\lceil\log_2 n\rceil$ additions. Eq. (20) is recursive matrix addition form and requires $9\lceil\log_2 n\rceil$ scalar additions. Eq. (18) is an inhomogeneous linear recurrence in vector form and requires $4\lceil\log_2 n\rceil - 1$ scalar multiplications and $3\lceil\log_2 n\rceil$ scalar additions.

(c) Eq. (25) is an upper triangular linear recurrence equation system and can be solved by applying the row-sweeping algorithm. It requires $3(n - 1)$ scalar additions.

(d) Eq. (27) is used to generate the bias vector $\mathbf{b}$ by calling the Parallel Newton-Euler subroutine and requires 84 scalar multiplications and $15\lceil\log_2 n\rceil+63$ scalar additions.

(e) Eq. (28) is used to solve the vector $\ddot{\mathbf{q}}(t)$ by calling the parallel linear system solver and requires $\lceil(n^2 - 1)/6\rceil + 4n + \lceil(n-1)/2\rceil - 2$ scalar multiplications, $\lceil(n^2 - 1)/2\rceil+2n - 2$ scalar additions, and 1 square root.

(f) The parallel evaluation of other equations can be calculated by simple parallel computations, yielding $9\lceil(n - 1)/2\rceil+48$ scalar multiplications and $3n + 5\lceil(n - 1)/2\rceil + 42$ scalar additions.

Combining the results of (a)-(f), the total computational complexity of the Parallel-Composite Rigid-Body method applied to an n-link manipulator results in $\lceil (n^2-1)/6 \rceil + 2n + 10\lceil (n-1)/2 \rceil + 31\lceil \log_2 n \rceil + 170$ scalar multiplications, $\lceil (n^2-1)/2 \rceil + 5n + 5\lceil (n-1)/2 \rceil + 45\lceil \log_2 n \rceil + 125$ scalar additions, and 1 square root. If $n=6$, it gives 334 mults, 328 adds, and 1 square root as compared with the complexity of the Composite-Rigid-Body method running on a uniprocessor [6]: 1627 mults and 1255 adds.

2.6. Triangular Array Processor for Cholesky's Factorization

Last section indicates that the bottleneck of the parallel computation of forward dynamics depends on factorizing the symmetric inertia matrix $\mathbf{H}(\mathbf{q})$ by Cholesky's method and the proposed parallel Cholesky factorization procedure has a time complexity of $O(n^2)$ by using $O(n)$ processors. It is possible to reduce the time complexity further if the Cholesky's method is implemented on VLSI array processors. Ahmed, Delosme, and Morf [12] described a triangular array of $n(n+1)/2$ CORDIC (COordinate Rotation DIgital Computer) processing elements for the implementation of hyperbolic Cholesky's method for a symmetric matrix with a computation time of $(2n-1)$ units. Later, Liu and Young [9] used $(n-1)n/2$ scalar multiply-and-add processors and n square root processors to compute the Cholesky's factorization procedure with a computation time of $O(n)$. The following triangular array processor for Cholesky's factorization is based on the modification of Liu and Young's scheme [9].

Equations (11) and (12) can be rewritten in a recursive procedure which is better for array processor implementation and may be expressed as:

Step 1: Initialization:

$$C_{ki}^{(1)} = h_{ki} \quad , \quad i \le k \le n \tag{29}$$

Step 2: Recursive Computation:

For $m = 1$ *Step* 1 *until* $(i-1)$, *Do*

$$C_{ki}^{(m+1)} = C_{ki}^{(m)} - h_{im}\, h_{km} \quad , \quad i \le k \le n \tag{30}$$

End_do

Step 3: Results:

$$l_{kk} = \sqrt{C_{kk}^{(k)}} \quad , \quad 1 \le k \le n \tag{31}$$

$$l_{ki} = C_{ki}^{(i)}\, h_{ii}^{-1} \quad , \quad i < k \le n$$

The above recursive procedure can be implemented on the triangular array processor as shown in Figure 1, which consists of two types of processing elements and latches. The circular cell P_{kk} (See Figure 2) performs the division and will perform the square root operation when it receives a flag signal which is denoted by the notation "*". At the same time, it also passes u_{in} upward and u_{in} will be stored in the latch L_{kk}^1. The square cell P_{ki} (See Figure 2) performs $C_{out} = C_{in} - u\, u_{in}$, and it performs $C_{out} = C_{in} - u^2$ when it receives a flag signal. C_{out} will be stored in the latch L_{ki}^2. It should be noted that h_{ki} will be stored in the internal register u of the cell P_{ki} at the appropriate cycle time when $(u_{in})_{ki} = h_{ki}$, $i \leq k$. Otherwise, it is then moved upward and stored in the latch above the cell.

Assuming that t_{ma} is the time for performing multiplication and addition, t_d is the time for performing division, and t_{sq} is the time for performing square root operation, it is known that the global system clock or cycle time for the synchronization of the array architecture will be $t_c = \max(t_{ma}, t_d, t_{sq})$. There are two input data streams coming from the stack memory modules and are piped one row (or column) deeper into the array in every cycle time. The input data streams from the bottom of the array processor provide the input data u_{in} of processing cells and will be stored in the internal register u of an assigned cell at an appropriate cycle. The input data streams from the left of the array processor are the initial values of the recursive computation. The value will be accumulated when the data propagates through the processing cells from left to right. That is, the recursive computation is executing and traveling from left to right through the processor array. After $(2n - 1)$ cycles, the input data streams from the bottom sweep through all the cells and are stored in the assigned cells. The flag signal (i.e. "*") will change the operation of assigned processing cells and be used to obtain the diagonal components l_{kk}. It is known that h_{44} and the flag signal are piped into the processing cell P_{41} at $2n$(=8) cycles simultaneously. Thus, $(C_{out})_{41} = h_{44} - h_{41}^2$ is computed and stored in the latch L_{41}^2. The computational activity then propagates to the neighboring cell P_{42}, which will execute $(C_{out})_{42} = (C_{out})_{41} - h_{42}^2 = h_{44} - h_{41}^2 - h_{42}^2$. The computational activity will propagate the next neighboring cell and so on. Once the data sweeps through the circular cell P_{44}, the square root operation is executed and the recursion is over. The total computation time is equal to $(3n - 1)$ cycles.

3. Parallel Conjugate Gradient Method

The conjugate gradient method is an iterative procedure for solving the joint accelerations and, at $O(n^2)$ time complexity, is theoretically the most efficient scheme given in [6]. The method requires an initial estimate for the joint accelerations and successive adjustments to these estimates will be made until they converge to the correct solution in less than n iterations. The detailed procedure of the conjugate gradient method can be found in Appendix B. A parallel algorithm is proposed here to improve the time complexity from $O(n^2)$ to $O(n)$ for multiplication operations and from $O(n^2)$ to $O(n\log_2 n)$ for addition operations. One observes that *Step* 5 in Appendix B, i.e., $\mathbf{t} = \mathbf{H}(\mathbf{q})\mathbf{u}_i$, can be evaluated by the Newton-Euler subroutine once for each iteration. However, the position-dependent parameters $\mathbf{z}_i$, $\mathbf{p}_i^*$, $\mathbf{s}_i$ and $\mathbf{J}_i$ will be evaluated in each iteration. Thus, in order to avoid these redundant evaluations, these parameters should be evaluated before the iteration starts, and we shall use the parallel Newton-Euler subroutine $NE1^{(P)}(\mathbf{q},\dot{\mathbf{q}},\ddot{\mathbf{q}}, \mathbf{z}_i, \mathbf{p}_i^*, \mathbf{s}_i, \mathbf{J}_i, \tau)$ for it, where $\mathbf{q}$, $\dot{\mathbf{q}}$, $\ddot{\mathbf{q}}$, $\mathbf{z}_i$, $\mathbf{p}_i$, $\mathbf{s}_i$, $\mathbf{J}_i$ are known input vectors and τ is the resultant output. For example, to evaluate $\mathbf{t} = \mathbf{H}(\mathbf{q})\mathbf{u}_i$ by subroutine $NE1^{(P)}$, one sets $\ddot{\mathbf{q}} = \mathbf{u}_i$, $\dot{\mathbf{q}} = \mathbf{0}$ and gravity constant $= 0$ in the subroutine $NE1^{(P)}$, and the resultant output τ will equal to the desired $\mathbf{t}$. The evaluation takes only 42 multiplications and $12\lceil \log_2 n\rceil+26$ additions. The proposed parallel conjugate gradient method consists of two parts, the linear recurrence part and the inner product part, and is described as follows:

Procedure PCGM (Parallel-Conjugate-Gradient Method). Given τ^*, $\mathbf{q}$, $\dot{\mathbf{q}}$, ${}^i\mathbf{p}_i^*$, ${}^i\mathbf{s}_i$, ${}^i\mathbf{J}_i$, and ${}^{i-1}\mathbf{R}_i$, for $1 \leq i \leq n$, this procedure computes $\ddot{\mathbf{q}}(t)$ parallelly based on the conjugate gradient method given in Appendix B.

Step 1. Parallel compute ${}^0\mathbf{R}_i = {}^0\mathbf{R}_{i-1}\ {}^{i-1}\mathbf{R}_i$, $1 \leq i \leq n$, by calling the subroutine FOHRA.

Step 2. Parallel compute $\mathbf{z}_i$, $\mathbf{p}_i^*$, $\mathbf{s}_i$, and $\mathbf{J}_i$, according to
$\mathbf{z}_i = {}^0\mathbf{R}_i\mathbf{z}_0$, $\mathbf{p}_i^* = {}^0\mathbf{R}_i\,{}^i\mathbf{p}_i^*$, $\mathbf{s}_i = {}^0\mathbf{R}_i\,{}^i\mathbf{s}_i$ and
$\mathbf{J}_i = {}^0\mathbf{R}_i\,{}^i\mathbf{J}_i\,{}^i\mathbf{R}_0 = {}^0\mathbf{R}_i\,{}^i\mathbf{J}_i({}^0\mathbf{R}_i)^T$.
where $1 \leq i \leq n$ and $\mathbf{z}_0 = [0,0,1]^T$

Step 3. Parallel compute $\mathbf{b} = \mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}}(t) + \mathbf{G}(\mathbf{q})$ by calling $NE1^{(P)}(\mathbf{q},\dot{\mathbf{q}}, \ddot{\mathbf{q}} = \mathbf{0}, \mathbf{z}_i, \mathbf{p}_i^*, \mathbf{s}_i, \mathbf{J}_i, \tau^{(0)})$

Step 4. Estimate $\mathbf{x}_0 = \ddot{\mathbf{q}}^{(0)}$ and compute each component of $\tau^{(1)} = \tau^* - \tau^{(0)}$ per processor, and let $\mathbf{u}_0 = \mathbf{w}_0 = \tau^{(1)}$.

Step 5. [*Starts the iteration*]
Parallel compute the inner product $e_i = \mathbf{w}_i^T\mathbf{w}_i$

Step 6. If $e_i = 0$, then stop; else continue.

Step 7. Parallel compute $\mathbf{t} = \mathbf{H}(\mathbf{q})\mathbf{u}_i$ by calling $NE1^{(P)}(\mathbf{q},\dot{\mathbf{q}}{=}\mathbf{0}, \ddot{\mathbf{q}}{=}\mathbf{u}_i, \mathbf{z}_i, \mathbf{p}_i^*, \mathbf{s}_i, \mathbf{J}_i, \tau)$ and let $\mathbf{x}_1 = \tau$.

Step 8. Parallel compute the inner product $\mathbf{u}_i^T\mathbf{t}$, and then $\gamma_i = e_i/\mathbf{u}_i^T\mathbf{t}$.

Step 9. Compute each component of $\mathbf{x}_{i+1} = \mathbf{x}_i + \gamma_i\mathbf{u}_i$ per processor, respectively.

Step 10. If $(i = m - 1)$, then stop; else continue.

Step 11. Compute each component of $\mathbf{w}_{i+1} = \mathbf{w}_i - \gamma_i\mathbf{t}$ per processor, respectively.

Step 12. Parallel compute the inner product $e_{i+1} = \mathbf{w}_{i+1}^T\mathbf{w}_{i+1}$.

Step 13. If $e_{i+1} = 0$, then stop; else continue.

Step 14. Compute $\beta_i = e_{i+1}/e_i$

Step 15. Compute each component of $\mathbf{u}_{i+1} = \mathbf{w}_{i+1} + \beta_i \mathbf{u}_i$ per processor.

Step 16. Set $i = i + 1$, and go to *Step* 5.

End PCGM

The evaluation of the total computational complexity of the PCGM algorithm can be derived as:

(a) The parallel computation of *Step* 1 by calling the subroutine FOHRA requires $(27\lceil \log_2 n \rceil - 19)$ scalar multiplications and $(18\lceil \log_2 n \rceil - 14)$ scalar additions.

(b) The parallel computation of $\mathbf{b}$ in *Step* 3 by calling $NE1^{(P)}$ requires 84 scalar multiplications and $15\lceil \log_n n \rceil + 63$ scalar additions. However, the parallel computation of $\mathbf{t} = \mathbf{H}(\mathbf{q})\mathbf{u}_i$ in *Step* 7 is much easier since the ignorance of the effects due to the velocity terms, the gravitation, and external forces and moments. It requires 42 scalar multiplications and $12\lceil \log_2 n \rceil + 26$ scalar additions.

(c) The parallel computation of the inner product between two n-vectors can be obtained in two steps: First, compute the product between components of both n-vectors per processor. Then, parallel compute the summation of those product terms by calling the subroutine FOHRA. So, the total parallel computation requires 1 scalar multiplication and $\lceil \log_2 n \rceil$ scalar additions. In the parallel algorithm, *Step* 5, *Step* 8, and *Step* 12 perform the inner product operation.

(d) It should be noted that the steps between 5 and 16 form an n-iteration loop, and the parallel computation of the steps inside the loop requires 49 scalar multiplications and $14\lceil \log_2 n \rceil + 30$ scalar additions. Since the loop is terminated after n times in the worst case, the total computation inside the loop, in general, requires $49n$ scalar multiplications and $14n\lceil \log_2 n \rceil + 30n$ scalar additions.

(e) *Step* 1, *Step* 2, and *Step* 3 are outside the loop and require $27\lceil \log_2 n \rceil + 124$ multiplications and $34\lceil \log_2 n \rceil + 87$ additions.

Based on the evaluations in (a)-(e), the total computational complexity of the parallel conjugate-gradient method is $49n + 27\lceil \log_2 n \rceil + 124$ scalar multiplications and $14n\lceil \log_2 n \rceil + 30n + 34\lceil \log_2 n \rceil + 87$ scalar additions. For a six-link PUMA manipulator, it takes 499 scalar multiplications and 621 scalar additions.

4. Conclusion

We have shown that the efficient computation of forward dynamics can be achieved by taking advantages of parallelism in the Composite Rigid-Body method and the Conjugate Gradient method. We developed an efficient parallel algorithm for the Composite Rigid-Body method with the time complexity of $O(n^2)$ with $O(n)$ processors. Further reduction of the order of time complexity was achieved by implementing the Cholesky's factorization procedure on array processors. This reduces the time complexity from $O(n^2)$ to $O(\log_2 n)$, but the number of processors is increased from n to $\frac{n(n+1)}{2}$. The second parallel algorithm, based on the conjugate gradient method, computes the joint accelerations with a time complexity of $O(n)$ for multiplication operation and $O(n \log n)$ for addition operation. For a small n (i.e. $n \leq 12$), the parallel computation of the Composite Rigid-Body method in an SIMD machine is found to be superior than the Conjugate Gradient method. The inherent sequential property of the Conjugate Gradient method makes it difficult to obtain the necessary speed-up for practical use. Both the parallel Composite Rigid-Body method with and without VLSI array processors and the Conjugate Gradient method are also tabulated in Table 1 for comparison.

Appendix A

The Composite Rigid-Body Method [6]

- Backward Recurrence

$$M_j = M_{j+1} + m_j \tag{A.1}$$

$$\mathbf{c}_j = \frac{1}{M_j}\{m_j(\mathbf{s}_j + \mathbf{p}_j^*) + M_{j+1}(\mathbf{c}_{j+1} + \mathbf{p}_j^*)\} \tag{A.2}$$

$$\begin{aligned}\mathbf{E}_j = \mathbf{E}_{j+1} &+ M_j[(\mathbf{c}_{j+1} + \mathbf{p}_j^* - \mathbf{c}_j)^T\cdot(\mathbf{c}_{j+1} + \mathbf{p}_j^* - \mathbf{c}_j)\mathbf{I}_{3\times 3} \\ &-(\mathbf{c}_{j+1} + \mathbf{p}_j^* - \mathbf{c}_j)(\mathbf{c}_{j+1} + \mathbf{p}_j^* - \mathbf{c}_j)^T] + \mathbf{J}_j \\ &+ m_j[(\mathbf{s}_j + \mathbf{p}_j^* - \mathbf{c}_j)^T\cdot(\mathbf{s}_j + \mathbf{p}_j^* - \mathbf{c}_j)\mathbf{I}_{3\times 3} \\ &- (\mathbf{s}_j + \mathbf{p}_j^* - \mathbf{c}_j)(\mathbf{s}_j + \mathbf{p}_j^* - \mathbf{c}_j)^T]\end{aligned} \tag{A.3}$$

where $1 \le j \le n - 1$, and $M_n = m_n$, $\mathbf{c}_n = \mathbf{s}_n + \mathbf{p}_n^*$, $\mathbf{E}_n = \mathbf{J}_n$

- Compute $\mathbf{F}_j$, $\mathbf{N}_j$

$$\mathbf{F}_j = \begin{cases}\mathbf{z}_{j-1} \times (M_j\mathbf{c}_j) & \text{, if joint } j \text{ is rotational} \\ M_j\mathbf{z}_{j-1} & \text{, if joint } j \text{ is translational}\end{cases} \tag{A.4}$$

$$\mathbf{N}_j = \begin{cases}\mathbf{E}_j\mathbf{z}_{j-1} & \text{, if joint } j \text{ is rotational} \\ \mathbf{0} & \text{, if joint } j \text{ is translational}\end{cases} \tag{A.5}$$

- $\mathbf{f}_{jj} = \mathbf{F}_j,\ \mathbf{n}_{jj} = \mathbf{N}_j + \mathbf{c}_j \times \mathbf{F}_j,\ 1 \le j \le n$ (A.6)

- Linear equation system

$$\begin{aligned}\mathbf{f}_{i,j} &= \mathbf{f}_{i+1,j} \\ \mathbf{n}_{i,j} &= \mathbf{n}_{i+1,j} + \mathbf{p}_i^* \times \mathbf{f}_{(i+1),j}\end{aligned} \qquad 1 \le i \le j-1,\ 1 \le j \le n \tag{A.7}$$

$$h_{ij} = \begin{cases}\mathbf{z}_{i-1}^T\,\mathbf{n}_{i,j}, & \text{, if joint } j \text{ is rotational} \\ \mathbf{z}_{i-1}{}^T\,\mathbf{f}_{i,j}, & \text{, if joint } j \text{ is translational}\end{cases} \tag{A.8}$$

where $1 \le i \le j$, $1 \le j \le n$.

Appendix B

The Conjugate Gradient Method [6]

```
Step 1.   Estimate solution $\mathbf{x}_0$
Step 2.   Set $i = 0$, $\mathbf{u}_0 = \mathbf{w}_0 = \tau - \mathbf{b}$
Step 3.   Set $e_i = \mathbf{w}_i^T \mathbf{w}_i$
Step 4.   If $e_i = 0$, then stop; else continue
Step 5.   Set $\mathbf{t} = \mathbf{H}(\mathbf{q})\mathbf{u}_i$
Step 6.   Set $\gamma_i = e_i / \mathbf{u}_i^T \mathbf{t}$
Step 7.   Set $\mathbf{x}_{i+1} = \mathbf{x}_i + \gamma_i \mathbf{u}_i$
Step 8.   If $(i = N - 1)$, then stop; else continue
Step 9.   Set $\mathbf{w}_{i+1} = \mathbf{w}_i - \gamma_i \mathbf{t}$
Step 10.  Set $e_{i+1} = \mathbf{w}_{i+1}^T \mathbf{w}_{i+1}$
Step 11.  If $e_{i+1} = 0$, then stop; else continue
Step 12.  Set $\beta_i = e_{i+1} / e_i$
Step 13.  Set $\mathbf{u}_{i+1} = \mathbf{w}_{i+1} + \beta_i \mathbf{u}_i$
Step 14.  Set $i = i+1$; go to Step 5.
```

Appendix C

An Example for Evaluating the Set-Ordering Method

Assume $\hat{l}_{ki} = h_{ki} + \sum_{j=1}^{i-1} h_{ijk}$, $i \le k \le n$, $2 \le i \le n$, where $n = 4$, and let $a_{ki}^{(0)} \leftarrow h_{ki}$; $a_{ki}^{(j)} \leftarrow h_{ijk}$ for $1 \le j \le (i-1)$, $i \le k \le 4$, $2 \le i \le 4$, then the corresponding sets $s_{ki} = \{a_{ki}^{(i-1)}, a_{ki}^{(i-2)}, \ldots, a_{ki}^{(0)}\}$, $i \le k \le 4$, $2 \le i \le 4$. The evaluation of $\hat{l}_{ki}$ by using the set-ordering technique can be described as follows:

Step 0 (i) Let $C_{ki} = 0$, $a_{ki}^{(0,0)} = a_{k,i}^{(0)}$ for $i \le k \le 4$, $2 \le i \le 4$

(ii) $N = n(n-1)/2 = 6$

(iii) $NA(S_{44}) = 3$, $NA(S_{43}) = NA(S_{33}) = 2$, $NA(S_{42}) = NA(S_{32}) = NA(S_{22}) = 1$

Step 1 [Set Ordering]

$$\begin{aligned}
S_{44}^{(1)} &= S_{44} = \{a_{44}^{(3)}, a_{44}^{(2)}, a_{44}^{(1)}, a_{44}^{(0)}\} \\
S_{43}^{(2)} &= S_{43} = \{a_{43}^{(2)}, a_{43}^{(1)}, a_{43}^{(0)}\} \\
S_{33}^{(3)} &= S_{33} = \{a_{33}^{(2)}, a_{33}^{(1)}, a_{33}^{(0)}\} \\
S_{42}^{(4)} &= S_{42} = \{a_{42}^{(1)}, a_{42}^{(0)}\} \\
S_{32}^{(5)} &= S_{32} = \{a_{32}^{(1)}, a_{32}^{(0)}\} \\
S_{22}^{(6)} &= S_{22} = \{a_{22}^{(1)}, a_{22}^{(0)}\}
\end{aligned}$$

Step 2 $N(=6) > n(=4)$, thus parallel compute the $n(=4)$ highest ordered sets $S_{44}^{(1)}, S_{43}^{(2)}, S_{33}^{(3)}, S_{42}^{(2)}$, respectively, and de-order the sets. Then, the results would be:

(i) $C_{44} = 1$, $C_{43} = 1$, $C_{33} = 1$, $C_{42} = 1$,

(ii) $NA(S_{44}) = 2$, $NS(S_{43}) = 1$, $NA(S_{33}) = 1$, $NS(S_{42}) = 0$

(iii) The resultant set in a descending order as:

$$\begin{aligned}
S_{44} &= \{a_{44}^{(3)}, a_{44}^{(2)}, a_{44}^{(1,0)}\} \\
S_{43} &= \{a_{43}^{(2)}, a_{43}^{(1,0)}\} \\
S_{33} &= \{a_{33}^{(2)}, a_{33}^{(1,0)}\} \\
S_{32} &= \{a_{32}^{(1)}, a_{32}^{(0)}\} \\
S_{22} &= \{a_{22}^{(1)}, a_{22}^{(0)}\} \\
S_{42} &= \{a_{42}^{(1,0)}\}
\end{aligned}$$

where $a_{44}^{(1,0)} = a_{44}^{(1)} + a_{44}^{(0)}$, $a_{43}^{(1,0)} = a_{43}^{(1)} + a_{43}^{(0)}$, $a_{33}^{(1,0)} = a_{32}^{(1)} + a_{33}^{(0)}$ and $a_{42}^{(1,0)} = a_{42}^{(1)} + a_{42}^{(0)}$.

Step 3 (i) It is known that $NA(S_{42}) = 0$, then $N_1 = 1$ and S_{42} will wait in output step.

(ii) $N \leftarrow N - N_1 \Rightarrow N = 5$

Step 4 Similarly, we order the remaindered sets and pick up n($= 4$) highest ordered sets as considered (because $N(= 5) > n(= 4)$). There are S_{44}, S_{43}, S_{33}, S_{32} which can be evaluated parallelly in the same procedure in Step 2. We have

(i) $C_{44} = 2, C_{43} = 2, C_{33} = 2, C_{32} = 1.$

(ii) $NA(S_{44}) = 1,\ NA(S_{43}) = 0,\ NA(S_{33}) = 0,\ NA(S_{32}) = 0$

(iii) The resultant sets in a descending order as

$$\begin{aligned} S_{44} &= \{a_{44}^{(3)},\ a_{44}^{(2,0)}\} \\ S_{22} &= \{a_{22}^{(1)},\ \ a_{22}^{(0)}\} \\ S_{43} &= \{\qquad a_{43}^{(2,0)}\} \\ S_{33} &= \{\qquad a_{33}^{(2,0)}\} \\ S_{32} &= \{\qquad a_{32}^{(1,0)}\} \end{aligned}$$

where, $a_{44}^{(2,0)} = a_{44}^{(2)} + a_{44}^{(1)} + a_{44}^{(0)}$, $a_{43}^{(2,0)} = a_{43}^{(2)} + a_{43}^{(1)} + a_{43}^{(0)}$, $a_{33}^{(2,0)} = a_{33}^{(2)} + a_{32}^{(1)} + a_{33}^{(0)}$, and $S_{32}^{(1,0)} = S_{32}^{(1)} + S_{32}^{(0)}$

Step 5 (i) It can be shown that $NA(S_{43}) = 0$, $NA(S_{33}) = 0$ and $NA(S_{32}) = 0$, then $N_1 = 3$ and S_{43}, S_{33}, S_{32} would wait in output step.

(ii) $N \leftarrow N - N_1 \Rightarrow N = 2$

Step 6 The remaindered sets now are S_{44}, S_{32}. In this case, $N(= 2) < n(= 4)$. Thus any two processors are active and assigned to evaluate S_{44}, S_{22}, respectively. And the resultant sets are $S_{44} = \{a_{44}^{(3,0)}\}$, $S_{22} = \{a_{22}^{(1,0)}\}$, where $a_{44}^{(3,0)} = a_{44}^{(3)} + a_{44}^{(2)} + a_{44}^{(1)} + a_{44}^{(0)}$ and $a_{22}^{(1,0)} = a_{22}^{(1)} + a_{22}^{(0)}$.

5. References

[1] K. S. Fu, R. C. Gonzalez, and C. S. G. Lee, *Robotics: Control, Sensing, Vision, and Intelligence,* McGraw-Hill, September 1986.

[2] J. Y. S. Luh, M. W. Walker, and R. P. Paul, "On-line Computational Scheme for Mechanical Manipulator," *Trans. ASME J. Dynam. Syst., Meas. and Contr.,* vol. 102, pp. 69-76, June 1980.

[3] C. S. G. Lee and P. R. Chang, "Efficient Parallel Algorithm for Robot Inverse Dynamics Computations," *IEEE Trans. Syst. Man, Cybern.,* vol. SMC-16, no. 4, pp. 532-542, July/August 1986.

[4] P. M. Kogge, "Parallel Solution of Recurrence Problems," IBM J. Res. Develop., vol. 18, pp. 138-148, Mar. 1974.

[5] P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Trans. Comput.,* vol. C-22, pp. 789-793, Aug. 1973.

[6] M. W. Walker and D. E. Orin, "Efficient Dynamic Computer Simulation of Robot Mechanisms," *Trans. ASME J. Dynam. Syst., Meas. and Contr.,* vol. 104, pp. 205-211, 1982.

[7] R. Featherstone, "The Calculation of Robot Dynamics Using Articulated-Body Inertia," *The Int'l J. of Robotics Res.,* vol. 2, no. 1, pp. 13-30, 1983.

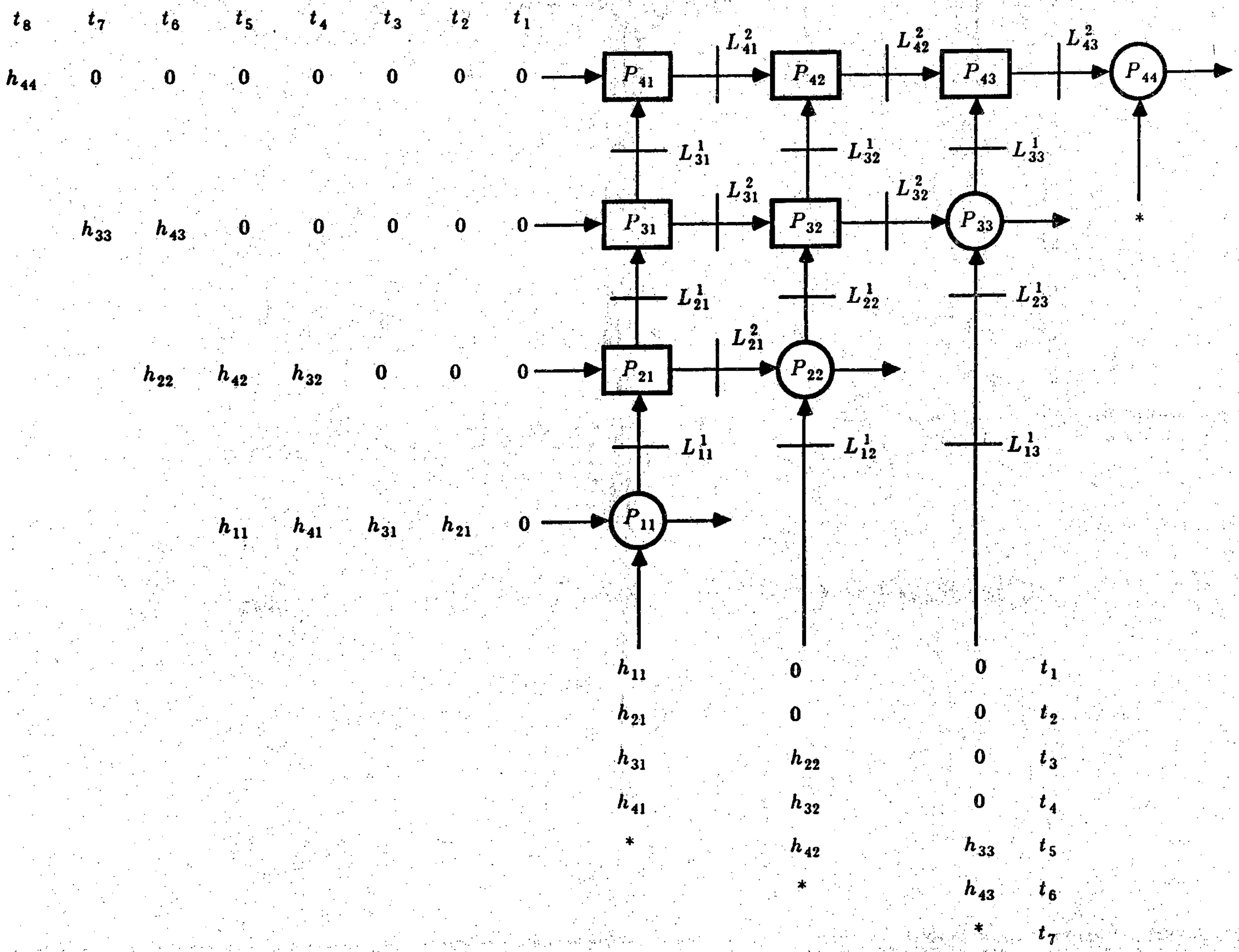
[8] H. T. Kung, "New Algorithm and Lower Bounds for the Parallel Evaluation of Certain Rational Expressions and Recurrence," *J. of Association for Computing Machinery,* vol. 23, no. 2, pp. 252-261, April 1976.

[9] P. S. Liu and T. Y. Young, "VLSI Array Design Under Constraint of Limited I/O Bandwidth," *IEEE Trans. Comput.,* vol. C-32, no. 12, pp. 1160-1170, Dec. 1983.

[10] D. J. Kuck, *The Structure of Computers and Computations,* volume 1, pp. 44-45, Wiley, 1978.

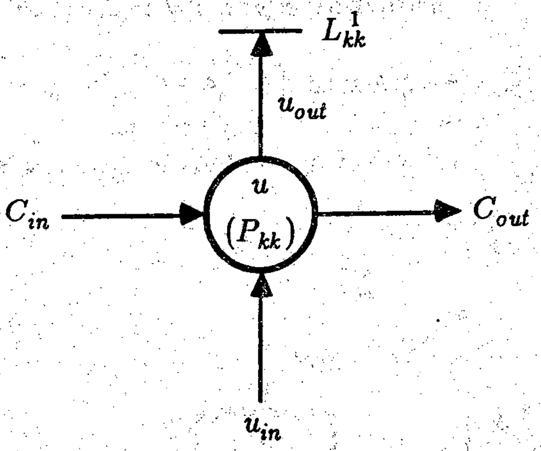
[11] J. R. Rice, *Matrix Computations and Mathematical Software,* McGraw-Hill, pp. 46-48, 1981.

[12] H. M. Ahmed, J. M. Delosme, and M. Morf, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," *IEEE Computer,* vol. 15, no. 1, pp. 65-82, January 1982.

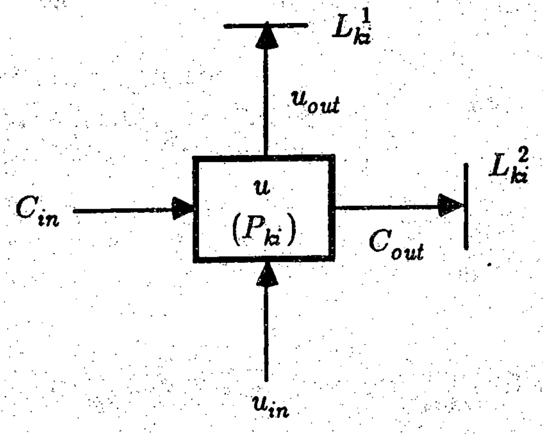


t_8 t_7 t_6 t_5 t_4 t_3 t_2 t_1
h_{44} 0 0 0 0 0 0 0
h_{33} h_{43} 0 0 0 0 0
h_{22} h_{42} h_{32} 0 0 0
h_{11} h_{41} h_{31} h_{21} 0
P_{41} P_{42} P_{43} P_{44}
P_{31} P_{32} P_{33}
P_{21} P_{22}
P_{11}
L^2_{41} L^2_{42} L^2_{43}
L^1_{31} L^1_{32} L^1_{33}
L^2_{31} L^2_{32}
L^1_{21} L^1_{22} L^1_{23}
L^2_{21}
L^1_{11} L^1_{12} L^1_{13}
*
h_{11} h_{21} h_{31} h_{41} *
0 0 h_{22} h_{32} h_{42} *
0 0 0 0 h_{33} h_{43} *
t_1 t_2 t_3 t_4 t_5 t_6 t_7

Figure 1. Triangular Array Processor for Cholesky's Factorization



Initial: $u = 0$
$u = u_{in}$, $u_{out} = C_{out} = null$, when the cell is activated.
In the remaining cycles:
If $u_{in} = *$, then $u_{out} \leftarrow u_{in}$, $C_{out} \leftarrow \sqrt{C_{in}}$, and $L_{kk}^1 \leftarrow u_{out}$.
If $u_{in} \neq *$, then $u_{out} \leftarrow u_{in}$, $C_{out} \leftarrow C_{in}/u$, and $L_{kk}^1 \leftarrow u_{out}$.
Note that P_{nn} performs the square root operation only.



Initial: $u = 0$
If $u_{in} = h_{ki}$, then $u \leftarrow u_{in}$, $u_{out} \leftarrow null$, $C_{out} \leftarrow C_{in} - u\ u_{in}$.
If $u_{in} = *$, then $u_{out} \leftarrow u_{in}$, $u_{out} \leftarrow C_{in} - u^2$, $L_{ki}^1 \leftarrow u_{out}$, and $L_{ki}^2 \leftarrow C_{out}$.
Otherwise, $u_{out} \leftarrow u_{in}$, $u_{out} \leftarrow C_{in} - u\ u_{in}$, $L_{ki}^1 \leftarrow u_{out}$, and $L_{ki}^2 \leftarrow C_{out}$.

Figure 2. Processing Cells
(a) Circular Processing Cell
(b) Square Processing Cell

Table 1. Comparison of the Number of Computations of Forward Dynamics Formulations and Parallel Forward Dynamics Formulations

Methods	Multiplication	Addition	Square Root	Number of Processors
Walker and Orin's Method 1	$\frac{1}{6}n^3+75\frac{1}{2}n^2$ $+114\frac{1}{2}n-22$ (3418)	$\frac{1}{6}n^3+55n^2$ $+82\frac{5}{6}n-11$ (2502)	0	1
Walker and Orin's Method 2	$\frac{1}{6}n^3+38\frac{1}{2}n^2$ $+151\frac{1}{2}n-22$ (2308)	$\frac{1}{6}n^3+28n^2$ $+109\frac{5}{6}n-11$ (1692)	0	1
Walker and Orin's Method 3	$\frac{1}{6}n^3+13\frac{1}{2}n^2$ $+192\frac{1}{3}n-49$ (1627)	$\frac{1}{6}n^3+8n^2$ $+165\frac{5}{6}n-64$ (1255)	0	1
Walker and Orin's Method 4	$76\frac{1}{2}n^2+12n-21$ (3435)	$56n^2+87n-6$ (2532)	0	1
Featherstone	$380n-198$† (2280)	$302n-173$† (1816)	0 0	1 1‡
Parallel Composite Rigid-Body Method	$\lceil\frac{(n^2-1)}{6}\rceil+10\lceil\frac{(n-1)}{2}\rceil+4n$ $+31\lceil\log_2 n\rceil+170$ (334)	$\lceil\frac{(n^2-1)}{2}\rceil+5n+5\lceil\frac{(n-1)}{2}\rceil+$ $45\lceil\log_2 n\rceil+125$ (328)	1 1	n 6
Parallel Composite Rigid-Body Method (with VLSI array processors)	$7n+9\lceil\frac{(n-1)}{2}\rceil$ $+31\lceil\log_2 n\rceil+169$ (331)	$8n+5\lceil\frac{(n-1)}{2}\rceil$ $+45\lceil\log_2 n\rceil+124$ (322)	1 1	$\frac{n(n+1)}{2}$ array processors n general-purpose processors 21 array processors and 6 g-p processors
Parallel Conjugate-Gradient Method	$49n+27\lceil\log_2 n\rceil+124$ (499)	$14n\lceil\log_2 n\rceil+30n+$ $34\lceil\log_2 n\rceil+87$ (621)	0 0	n 6

The number inside the parenthesis indicates number of computations when $n = 6$.

† In [7], Featherstone excludes the evaluations of computing the bias vector b. Here, we include the bias vector evaluations.

‡ This indicates that the method cannot be parallelized.