5-1-1987

Efficient Parallel Algorithms and VLSI Architectures for Manipulator Jacobian Computation

T. B. Yeung
Purdue University

C. S. G. Lee
Purdue University

Follow this and additional works at: <https://docs.lib.purdue.edu/ecetr>

Yeung, T. B. and Lee, C. S. G., "Efficient Parallel Algorithms and VLSI Architectures for Manipulator Jacobian Computation" (1987).
Department of Electrical and Computer Engineering Technical Reports. Paper 563.
<https://docs.lib.purdue.edu/ecetr/563>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Efficient Parallel Algorithms and VLSI Architectures for Manipulator Jacobian Computation

**T. B. Yeung
C. S. G. Lee**

**TR-EE 87-16
May 1987**

**School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907**

Efficient Parallel Algorithms and VLSI Architectures for Manipulator Jacobian Computation

T. B. Yeung and C. S. G. Lee

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

TR-EE 87-16

May 1987

Abstract

Real-time computations of manipulator Jacobian are examined for executing on uniprocessor computers, parallel computers, and VLSI pipelines. The characteristics of the Jacobian equations are found to be in the form of the first-order linear recurrence. The time lower bound of computing the first-order linear recurrence, and hence the Jacobian, is of order $O(N)$ on uniprocessor computers, and of order $O(\log_2 N)$ on parallel SIMD computers, where N is the number of degrees-of-freedom of the manipulator. The Generalized- k method, which achieves the time lower bound on uniprocessor computers, is derived to compute the Jacobian at any desired reference coordinate frame k from the base coordinate frame to the end-effector coordinate frame. We find that if the reference coordinate frame k is in the range $[3, N-4]$, then the computational effort is the minimum. To reduce the computational complexity from the order of $O(N)$ to $O(\log_2 N)$, we derive the parallel forward and backward recursive doubling algorithm to compute the Jacobian on parallel computers. Again, any reference coordinate frame k can be used, and the minimum computation occurs at $k = (N-1)/2$. To further reduce the Jacobian computation complexity, we design two VLSI systolic pipelined architectures. A linear VLSI pipe, which uses the least number of modular processors, takes $3N$ floating-point operations to compute the Jacobian, and a parallel VLSI pipe takes 3 floating-point operations. We also show that if the reference coordinate frame is selected at $k = (N-1)/2$, then the parallel pipe will require the least number of modular processors, and the communication paths are much shorter.

1. Introduction

The control of robot manipulators involves actuating appropriate joint motors to position and orient the manipulator hand along a planned trajectory. For some applications such as paint spraying and seam welding, it is desirable to command the manipulator hand to move along a desired Cartesian direction in a coordinated rate control. The motions of the various joint motors are combined and resolved into separately controllable hand motions along the world coordinate system. This implies that several joint motors must run simultaneously at different time-varying rates in order to achieve the desired coordinated hand rate motion along any world coordinate axis. This resolved motion concept enables the user to specify the direction and speed along any arbitrarily oriented path for the manipulator to follow, and it greatly simplifies the specification of the sequence of motions for completing a task.

In resolved motion control, one needs to determine how each infinitesimal joint motion affects the infinitesimal motion of the manipulator hand. One advantage of resolved motion is that there exists a linear mapping between the infinitesimal joint motion space and the infinitesimal hand motion space. This mapping is defined by the manipulator Jacobian. Mathematically, a Jacobian relates the derivatives of a vector space to the derivatives of another vector space. In robotics, the position of the manipulator hand relative to a fixed inertial coordinate system can be described by a set of 6 algebraic equations containing the joint variables $q_1, q_2, \dots, q_N$ as

$$\mathbf{x} = \mathbf{f}(\mathbf{q}) \quad (1)$$

where $\mathbf{x} \triangleq (p_x, p_y, p_z, \phi_x, \phi_y, \phi_z)^T$ is a 6-dimensional vector describing the Cartesian position and orientation of the manipulator hand with respect to the inertial coordinate frame, $\mathbf{q}$ is an N -dimensional joint-variable vector for an N -link manipulator, and the superscript "T" indicates matrix/vector transpose. Differentiating Eq. (1) with respect to time yields

$$\dot{\mathbf{x}} \triangleq \begin{bmatrix} \mathbf{v} \\ \boldsymbol{\omega} \end{bmatrix} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}} \quad (2)$$

where $\mathbf{v}$ and $\boldsymbol{\omega}$ are, respectively, the linear and angular velocities of the manipulator hand, $\dot{\mathbf{q}} \triangleq (\dot{q}_1, \dot{q}_2, \dots, \dot{q}_N)^T$ is the joint-velocity vector of the manipulator, and $\mathbf{J}(\mathbf{q})$ is the manipulator Jacobian and it is a function of joint variables. Thus, the manipulator Jacobian specifies the mapping from the joint velocities in the joint-variable space to the linear and angular velocities in the Cartesian space. Furthermore, the transpose of the Jacobian matrix relates static contact forces and moments to a set of joint torques as

$$\boldsymbol{\tau} = \mathbf{J}^T(\mathbf{q})\mathbf{F} \quad (3)$$

where $\mathbf{F} \triangleq (F_x, F_y, F_z, M_x, M_y, M_z)^T$ is a 6-dimensional static force/moment vector, and $\boldsymbol{\tau}$ is an N -dimensional joint-torque vector. Consequently, the study of the

Jacobian computation is an important aspect in robotic kinematics and control. This paper focuses on the real-time computations of manipulator Jacobian in uniprocessor computers, parallel computers, and VLSI pipelines.

Existing methods in computing the Jacobian are formulated to be computed by uniprocessor computers and can be classified into two categories. The first category exploits the linear recurrence characteristics of the Jacobian equations [12, 15, 18], and the second category consists of various other methods [2, 4, 14, 16, 17, 19]. The best computational order from these methods for an N -link manipulator is $O(N)$. Based on the linear recurrence characteristics of the Jacobian equations, Renaud [15], Waldron [18], and Orin and Schrader [12] all developed efficient computational schemes for the Jacobian to be computed on uniprocessor computers. All their methods share a common characteristics, and only deviate from each other by a different selection of the reference coordinate frame for computation. The reference coordinate frame is set such that all the vectors and the Jacobian computed are referred to that reference coordinate system. In particular, for an N -link robot manipulator, Waldron's method corresponds to the reference coordinate frame k being selected at the base coordinates $k = 0$, Renaud's method corresponds to $k = N/2$, and Orin and Schrader's method corresponds to selecting the reference coordinate frame at the end-effector coordinates $k = N$. They all have a computational order of $O(N)$ for an N -link manipulator.

Other methods for computing the Jacobian include Uicker [16], Groome [4], Whitney [19], Paul et al. [14], Vukobratovic and Potkonjak [17], and Fu et al. [2]. Uicker [16] obtained the Jacobian in terms of the differential change of transform elements. Groome [4] computed the Jacobian based on vector methods. Whitney [19] also used the vectorial approach and adopted the method to the standard form of manipulator coordinates. Paul et al. [14] combined the matrix method of Uicker and the vector representation of Groome to provide another approach to obtaining the Jacobian directly from the kinematic equations. Vukobratovic and Potkonjak [17] calculated the N individual columns of the Jacobian matrix from the base coordinates to the end-effector coordinates expressed with respect to the end-effector coordinates. In Fu et al. [2], the Jacobian is the direct proportional constant of the joint accelerations and the linear and angular accelerations of the Cartesian space by ignoring the joint velocity terms. Then the Jacobian elements can be "probed" by numerical methods. It can be a by-product of the computation from the Newton-Euler equations of motion in solving the inverse dynamics.

Recently, Orin et al. [13] developed pipeline and parallel algorithms for configuring a systolic array of processors to implement the Jacobian. $(N + 1)/2$ time units are taken to obtain the Jacobian for the pipeline algorithm. $2N$ processors are used with initiation rate equals to 2. The parallel algorithm, which is based on a "divide and conquer" strategy, reduces the computational order to $O(\log_2(N+1))$. Hybercube interconnection network is selected for connecting the processors.

The linear recurrence property of the Jacobian equations is obviously well suited to uniprocessor computers. Based on this property, this paper generalizes Waldron's, Renaud's, and Orin and Schrader's methods to develop the Generalized- k algorithm, which computes the Jacobian in the order of $O(N)$ at any given desired reference coordinate frame (k). We also find that, from the Generalized- k algorithm, the *coefficient* and the *order* are fixed with respect to any k ; however, the total number of computations is minimum if k is in the range of $[3, N-4]$ for an N -link manipulator.

Although the recursive structure of the Jacobian equations is well suited to uniprocessor computers, it is, however, not an efficient parallel processing method for new single-instruction-stream multiple-data-stream (SIMD) computers. In order to overcome the recurrence problem in the Jacobian, this paper presents a parallel forward and backward recursive doubling algorithm running on SIMD computers to yield the time lower bound of $O(\log_2 N)$. To further reduce the computation, two designs of VLSI systolic pipelines for implementing the Jacobian computation are considered. The first one is a linear VLSI pipe, which uses the least number of modular processors to compute the Jacobian. It takes $3N$ floating-point operations (flops) to complete the task. The second design is a parallel VLSI pipe, which takes the least number of time steps to compute the Jacobian. The time steps needed are 3 flops. Existing methods and the proposed methods for the computation of the Jacobian are tabulated in Tables 1 and 2 for comparison.

2. Notation and Jacobian Equations

To describe the translational and rotational relationship between adjacent links, a Denavit-Hartenberg matrix representation [1] for each link is used. An orthonormal coordinate frame system (x_i, y_i, z_i) is assigned to link i , where the z_i axis passes through the axis of motion of joint $i+1$. With this orthonormal coordinate frame, four parameters are used to characterize two successive coordinate systems: a_i , the common normal distance between the z_{i-1} and z_i axes; α_i , the twist angle measured between the z_{i-1} and z_i axes in a plane perpendicular to a_i ; d_i , a distance parameter measured between the x_{i-1} and x_i axes; and θ_i , a joint angle between the normals and measured in a plane normal to the joint axis. Once the link coordinate systems have been established for each link, a homogeneous transformation matrix, ${}^{i-1}A_i$, can easily be developed relating the i th coordinate frame to the $(i-1)$ th coordinate frame [2]

$${}^{i-1}\mathbf{A}_i = \begin{bmatrix} \cos \theta_i & -\cos \alpha_i \sin \theta_i & \sin \alpha_i \sin \theta_i & | & a_i \cos \theta_i \\ \sin \theta_i & \cos \alpha_i \cos \theta_i & -\sin \alpha_i \cos \theta_i & | & a_i \sin \theta_i \\ 0 & \sin \alpha_i & \cos \alpha_i & | & d_i \\ \hline 0 & 0 & 0 & | & 1 \end{bmatrix} \quad (4)$$

$$\triangleq \begin{bmatrix} {}^{i-1}\mathbf{R}_i & | & {}^{i-1}\mathbf{r}_i \\ \hline 0 & | & 1 \end{bmatrix}, \quad i = 1, 2, \dots, N$$

where ${}^{i-1}\mathbf{R}_i$ is the rotation matrix relating the orientation of link i coordinate frame with respect to link $i-1$ coordinate frame, ${}^{i-1}\mathbf{r}_i$ is the position vector from the origin of link $i-1$ to the origin of link i . In order to derive the Jacobian equations, we will consistently use the coordinate system shown in Figure 1 and its parameters are defined as follows:

- $\mathbf{O}_i$ is the origin of the i th coordinate frame;
- $\mathbf{z}_i$ is the unit vector along the $\mathbf{z}$ axis of the i th coordinate frame with reference to the inertial coordinate frame;
- λ_i is the joint indicator indicating whether joint $i+1$ is prismatic ($\lambda_i = 1$) or revolute ($\lambda_i = 0$);
- $\mathbf{J}_k$ is the Jacobian matrix with elements referenced to the k th coordinate frame;
- ${}^k\boldsymbol{\omega}_i$ is the angular velocity of link i in the Cartesian space expressed in the k th reference coordinate frame;
- ${}^k\mathbf{v}_i$ is the linear velocity of link i in the Cartesian space expressed in the k th reference coordinate frame;
- ${}^k\mu_i$ is an entry of $\mathbf{J}_k$ relating $\dot{q}_{i+1}$ to ${}^k\mathbf{v}_i$;

$${}^k\mathbf{v}_i = {}^k\mu_i \dot{q}_{i+1}, \quad \text{where} \quad {}^k\mu_i = \begin{bmatrix} {}^k\mu_{ix} \\ {}^k\mu_{iy} \\ {}^k\mu_{iz} \end{bmatrix}; \quad (5)$$

${}^k\beta_i$ is an entry of $\mathbf{J}_k$ relating $\dot{q}_{i+1}$ to ${}^k\boldsymbol{\omega}_i$;

$${}^k\omega_i = {}^k\beta_i \dot{q}_{i+1}, \text{ where } {}^k\beta_i = \begin{bmatrix} {}^k\beta_{ix} \\ {}^k\beta_{iy} \\ {}^k\beta_{iz} \end{bmatrix}; \quad (6)$$

${}^k\mathbf{p}_i$ is the position vector from the origin of link k to the origin of link i in the k th reference coordinate frame. From Figure 1,

$${}^k\mathbf{p}_i = {}^k\mathbf{p}_{i-1} - {}^k\mathbf{R}_i {}^i\mathbf{p}_i^*, \text{ for } i \geq k, \quad (7a)$$

or

$${}^k\mathbf{p}_i = {}^k\mathbf{p}_{i+1} + {}^k\mathbf{R}_{i+1} {}^{i+1}\mathbf{p}_{i+1}^*, \text{ for } i < k. \quad (7b)$$

In particular,

${}^i\mathbf{p}_i^*$ is the position vector from the origin of link i coordinate frame to the origin of link $i-1$ coordinate frame with respect to the link i coordinate frame; it can be shown that

$${}^i\mathbf{p}_i^* = \begin{bmatrix} -a_i \\ -d_i \sin \alpha_i \\ -d_i \cos \alpha_i \end{bmatrix}; \quad (8)$$

From Eqs. (2), (5), and (6), the Jacobian for an N -link manipulator can be expressed as

$$\mathbf{J}_k \triangleq \begin{bmatrix} {}^k\mu_0 & {}^k\mu_1 & \cdots & {}^k\mu_{N-1} \\ {}^k\beta_0 & {}^k\beta_1 & \cdots & {}^k\beta_{N-1} \end{bmatrix}_{6 \times N} \quad (9)$$

In this paper, we shall assume that ${}^i\mathbf{R}_{i+1}$, ${}^{i+1}\mathbf{p}_{i+1}^*$, and λ_i , for all i from 0 to $N-1$, are available. For a given robot manipulator, α_i , a_i and λ_i are fixed. θ_i and d_i are, respectively, joint variables for a revolute joint and a prismatic joint. These variables determine the entries of the Jacobian matrix of a manipulator.

Since a robot manipulator consists of links in serial, velocities propagate from one link to another. The angular velocity of link $i+1$ is that of link i plus a new rotational component at joint $i+1$ as in

$${}^i\omega_{i+1} = {}^i\omega_i + {}^i\mathbf{R}_{i+1} \dot{q}_{i+1} {}^i\mathbf{z}_i, \quad (10)$$

where ${}^i\mathbf{z}_i = (0, 0, 1)^T$ for $i = 0, 1, \dots, N-1$. By pre-multiplying a rotation matrix, ${}^{i+1}\mathbf{R}_i$, Eq. (10) can be expressed as

$${}^{i+1}\omega_{i+1} = \begin{cases} {}^{i+1}\mathbf{R}_i {}^i\omega_i + \dot{q}_{i+1} {}^i\mathbf{z}_i & , \text{ if link } i+1 \text{ is rotational} \\ {}^{i+1}\mathbf{R}_i {}^i\omega_i & , \text{ if link } i+1 \text{ is translational} \end{cases} \quad (11)$$

$$= {}^{i+1}\mathbf{R}_i {}^i\omega_i + (1 - \lambda_i) \dot{q}_{i+1} {}^i\mathbf{z}_i$$

Similarly, the linear velocity of link $i+1$ is the sum of the linear velocity of link i and the rotational velocity of link $i+1$,

$${}^{i+1}\mathbf{v}_{i+1} = \begin{cases} {}^{i+1}\mathbf{R}_i {}^i\mathbf{v}_i + {}^{i+1}\omega_{i+1} \times {}^{i+1}\mathbf{p}_{i+1}^* & , \text{ if link } i+1 \text{ is rotational} \\ {}^{i+1}\mathbf{R}_i {}^i\mathbf{v}_i + {}^{i+1}\omega_{i+1} \times {}^{i+1}\mathbf{p}_{i+1}^* + {}^{i+1}\mathbf{R}_i \dot{q}_{i+1} {}^i\mathbf{z}_i & , \text{ if link } i+1 \text{ is translational} \end{cases}$$

$$= {}^{i+1}\mathbf{R}_i {}^i\mathbf{v}_i + {}^{i+1}\omega_{i+1} \times {}^{i+1}\mathbf{p}_{i+1}^* + \lambda_i {}^{i+1}\mathbf{R}_i \dot{q}_{i+1} {}^i\mathbf{z}_i \quad (12)$$

From Eqs. (11), (12), (5), and (6), we have the following relations regarding the elements of the Jacobian:

$${}^k\mu_i = (1 - \lambda_i)({}^k\beta_i \times (-{}^k\mathbf{p}_i)) + \lambda_i({}^k\mathbf{R}_i \mathbf{z}_0). \quad (13)$$

and

$${}^k\beta_i = (1 - \lambda_i) {}^k\mathbf{R}_i \mathbf{z}_0, \quad (14)$$

where $\mathbf{z}_0 = (0, 0, 1)^T$. Hence, the Jacobian is

$$\mathbf{J}_k = \begin{bmatrix} (1-\lambda_0)({}^k\beta_0 \times (-{}^k\mathbf{p}_0)) + \lambda_0 {}^k\mathbf{R}_0 \mathbf{z}_0 & \dots & (1-\lambda_{N-1})({}^k\beta_{N-1} \times (-{}^k\mathbf{p}_{N-1})) + \lambda_{N-1} {}^k\mathbf{R}_{N-1} \mathbf{z}_0 \\ (1-\lambda_0) {}^k\mathbf{R}_0 \mathbf{z}_0 & \dots & (1-\lambda_{N-1}) {}^k\mathbf{R}_{N-1} \mathbf{z}_0 \end{bmatrix} \quad (15)$$

Equations (13) and (14) are the equations of the Jacobian calculation. In order to compute Eqs. (13) and (14), ${}^k\mathbf{R}_i$ and ${}^k\mathbf{p}_i$ have to be found in advance. Using the "chain rule," we see that

$${}^k\mathbf{R}_i = {}^k\mathbf{R}_{i-1} {}^{i-1}\mathbf{R}_i, \quad \text{for } i \geq k, \quad (16a)$$

$${}^k\mathbf{R}_i = {}^k\mathbf{R}_{i+1} {}^i\mathbf{R}_{i+1}^T, \quad \text{for } i < k. \quad (16b)$$

If we start from ${}^k\mathbf{R}_k = \mathbf{I}_{3 \times 3}$ (an identity matrix), ${}^k\mathbf{R}_{k+1}$, ${}^k\mathbf{R}_{k+2}$, ..., ${}^k\mathbf{R}_{N-1}$ can be found consecutively. Similarly, ${}^k\mathbf{R}_{k-1}$, ${}^k\mathbf{R}_{k-2}$, ..., ${}^k\mathbf{R}_0$ can also be found. In the same manner, ${}^k\mathbf{p}_i$ can be found from Eqs. (7a) and (7b) for all i , with ${}^k\mathbf{p}_k = \mathbf{0}_{3 \times 1}$. That is, ${}^k\mathbf{p}_{k+1}$, ${}^k\mathbf{p}_{k+2}$, ..., ${}^k\mathbf{p}_{N-1}$, and ${}^k\mathbf{p}_{k-1}$, ${}^k\mathbf{p}_{k-2}$, ..., ${}^k\mathbf{p}_0$ can be found one

after another. Equations (7a), (7b), (16a), and (16b) are called the first-order linear recurrence equations. An m th order recurrence equation is defined as the computation of the series, $x_1, x_2, \dots, x_N$, where $x_i = F_i(x_{i-1}, \dots, x_{i-m})$ for some function F_i . The recurrence is linear if it is of the following form:

$$x_i = a_i x_{i-1} + b_i, \quad i \geq 1, \quad (17)$$

where x_0, a_i , and b_i are constants. If b_i is zero for all i , then the equations are homogeneous; otherwise, it is called inhomogeneous. The Jacobian equations (Eqs. (13) and (14)) consist of a set of first-order homogeneous and inhomogeneous linear recurrence equations. Equations (16a), (16b), (7a), and (7b) constitute a computation bottleneck in computing the Jacobian. Lee and Chang [10] showed that the speed up in computing $x_i, i = 1, 2, \dots, N$, using a p -processor parallel computer, $1 \leq p \leq N$, is $O(k_1 \lceil N/p \rceil + k_2 \lceil \log_2 p \rceil)$, where k_1 and k_2 are constants. With uniprocessor computers, p is set to 1. Hence, the order of computational complexity of a linear recurrence problem running on uniprocessor computers is $O(N)$.

3. Jacobian Computation on Uniprocessor Computers

Based on the linear recurrence characteristics of the Jacobian equations, Renaud [15], Waldron [18], and Orin and Schrader [12] all developed efficient computational schemes for the Jacobian to be computed on uniprocessor computers. All their methods deviate from each other by a different selection of the reference coordinate frame for computation. The reference coordinate frame is selected such that all the vectors and the Jacobian components computed are referred to that reference coordinate system. Waldron's, Renaud's, and Orin and Schrader's methods have their reference frames set at $k = 0, N/2$, and N , respectively. The most efficient method is by Renaud, when the Jacobian components are referenced in their respective specified reference frame. However, if the reference frame is selected at the end-effector coordinates, Orin and Schrader's method is more efficient.

Since the methods proposed by Waldron, Renaud, and Orin and Schrader all achieve the time lower bound of $O(N)$ in computing the Jacobian on uniprocessor computers, we would like to generalize their methods to develop the Generalized- k algorithm, which also achieves the time lower bound of $O(N)$ running on uniprocessor computers. The objective of developing the Generalized- k algorithm is to determine which and why certain reference coordinate frame k is more efficient in computing the Jacobian, and an optimal reference coordinate frame k which requires the minimum number of mathematical operations in computing the Jacobian will be determined. The Generalized- k algorithm determines all the Jacobian matrices with the reference coordinate frame set at $k = 0, 1, 2, \dots, N$. As k varies from 0 to N , $N+1$ Jacobian matrices will be obtained. The methods proposed by Waldron, Renaud, and Orin and Schrader can be considered to be special cases of the Generalized- k algorithm. Using

the Generalized- k algorithm, we would like to determine an optimal reference coordinate frame k that requires the minimum number of mathematical operations in computing the Jacobian.

For a linear order of computational complexity ($O(N)$), the total number of computations in terms of the number of multiplications/divisions and additions/subtractions can be expressed by $aN + b$, where a and b are constants. We would like to determine an optimal k such that the coefficient a and/or the constant term b in the total number of calculations is minimum in computing the Jacobian. Furthermore, we would like to investigate the significant influence on the total number of calculations if we vary the twist angle (α_i in the link transformation matrix) of each link to 0 degree and ± 90 degrees.

The Jacobian computation requires the evaluation of Eqs. (7a), (7b), (16a), (16b), (13), and (14). Given a desired reference coordinate frame k , the following Generalized- k algorithm computes the Jacobian with respect to the reference frame k .

Algorithm Gen-k (Generalized- k algorithm). This algorithm determines the Jacobian matrix at a desired reference coordinate frame k (k is given).

Step 1: [Initialization]

$${}^k\mathbf{R}_k = \mathbf{I}_{3 \times 3};$$

$${}^k\mathbf{p}_k = \mathbf{0}_{3 \times 1};$$

Step 2: [Forward Recurrence]

FOR $i = k+1$ to $N-1$ Step 1 DO

$${}^k\mathbf{R}_i = {}^k\mathbf{R}_{i-1} {}^i\mathbf{R}_i; \quad (18)$$

$${}^k\mathbf{p}_i = {}^k\mathbf{p}_{i-1} - {}^k\mathbf{R}_i {}^i\mathbf{p}_i^*. \quad (19)$$

End

Step 3: [Backward Recurrence]

FOR $i = k-1$ down to 0 Step -1 DO

$${}^k\mathbf{R}_i = {}^k\mathbf{R}_{i+1} {}^i\mathbf{R}_{i+1}^T; \quad (20)$$

$${}^k\mathbf{p}_i = {}^k\mathbf{p}_{i+1} + {}^k\mathbf{R}_{i+1} {}^i\mathbf{p}_{i+1}^*; \quad (21)$$

End

Step 4: [Jacobian Computation]

FOR $i = 0$ to $N-1$ Step 1 DO

$${}^k\beta_i = (1 - \lambda_i) {}^k\mathbf{R}_i \mathbf{z}_0 \quad (22)$$

$${}^k\mu_i = (1 - \lambda_i)({}^k\beta_i \times (-{}^k\mathbf{p}_i)) + \lambda_i({}^k\mathbf{R}_i \mathbf{z}_0) \quad (23)$$

End

END Gen-k

From the Generalized- k algorithm, if the reference frame is set at some k , then the calculations can be split into forward and backward recursions. The forward recursion propagates rotation matrix and position vector from the reference coordinate frame k to the end-effector coordinate frame; whereas the backward recursion propagates rotation matrix and position vector from the reference coordinate frame k to the base coordinate frame. We shall now show that the computational order of the Generalized- k algorithm is of the order $O(N)$.

The multiplication of two general rotation matrices ${}^{i-1}\mathbf{R}_i {}^i\mathbf{R}_{i+1}$ requires 27 multiplications and 18 additions/subtractions. However, Eq. (4) shows that a general rotation matrix ${}^{i-1}\mathbf{R}_i$ has a zero element in its (3,1) entry. Thus, Eqs. (18) and (20) will only need $24N$ multiplications and $15N$ additions/subtractions in the first two FOR loops, neglecting constant terms. Similarly, Eqs. (19) and (21) will take $9N$ multiplications and $9N$ additions/subtractions in the first two FOR loops. Equation (22) does not involve any multiplications or additions since the third column of ${}^k\mathbf{R}_i$ is selected. The cross-product in Eq. (23) can be expressed as a matrix-vector multiplication and takes $6N$ multiplications and $3N$ additions/subtractions in the third FOR loop. Summing up, we have $39N$ multiplications and $27N$ additions/subtractions in the algorithm, which is obviously of the order $O(N)$.

The above computation is an upper bound on the total number of calculations needed. However, as the initial rotation matrix has a zero element in the matrix, the resulting computation needed is reduced if we can identify the zero elements in the matrix equation, since they are not operated on. As we set the reference frame k differently, the total number of computations in terms of multiplications and additions/subtractions will be different. Since the computation is extensive for us to carry out by hand, a computer program ORDER is written to count the total number of calculations at each reference coordinate frame k , $k = 0, 1, \dots, N$, and to assist us to determine an optimal reference coordinate frame k , $k \in [0, N]$. The complexity is measured in terms of the total number of multiplications and additions/subtractions. The program considers the entries of matrices and vectors to be composed of either zero or non-zero values. Zero valued elements will be identified. Multiplication/division or addition/subtraction of a zero operand will not be counted.

The program results are listed in Table 3. We observe that the order of the Jacobian computation is $O(N)$ and is independent of the reference coordinate frame k . In addition, the coefficient is the same, 39 in the case of multiplication and 27 in the case of addition/subtraction. They only differ by the constant terms.

Since the twist angles of most manipulators are either 0 degree or ± 90 degrees, we replace the general twist angle α_i , and set $\alpha_i = 0$ degree and $\alpha_i = \pm 90$ degrees in succession in the program ORDER. Similar results are obtained. The multiplication coefficient of the computation changes from 39 to 16 for $\alpha_i = 0$ degree, and to 30 for $\alpha_i = \pm 90$ degrees. The addition/subtraction coefficient changes from 27 to 9 for $\alpha_i = 0$ degree, and to 18 for $\alpha_i = \pm 90$ degrees.

From the above results and analysis, we have the following conclusions concerning the computation of the Jacobian on uniprocessor computers:

- (1) The optimal reference coordinate frame k for computing the Jacobian is found to be ranged from 3 to $N-4$, regardless which twist angles are selected;
- (2) If the desired reference coordinate frame is outside the range of the optimal k , it is less efficient to select an optimal reference coordinate frame k for computing the Jacobian, and then pre-multiply the Jacobian by a transformation matrix to reference it back to the desired reference coordinate frame.
- (3) The twist angle $\alpha_i = 0$ degree has greatly reduced the coefficient of the total number of computations. $\alpha_i = \pm 90$ degrees also reduces the coefficient but with a lesser extent. The result indicates that if there is a choice between the twist angle of $\alpha_i = 0$ degree and $\alpha_i = \pm 90$ degrees, it is computationally advantageous to select $\alpha_i = 0$ degree.
- (4) From Table 1, we see that the computation of the Jacobian using uniprocessor computers is bounded by $O(N)$ at best. Each respective coefficient is also identical. Further improvement of the computational order can only be exploited by parallelism.

4. Jacobian Computation on SIMD Computers

The best uniprocessor computation of the Jacobian is of order $O(N)$. In order to reduce the order of the time complexity, parallel SIMD computers must be exploited for real-time computations of the Jacobian. Our goal is to reduce the order of the time complexity from $O(N)$ to $O(\log_2 N)$ [†] using N processors.

Flynn [3] classified parallel computers into four distinct groups by their instruction stream and data stream replication. Instruction stream is a sequence of operations that are executed within a processor; data stream is the flow of data

[†]A logarithmic function will be assumed to be in base 2.

manipulated within the processor. The classifications are: 1) single-instruction-single-data-stream (SISD) computers, 2) single-instruction-multiple-data-stream (SIMD) computers, 3) multiple-data-single-instruction-stream (MISD) computers, and 4) multiple-instruction-multiple-data-stream (MIMD) computers.

Of the four classes of computers, only SIMD and MIMD computers are of interest for parallel computations. An SIMD computer is suitable for computing problems with regular structured data such as matrices. It is because data are to be distributed into different processors, and each of which will be operated by the same instruction. For example, in matrix addition of N $m \times 1$ vectors, the first element of all vectors will be put in the first processor; the second element of all vectors will be put in the second processor, and so on. An add instruction will add all data within each processor to form a new element of the sum at the same time. A uniprocessor will take $N(m-1)$ unit of time to complete, whereas an N -processor SIMD computer will need $(m-1)$ unit of time. The improvement is N -fold, which is an ideal enhancement by using parallelism. In practice, the speed up is much less than N -fold. It is because some processors are idle at a certain time due to memory access conflicts or communication between processors. In [5], it is shown that an N -processor machine (both SIMD and MIMD) will have estimates of the actual speed up ranged from a lower bound of $\log N$ to an upper bound of $N/(\log N)$. The lower bound is known as the Minsky's conjecture.

An MIMD computer is suitable to compute any general problems; usually problems that cannot be computed by SIMD computers will be considered on MIMD computers. It is obviously more powerful; however, more efforts and resources are needed because of the complexities of partitioning a problem and scheduling its subproblems. The selection between SIMD and MIMD computers is task dependent. We shall focus on SIMD computers for our Jacobian computation because we are able to arrange the data in a regular manner.

The physical structure of an SIMD computer can be viewed as a set of N processing elements (PEs), where each PE consists of a processor with its own memory and the operations performed by each processor involve at most two operands. A network connects each PE to some subset of the other PEs. The interconnection network provides a communication path between processors. For a fully-connected network, every processor is directly connected to every other processor and requires $N(N-1)/2$ bidirectional links between N processors so that data can be exchanged in one transfer, which is the ideal case of an interconnection network. Although for a large N , the cost of the links grows enormously, however, for a six-link robot arm, we need only seven processors $N = 7$ (one for the base) to handle all the necessary computations, and it requires only 21 bidirectional links. Even to allow for redundant robot arms, we can assume that N will not be a large number, say $N \leq 12$, and it requires only 66 bidirectional links. Thus, it is worthwhile to consider a fully-connected interconnection network for our SIMD computers. Furthermore, this

network provides three advantages: N is not confined to be a power of 2; any transfer takes just one move; and the interconnection network is much simpler. Our discussion on the Jacobian computation will focus on this fully-connected SIMD computer.

The Jacobian equations consist of a set of first-order homogeneous and inhomogeneous linear recurrence equations and independent equations. From the viewpoint of parallel computations, independent equations can be computed simultaneously in one time step. However, linear recurrence equations cannot be done in one single step. This constitutes a computational bottleneck in computing the Jacobian. Hyafil and Kung [6] showed that the speed up in computing the x_N of a linear recurrence equation using an idealized N -processor parallel computer is $(2/3)N + 1/3$. Thus, the time lower bound becomes $O(1)$. However, we need to compute $x_1, \dots, x_{N-1}$. Such requirement brings the time lower bound back to $O(\log N)$ according to [10]. The improvement from $O(N)$ to $O(\log N)$ is significant as shown in Figure 2.

An efficient technique for parallel solution of a large class of linear recurrence problems, called "recursive doubling" [7], is especially suited for SIMD computers. Recursive doubling involves the splitting of the computation of such problem into two subproblems. The evaluation of the subproblems can be performed simultaneously in two separate processors. By repeating the same procedure, each subproblem can further be split and spread over more processors. For an arbitrary N , there will be $(N+1)/2^k$ parallel operations at the k th splitting until $\lceil \log_2(N+1) \rceil$ splits. The resulting algorithm computes the entire series $x_0, x_1, \dots, x_N$ in time complexity of $O(\log_2 N)$ on an N -processor computer. Thus, the recursive doubling technique achieves the time lower bound and hence is the best solution for linear recurrence problems.

The computational bottleneck in the Generalized- k algorithm comes mainly from the forward and backward first-order linear recurrence equations (i.e. Steps 2 and 3). This set of forward and backward recurrence equations can be evaluated by applying the recursive doubling technique twice, one for the forward recursion and another for the backward recursion, to achieve the time lower bound. Further reduction in the computations can be accomplished by minimizing the *coefficient* of the logarithmic function of the time complexity. This is done by computing the forward and backward recurrence equations concurrently. A new algorithm, the forward and backward recursive doubling algorithm (Algorithm FABRD), is developed to compute the forward and backward recurrence equations concurrently. This method is depicted in Figure 3.

Let the base be the reference coordinate system k such that the forward recurrence starts from node k to node $N-1$, and the backward recurrence starts from node k back to node 0 (base coordinate system). Each part is itself an independent recursive doubling problem. We can rename the nodes according to their relative

positions to the start node k . Node $(k+i)$ will become $Q^+(i, k)$ and node $(k-i)$ will become $Q^-(i, k)$, where the superscripts "+" and "-" indicate, respectively, forward and backward recurrences. Nodes with Q^+ form the recurrence as pure recursive doubling. Nodes with Q^- form the similar recurrence. They all start from the node k . The forward recurrence equation becomes

$$Q^+(i, k) = Q^+(i-1, k) * X(k+i) \quad (24)$$

and the backward recurrence equation becomes

$$Q^-(i, k) = Q^-(i-1, k) * X(k-i) \quad (25)$$

where "*" indicates an associative operator. The forward and backward recursive doubling algorithm is summarized below.

Algorithm FABRD (Forward And Backward Recursive Doubling Algorithm). Given k , N , and $X(i)$ in $PE(i)$, $0 \leq i \leq N$, this algorithm computes the Jacobian by splitting the computations of the linear recurrence equations into forward and backward recurrences. $[\cdot]$ indicates computation occurs in the mask active, that is, the PE satisfying the mask $[\cdot]$ will perform the execution. $Y_i(l)$ is the i th sequence in the iteration at the $PE(l)$, and $Y(l)$ is the output.

Step 1 [Initialization]

$$a = \begin{cases} 0, & k \geq N-1 \\ \lceil \log_2(N-k-1) \rceil, & 0 \leq k < N-1 \end{cases}$$

$$b = \begin{cases} 0, & k \leq 0 \\ \lceil \log_2(k) \rceil, & 0 < k \leq N \end{cases}$$

$$s = \begin{cases} a, & a \leq b \\ b, & b < a \end{cases}$$

$$Y_0(l) = X(l) \quad [0 \leq l < N]$$

Step 2 [Computation of $Y_i(l)$]

FOR $i = 1$ to s step 1 DO

$$m = -2^{i-1}, \quad [2^{i-1} + k \leq l < N]$$

$$m = 2^{i-1}, \quad [0 \leq l \leq k - 2^{i-1}]$$

$$Y_i(l) = Y_{i-1}(l+m) * Y_{i-1}(l), \quad [2^{i-1} + k \leq l < N, 0 \leq l \leq k - 2^{i-1}]$$

$$Y_i(l) = Y_{i-1}(l), \quad [k-2^{i-1} < l < 2^{i-1}+k]$$

End_do

Step 3 [Remainder for $a \neq b$]

If $s = a$, then

$$Y(l) = Y_a(l), \quad [k < l < N]$$

FOR $i = (s+1)$ to b step 1 DO

$$m = -2^{i-1}, \quad [2^{i-1}+k \leq l < N]$$

$$m = 2^{i-1}, \quad [0 \leq l \leq k-2^{i-1}]$$

$$Y_i(l) = Y_{i-1}(l+m) * Y_{i-1}(l), \quad [2^{i-1}+k \leq l < N, 0 \leq l \leq k-2^{i-1}]$$

$$Y_i(l) = Y_{i-1}(l), \quad [k-2^{i-1} < l < 2^{i-1}+k]$$

End_do

$$Y(l) = Y_b(l), \quad [0 \leq l \leq k]$$

Else

$$Y(l) = Y_b(l), \quad [0 \leq l \leq k]$$

FOR $i = (s+1)$ to a step 1 DO

$$m = -2^{i-1}, \quad [2^{i-1}+k \leq l < N]$$

$$m = 2^{i-1}, \quad [0 \leq l \leq k-2^{i-1}]$$

$$Y_i(l) = Y_{i-1}(l+m) * Y_{i-1}(l), \quad [2^{i-1}+k \leq l < N, 0 \leq l \leq k-2^{i-1}]$$

$$Y_i(l) = Y_{i-1}(l), \quad [k-2^{i-1} < l < 2^{i-1}+k]$$

End_do

$$Y(l) = Y_a(l), \quad [k < l < N]$$

End_if

END FABRD

The Generalized- k algorithm can be implemented by the FABRD algorithm for an N -link manipulator to achieve a time lower bound of $O(\log_2 N)$, once the preferred

reference coordinate frame k is chosen. k can be any value in the range of 0 to N . We have to apply the FABRD algorithm twice to compute ${}^k\mathbf{R}_i$ and ${}^k\mathbf{p}_i$ in the Generalized- k algorithm. If $(N-k) > k$, then the forward recurrence will take more steps than the backward recurrence, and vice versa. The forward and backward recurrences are computed concurrently. Soon after the backward recurrence is finished, the forward recurrence will continue by itself until it is done. After ${}^k\mathbf{R}_i$ and ${}^k\mathbf{p}_i$ are calculated, ${}^k\beta_i$ and ${}^k\mu_i$ can be calculated all in two time steps. Detailed steps in computing the Jacobian are described in the following parallel forward and backward recursive doubling algorithm (Algorithm PFABRD).

Algorithm PFABRD (Parallel Forward And Backward Recursive Doubling). This algorithm computes the Jacobian by splitting the linear recurrence equations of the Jacobian into forward and backward recurrences.

Step 1. [Initialization] ${}^{i-1}\mathbf{R}_i$, λ_i , $-{}^i\mathbf{p}_i^*$ (for $i > k$), and ${}^i\mathbf{p}_i^*$ (for $i < k$) are first loaded into $PE(i)$ for $i = 0$ to N . k and N are loaded into the control unit. ${}^k\mathbf{R}_k = \mathbf{I}_{3 \times 3}$, and ${}^k\mathbf{p}_k = \mathbf{0}_{3 \times 1}$ are initialized in $PE(k)$. a , b , and s in Step 1 of the FABRD algorithm are computed in the control unit.

Step 2. [Compute ${}^k\mathbf{R}_i$] Steps 2 and 3 of the FABRD algorithm are used to compute ${}^k\mathbf{R}_i$ in parallel.

$$\text{Forward: } {}^k\mathbf{R}_i = {}^k\mathbf{R}_{i-1} {}^{i-1}\mathbf{R}_i; \quad (26a)$$

$$\text{Backward: } {}^k\mathbf{R}_i = {}^k\mathbf{R}_{i+1} {}^i\mathbf{R}_{i+1}^T; \quad (26b)$$

Step 3. [Compute ${}^k\mathbf{p}_i$] The product of ${}^k\mathbf{R}_i {}^i\mathbf{p}_i^*$ is computed. Steps 2 and 3 of the FABRD algorithm are used to compute ${}^k\mathbf{p}_i$ in parallel.

$$\text{Forward: } {}^k\mathbf{p}_i = {}^k\mathbf{p}_{i-1} + (-{}^k\mathbf{R}_i {}^i\mathbf{p}_i^*) \quad (27a)$$

$$\text{Backward: } {}^k\mathbf{p}_i = {}^k\mathbf{p}_{i+1} + ({}^k\mathbf{R}_{i+1} {}^{i+1}\mathbf{p}_{i+1}^*) \quad (27b)$$

Step 4. [Compute ${}^k\beta_i$] ${}^k\beta_i$ can be calculated in one time step for $i = 0$ to $N-1$.

$${}^k\beta_i = {}^k\mathbf{R}_i \mathbf{z}_0 (1 - \lambda_i) \quad (28)$$

Step 5. [Compute ${}^k\mu_i$] ${}^k\mu_i$ can be calculated in one time step for $i = 0$ to $N-1$.

$${}^k\mu_i = (1 - \lambda_i) {}^k\beta_i \times (-{}^k\mathbf{p}_i) + \lambda_i {}^k\mathbf{R}_i \mathbf{z}_0 \quad (29)$$

END PFABRD

Recall that in the FABRD algorithm, s is the smaller of $\lceil \log(N-k-1) \rceil$ and $\lceil \log k \rceil$. Forward and backward computations are executed concurrently in s time steps. If y is the larger of $\lceil \log(N-k-1) \rceil$ and $\lceil \log k \rceil$, then $y-s$ more steps will be

needed to complete the computations.

In the PFABRD algorithm, y time steps are taken in each Step 2 and Step 3. In Step 2, each time step executes 24 multiplications and 15 additions/subtractions. In Step 3, each time step executes 9 multiplications and 9 additions/subtractions. Combining Steps 2 and 3, we have $\max(33\lceil\log(N-k-1)\rceil, 33\lceil\log k\rceil)$ multiplications, and $\max(24\lceil\log(N-k-1)\rceil, 24\lceil\log k\rceil)$ additions/subtractions. Step 4 takes one time step to complete and no multiplications and additions/subtractions are needed. Step 5 also takes one time unit to complete with 6 multiplications and 3 additions/subtractions for the vector cross product.

Summing up the computation in Steps 2-5, we have

$$\begin{cases} (33\lceil\log(N-k-1)\rceil + 6) \text{ multiplications,} & \text{for } (N-k-1) > k, \\ (33\lceil\log k\rceil + 6) \text{ multiplications,} & \text{for } (N-k-1) \leq k; \end{cases}$$

and

$$\begin{cases} (24\lceil\log(N-k-1)\rceil + 3) \text{ additions/subtractions,} & \text{for } (N-k-1) > k, \\ (24\lceil\log k\rceil + 3) \text{ additions/subtractions,} & \text{for } (N-k-1) \leq k. \end{cases}$$

Obviously, when $k = (N-1)/2$, the total number of computations is at its minimum. That is, only $(33\lceil\log(N-1)\rceil - 27)$ multiplications and $(24\lceil\log(N-1)\rceil - 21)$ additions/subtractions are needed. This coincides with our results in the uniprocessor case. The least optimal case is when $k = N$. In that case, there are $(33\lceil\log N\rceil + 6)$ multiplications and $(24\lceil\log N\rceil + 3)$ additions/subtractions.

5. Jacobian Computation on VLSI Architectures

With the advent of Very Large Scale Integration (VLSI) technology [11], the rapid decrease in computational costs, reduced power consumption and physical size, and increase in computational power suggest that VLSI processors, which are configured and arranged based on the functional/data flow of the Jacobian equations, provide a better solution to computing the Jacobian. The main step in this approach is the design of high-level algorithms for computing the Jacobian in a VLSI chip. To gain the most from VLSI, parallel structures must be properly designed so that modular cells communicate only with their closest neighbors (i.e. local communication). In addition, efforts must be made to use simple and regular processors and to minimize power dissipation, I/O pin numbers, and communication delays. We shall only limit ourselves in the design level of the Jacobian computation algorithms. Systolic pipelining will be used, and two designs will be presented with comparisons. One design is to compute the Jacobian linearly using the least number of modular processors; the other design is to compute the Jacobian in parallel with the least time units. The first architecture takes $3N$ floating-point operations (flops) for an N -link manipulator, and the second architecture takes 3 flops.

Kung [8] in 1979 developed the concept of systolic architecture as an approach to design cost-effective, high-performance, special-purpose systems for a wide range of potential applications. In [8], systolic system is defined as a network of (identical) simple processors that circulate data in a regular fashion; and a systolic algorithm is one whose implementation leads to a systolic system. Systolic computations are characterized by the strong emphasis upon data movement, especially in pipelining.

A systolic system consists of a set of interconnected cells; each cell performs some simple operations. Information in a systolic system flows between cells in a pipeline manner, and communication with the outside world occurs only at the boundary cells. Advantages of using systolic systems include modular expansibility, simple and regular data and control flows, use of simple and uniform cells, elimination of global broadcasting and fan-in, fast response time, and being able to use each input data item a number of times (and thus achieving high computation throughput with only modest memory bandwidth) [9]. Criteria for evaluating the design of systolic structures include: multiple use of each input data, extensive concurrency, few types of simple cells, and simple and regular data and control flows [9].

A VLSI chip is commonly comprised of a few blocks of thousands of identical modular processors. A modular processor performs simple operations such as matrix to matrix multiplication, cross product of two vectors, vectors addition, or simply passing of data. In our implementation of VLSI for the Jacobian computation, a few modular processors will be needed. Arithmetic computations can be grouped into matrix to matrix products (MMP), matrix to vector products (MVP), vector cross products (VCP), and vector to vector additions (VVA). All of the MMPs, MVPs, VCPs, and VVAs can be built based on simple modular processors (MPs). Each MP has three processing units which can perform scalar addition and multiplication simultaneously. It can be used to evaluate the operations of two 3×1 vectors such as vector dot product and vector addition. We shall assume one scalar multiplication takes the same processing time as one scalar addition/subtraction in one floating-point operation (flop). Hence, a vector dot product will take three flops, one multiplication and two additions. Table 4 shows the number of MPs and flops for each of the MMP, MVP, VCP, and VVA. We will use these MMPs, MVPs, VCPs, and VVAs as building modular cells to complete our design of the Jacobian computation in a VLSI chip.

5.1. Linear VLSI Pipeline Design

Figure 4 shows the data flow and systolic array implementation on the linear computation of the Jacobian (cf. the Generalized- k algorithm). In this linear pipeline design, five cells are needed. Cell 1 is a MMP. It computes Eqs. (18) and (20) in 3 flops. Cell 2 is a MVP, which computes part of Eqs. (19) and (21) in 3 flops. Cell 3 is a 2-to-1 multiplexer in which λ_i is a switch control. If $\lambda_i = 0$, input A will be used; if $\lambda_i = 1$, input B will be used. If input B is used, then a MVP will be used to evaluate

the input data; otherwise the output is the data at input A . Cell 3 computes Eq. (22) in 2 flops. Cell 4 is a VVA, which completes calculation of Eqs. (19) and (21) in 1 flop. Cell 5 is another 2-to-1 multiplexer with λ_i as a switch control. If input B is used, a VCP will be used. Cell 5 computes Eq. (23) in 2 flops. There is a time delay cell, indicated by a dotted square, to synchronize the data flow.

Summing up the processors used in the above 5 cells, 1 MMP, 2 MVPs, 1 VCP, and 1 VVA are used. The critical path of the linear pipe starts from Cell 1 through Cell 2, Cell 4, and to Cell 5. Each cell takes 3 flops to get through. From the critical path of the pipe, it takes twelve flops to fill the pipe (i.e. initial delay), and once the pipe is full, each successive output takes three flops. $3N$ flops are needed to compute the Jacobian, since at one time, only one entry of ${}^k\mu_i$ and ${}^k\beta_i$ are obtained. Data for the forward recurrence are entered followed by the data for the backward recurrence into the pipe. There is no distinction of where the reference coordinate frame k should be in the evaluation of efficiency. This means that the performance of the linear pipe is independent of the selection of the reference coordinate frame k .

According to the criteria described above, our linear pipe design satisfies three of the four criteria, namely multiple use of each input data, a few types of simple cells, and simple and regular data and control flows. Extensive concurrency criterion cannot be satisfied inherently because of a linear pipe design.

5.2. Parallel VLSI Pipeline Design

In order to exploit extensive concurrency, the design of a parallel pipe is performed. Figure 5 shows a block diagram for a VLSI design on the parallel computation of the Jacobian using the PFABRD algorithm. There are four major blocks in the design. Block 1 computes Step 2 of the PFABRD algorithm; block 2 computes Step 3; block 3 computes Step 4, and ${}^k\beta_i^* = {}^k\mathbf{R}_i \mathbf{z}_0$, by selecting the third column of ${}^k\mathbf{R}_i$; and finally block 4 computes Step 5. To apply the PFABRD algorithm to block 1 and block 2, we replace each node in Figure 3 by an appropriate modular cell.

Block 1 is shown in Figure 6. Here, darkened squares represent processor cells, and open squares represent a simple pass of data. In this structure, there are $N-2$ rows and $\max(\lceil \log_2 k \rceil, \lceil \log_2(N-1-k) \rceil)$ columns. For each column from left to right, open squares increase in the multiple of 2. Each darkened square is an MMP. Block 2, shown in Figure 7, is similar to block 1, except that a column of circles are added to the left, and each darkened square is a VVA. Darkened circles are MVPs. Block 3 and block 4 are shown in Figures 8 and 9 respectively. Each square in these two figures represent a 2-to-1 multiplexer. λ_i is a switch control. When $\lambda_i = 0$, the top input will be used; when $\lambda_i = 1$, the bottom input will be used. In Figure 8, when the top input is selected, the data will pass through an MVP. In Figure 9, when the top input is selected, the data will pass through a VCP. In both cases, when the bottom input is selected, the output is the data at the input.

Let us define three important parameters for analyzing the components and performance of the parallel VLSI pipeline design.

$$m1 \triangleq 2^{\lceil \log_2(N-1-k) \rceil}, \quad k < N-1 \quad (30)$$

$$m2 \triangleq 2^{\lceil \log_2 k \rceil}, \quad k > 0 \quad (31)$$

$$m3 \triangleq \max(\lceil \log_2 k \rceil, \lceil \log_2(N-1-k) \rceil) \quad (32)$$

From Figure 6, it can be verified that the total number of MMPs is given by

$$\begin{aligned} m4 = & m1(\log_2 m1 - 1) + (N-1-k-m2)\lceil \log_2(N-1-k) \rceil + 1 \\ & + m2(\log_2 m2 - 1) + (k-m2)\lceil \log_2 k \rceil + 1 \end{aligned} \quad (33)$$

Block 1 needs $m4$ MMPs; block 2 needs $m4$ VVAs and $(N-1)$ MVPs; block 3 needs N MVPs; and block 4 needs N VCPs. From Table 3, the total number of MPs to implement the system shown in Figure 5 is $9N + 4m4 - 3$. The critical path of the parallel pipe starts from block 1, through block 2, to block 4. From the critical path, it takes $(6m3 + 6)$ flops to fill the pipe as the initial delay. Once the pipe is full, each successive output takes three flops. Hence, the Jacobian can be obtained in three flops whenever the pipe is full.

From Eq. (32), it is clear that the initial delay time will be minimum when the reference coordinate frame is selected at $k = (N-1)/2$. The minimum number of MPs required is when $k = (N-1)/2$, derived by taking partial derivative of Eq. (33) with respect to k . In addition, when $k = (N-1)/2$, the data flow from one cell to another is exactly one-half shorter than when $k = N$ or $k = 0$ in block 1 and block 2. This is very important in the VLSI design. The fact that $k = (N-1)/2$ is the best coincides with the results in sections 3 and 4.

6. Conclusion

To investigate real-time Jacobian computation, we proceed by determining an optimal reference coordinate frame k for efficient computation in uniprocessor computers, then extend the concept to be computed by parallel SIMD computers, and finally design two VLSI pipelines. The order of complexity of the computation of the Jacobian and the number of processors needed in various methods are summarized in Tables 1 and 2. Summarizing the results, we have achieved the followings:

- (1) By analyzing the characteristics of the Jacobian equations explicitly, we pin-point that the equations are of the first-order linear recurrence. The time lower bound of computing the Jacobian using uniprocessor computers is $O(N)$.
- (2) The Generalized- k algorithm, which achieves the time lower bound for uniprocessors, is then derived to compute the Jacobian. The method allows the user to

compute the Jacobian at any desired reference coordinate frame k from the base coordinates to the end-effector coordinates. The optimal computation occurs when the reference coordinate frame k is set in the range $[3, N-4]$ for an N -link manipulator.

- (3) We show that if given a choice of any twist angle α_i to be ± 90 degrees or 0 degree, it is computationally more efficient to pick the twist angle to be 0 degree.
- (4) In order to reduce the computational order of $O(N)$, parallel computation of the Jacobian in SIMD computers is necessary. The time lower bound of parallel computation is shown to be $O(\log_2 N)$. A recursive doubling algorithm can complete the Jacobian computation in the order of $O(\log_2 N)$. To speed up the process 2-fold at the best, we derive the parallel forward and backward recursive doubling (PFABRD) method, which also achieves the time lower bound of $O(\log_2 N)$. The algorithm splits the computation into the forward and backward recursions. Both recursions are scheduled to be processed concurrently to enhance the speed up. Again, any reference coordinate frame k can be used, and the minimum computation is found when k is selected at $(N-1)/2$.
- (5) To further speed up the Jacobian computation, VLSI technology is utilized. Two systolic pipelined architectures are designed. The first one is a linear VLSI pipe, which uses the least number of modular processors; however, $3N$ flops are needed to compute the Jacobian.
- (6) Further reduction in computing the Jacobian can be achieved through designing a parallel VLSI pipe, which just takes 3 flops to compute the Jacobian. When the reference coordinate frame k is selected at $(N-1)/2$, it requires the least number of modular processors, and the communication links between the processors are much shorter.

7. References

- [1] Denavit, J., and Hartenberg, R., "A Kinematic Notation for Low-Pair Mechanism Based on Matrices," *Trans. ASME Journal of Applied Mechanics*, Vol. 22, pp. 215-221, 1955.
- [2] Fu, K. S., Gonzalez, R. C., and Lee, C. S. G., *Robotics: Control, Sensing, Vision, and Intelligence*, McGraw-Hill, 1987.
- [3] Flynn, M. J., "Very High-Speed Computing Systems," *Proc. of the IEEE*, pp. 1901-9, 1966.
- [4] Groome, R. C. Jr., "Force Feedback Steering of a Teleoperator System," M.S. Thesis, Massachusetts Institute of Technology, Cambridge, MA, 1972.
- [5] Hwang, K., and Briggs, F. A., *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.
- [6] Hyafil, L., and Kung, J. T., "The Complexity of Parallel Evaluation of Linear Recurrences," *Journal of Association for Computing Machinery*, Vol. 24, No. 3, pp. 513-521, July 1977.
- [7] Kogge, P. M., and Stone, H. S., "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Trans. on Computers*, Vol. C-22, No. 8, pp. 786-792, August 1973.
- [8] Kung, H. T., "Let's Design Algorithms for VLSI Systems," *Proc. Conf. Very Large Scale Integration: Architecture, Design, Fabrication*, California Institute of Technology, pp. 65-90, January 1979.
- [9] Kung, H. T., "Why Systolic Architectures?" *IEEE Comput.*, pp. 37-46, January 1982.
- [10] Lee, C. S. G., and Chang, P. R., "Efficient Parallel Algorithm for Robot Inverse Dynamics Computation," *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. SMC-16, No. 4, pp. 532-542, July/August 1986.
- [11] C. Mead and L. Conway, *Introduction to VLSI systems*, Addison-Wesley, 1980.
- [12] Orin, D. E., and Schrader, W. W., "Efficient Computation of the Jacobian for Robot Manipulators," *The International Journal of Robotics Research*, Vol. 3, No. 4, pp. 66-75, Winter 1984.
- [13] Orin, D. E. et al., "Systolic Architectures for Computation of the Jacobian for Robot Manipulators," in *Computer Architectures for Robotics and Automation*, (J. H. Graham, editor) Gordon and Breach Science Publishers, pp. 39-68, 1987.
- [14] Paul, R. P., Shimano, B., and Mayer, G. E., "Differential Kinematic Control Equations for Simple Manipulators," *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. SMC-11, No. 6, pp. 456-460, June 1981.
- [15] Renaud, M., "Geometric and Kinematic Models of a Robot Manipulator: Calculation of the Jacobian Matrix and its Inverse," *Proc. 11th International Symp. Industrial Robots*, pp. 757-763, Oct. 1981.

- [16] Uicker, J. J. Jr., "Dynamic Force Analysis of Spatial Linkages, ASME Paper, No. 66-Mech-1, Mechanisms Conf., Oct. 1966.
- [17] Vukobratovic, M. and Potkonjak, V., "Contribution of the Forming of Computer Methods for Automatic Modeling of Spatial Mechanisms Motions," *Mechanism and Machine Theory*, Vol. 14, pp. 179-200, 1979.
- [18] Waldron, K. J., "Geometrically Based Manipulator Rate Control Algorithms," *Mechanism and Theory*, Vol. 17, No. 6, pp. 379-385, 1982.
- [19] Whitney, D. E., "The Mathematics of Coordinated Control of Prosthetic Arms and Manipulators," *Trans. ASME. J. Dynamic Syst., Measurement, Contr.*, pp. 303-309, December 1972.
- [20] Yeung, T. B., "Efficient Parallel Algorithms and VLSI Architectures for Manipulator Jacobian Computation," MSEE Thesis, School of Electrical Engineering, Purdue University, West Lafayette, Indiana, May 1987.

Table 1 Comparison of Various Methods for Jacobian Computation

Methods	Reference Coordinate Frame k	Multiplications	Additions/Subtraction	No. of Processors
1. Vukobratovic/Potkonijak ¹	N	$10N^2-15N+9$	N^2+7N-2	1
2. Paul et al. ¹	N	$30N-18$	$14N-15$	1
3. Fu et al.	0	$12N^2+12N$	$\frac{19}{2}(N^2+N)$	1
4. Waldron ²	0	$30N-61$	$18N-42$	1
5. Orin and Schrader ²	N	$30N-44$	$18N-33$	1
6. Renaud ²	$N/2$	$30N-105$	$18N-75$	1
7. Generalized- k ^{2,3}	any optimal $k \in [3, N-4]$	$30N-105$	$18N-75$	1
8. PFABRD ³	any optimal $k = \lceil (N-1)/2 \rceil$	$33\lceil \log_2(N-1) \rceil - 27$	$24\lceil \log_2(N-1) \rceil - 21$	N

¹ Order quoted from the reference [12].

² Twist angle α_i is set to ± 90 degrees.

³ The optimal k case is quoted.

Table 2 Comparison of VLSI Architectures for Jacobian Computation

Methods	Reference Coordinate Frame k	No. of Processors	Computation Time	Initial Delay	Time-Processor Product
1. Orin et al. [13] linear pipe	N	$O(N)$	$(N+1)/2$	$O(1)$	$O(N)$
2. Orin et al. [13] parallel pipe	N	N	$\lceil \log_2(N+1) \rceil$	$\lceil \log_2(N+1) \rceil$	$O(N \log N)$
3. Yeung and Lee linear pipe	$0 \leq k \leq N$	19	$3N$	12	$O(N)$
4. Yeung and Lee parallel pipe	$0 \leq k \leq N$	$O(N \log_2 N)$	3	$\lceil \log_2(N+1)/2 \rceil$	$O(N \log N)$

Table 3 Number of Computations at Each Reference Coordinate Frame of an N -link Manipulator, (general twist angle α_i).

Reference Coordinate Frame k	Multiplications	Additions
0	$39N - 67$	$27N - 49$
1	$39N - 102$	$27N - 75$
2	$39N - 106$	$27N - 79$
3	$39N - 106$	$27N - 79$
4	$39N - 106$	$27N - 79$
.	.	.
.	.	.
.	.	.
$N-4$	$39N - 103$	$27N - 76$
$N-3$	$39N - 103$	$27N - 76$
$N-2$	$39N - 103$	$27N - 76$
$N-1$	$39N - 78$	$27N - 57$
N	$39N - 39$	$27N - 30$

Table 4 Modular Processors for VLSI Implementations

Name of Processing Elements	Number of Modular Processors	Usage	Flops Required
MMP	9	evaluate a product of 2 3×3 matrices	3
MVP	3	evaluate a product of a 3×3 matrix and a 3×1 vector	3
VCP	3	evaluate a vector cross product	2
VVA	1	evaluate a vector addition	1

Figure 1 The Coordinate System and Its Parameters

Figure 2 Comparison of $O(N)$ and $O(\log N)$

Parallel computation of $Q^+(i,k) = Q^+(i-1,k) * x(k+i)$,
 $Q^-(i,k) = Q^-(i-1,k) * x(k-i)$;

Q^+ represents forward recursive linear equations,
 Q^- represents backward recursive linear equations,

Dark circles represent positive operations,
 Open circles represent null operations.

Figure 3 Forward and Backward Recursive Doubling Algorithm

Figure 4 Data Flow and Systolic Array Implementation on Linear Computation of the Jacobian

Figure 5 Block Diagram for VLSI on Parallel Computation of the Jacobian

Figure 6 Block 1 for any N with Reference Coordinate Frame k

Figure 7 Block 2 for any N with Reference Coordinate Frame k

Figure 8 Block 3 for any N with Reference Coordinate Frame k

Figure 9 Block 4 for any N with Reference Coordinate Frame k