N90-23010

Parallel Algorithms and Architecture for Computation of Manipulator Forward Dynamics

Amir Fijany and Antal K. Bejczy

Jet Propulsion Laboratory, California Institute of Technology

Abstract- In this paper parallel computation of manipulator forward dynamics is investigated. Considering three classes of algorithms for the solution of the problem, that is, the $O(n)$, the $O(n^2)$, and the $O(n^3)$ algorithms, parallelism in the problem is analyzed. It is shown that the problem belongs to the class of *NC* and that the time and processors bounds are of $O(\log_2^2 n)$ and $O(n^4)$, respectively. However, the fastest stable parallel algorithms achieve the computation time of $O(n)$ and can be derived by parallelization of the $O(n^3)$ serial algorithms. Parallel computation of the $O(n^3)$ algorithms requires the development of parallel algorithms for a set of fundamentally different problems, that is, the Newton-Euler formulation, the computation of the inertia matrix, decomposition of the symmetric, positive definite matrix, and the solution of triangular systems. Parallel algorithms for this set of problems are developed which can be efficiently implemented on a unique architecture, a triangular array of $n(n+1)/2$ processors with a simple nearest-neighbor interconnection. This architecture is particularly suitable for VLSI and WSI implementations. The developed parallel algorithm, compared to the best serial $O(n)$ algorithm, achieves an asymptotic speedup of more than two orders-of-magnitude in the computation the forward dynamics.

I. INTRODUCTION

The manipulator forward dynamics problem concerns the determination of the motion of the mechanical system resulting from the application of a set of joint forces/torques which is essential for dynamic simulation. The motivation for devising fast algorithms for forward dynamics computation stems from applications which require extensive off-line simulation as well as applications which require real-time dynamic simulation capability. In particular, for many anticipated space teleoperation applications, a faster-than-real-time simulation capability will be essential. In fact, in the presence of unavoidable delay in information transfer, such a capability would allow a human operator to preview a number of scenarios before run-time [1].

The forward dynamics problem can be stated as follows: Given the vectors of actual joint positions (Q) and velocities ($\dot{Q}$), the external force (f_E) and moment (n_E) exerted on the End-Effector (EE), and the vector of applied joint

forces/torques (τ), find the vector of joint accelerations ($\ddot{Q}$). Integrating the vector of joint accelerations leads to the new values for Q and $\dot{Q}$, and the process is then repeated for the next τ. The first step in computing the forward dynamics is to derive a linear relation (for the given manipulator configuration described by the vector of joint positions) between the vector of joint accelerations and the vector of applied inertial forces/torques.

Given the dynamic equations of motion as

$$A(Q)\ddot{Q} + C(Q,\dot{Q}) + G(Q) + J^t(Q)F_E = \tau \quad (1)$$

and defining the bias vector b as

$$b = C(Q,\dot{Q}) + G(Q) + J^t(Q)F_E \quad (2)$$

the linear relation is derived:

$$A(Q)\ddot{Q} = \tau - b = \Gamma \quad (3)$$

where Q, $\dot{Q}$, and $\ddot{Q}$ are nx1 vectors and F_E, a 6x1 vector, is a combined representation of f_E and n_E. A(Q) is an nxn symmetric, positive definite, inertia matrix, and J is the 6xn Jacobian matrix (t denotes matrix transpose). The bias vector b represents the contribution due to coriolis and centrifugal terms $C(Q,\dot{Q})$, gravitional terms G(Q), and the external force and moment. Hence, in Eq. (3), Γ is the nx1 vector of applied inertia forces/torques. The bias vector b can be obtained by solving the inverse dynamics problem, using the Newton-Euler (N-E) formulation [2], while setting the vector of joint accelerations to zero. The computation of the vectors b and Γ represent the common first step in any algorithm for solving the forward dynamics problem.

The proposed algorithms for the forward dynamics problem differ in their approaches to solving Eq. (3), which directly affects their asymptotic computation complexity. These algorithms can be classified as:

1) The $O(n)$ algorithms [3]-[6] which, by taking a more explicit advantage of the structure of problem, e.g., by using the Articulated-Body Inertia [3]-[4] and recursive factorization and inversion of the inertia matrix [5]-[6], solve Eq. (3) in $O(n)$ steps without explicit computation and inversion of the inertia matrix.

2) The $O(n^2)$ conjugate gradient algorithms [7, 10] which iteratively solve Eq. (3) without explicit computation and inversion of the inertia matrix. The conjugate gradient algorithm is guaranteed to converge to the solution in at most n iterations which, given the $O(n)$ computational complexity of each iteration, leads to an overall $O(n^2)$ computational complexity.

3) The $O(n^3)$ algorithms [7] which solve Eq. (3) by explicit computation and inversion of the inertia matrix, leading to an $O(n^3)$ computational complexity.

However, any analysis of the relative efficiency of these algorithms should be based on the realistic size of the problem, i.e., the number of Degree-Of-Freedom (DOF), rather than the asymptotic complexity. In fact, the comparative study in [3]-[4] shows that the $O(n^3)$ Composite Rigid-Body algorithm is the most efficient for n less than 12. It should be pointed out

that efficiency of the $O(n^3)$ and $O(n^2)$ algorithms has been recently improved [9]-[10]. However, despite these improvements, even the fastest $O(n^3)$ algorithm is far from providing the efficiency required for real-time or faster-than-real-time simulation. This observation clearly suggests that the exploitation of a high degree of parallelism in the computation is the key factor in achieving the required efficiency.

The analysis of the efficiency of the different algorithms for parallel computation is more complex than that for serial computation. In the next section, the three classes of algorithms are analyzed based on their efficiency for parallel computation and it is shown that the $O(n^3)$ algorithms are also the most efficient for parallel computation. However, parallelization of the $O(n^3)$ algorithms represents a challenging problem since it requires the development of parallel algorithms for computation of a set of fundamentally different problems, i.e., the N-E formulation, the inertia matrix, the factorization of the inertia matrix, and the solution of triangular systems.

Lee and Chang [15] were first to investigate the computation of the forward dynamics by parallelization of the $O(n^3)$ algorithms. Considering an SIMD architecture with n processors interconnected through a generalized-cube network, a modified version of their $O(\log_2 n)$ algorithm in [16] and an $O(n)$ parallel version of the Composite Rigid-Body algorithm were developed for computation of the N-E formulation and the inertia matrix. A parallel $O(n^2)$ Cholesky algorithm and the $O(n)$ Column-Sweep algorithms were also proposed for the factorization of the inertia matrix and the solution of the resulting triangular systems, leading to an $O(n^2)$ complexity of the overall computation. However, the main drawbacks of the proposed algorithms reside in the complexity of the required interconnection network and the $O(n^2)$ communication complexity which mainly results from the excessive data alignment needed for different algorithms.

In this paper, we present a set of efficient parallel algorithms for the computation of the forward dynamics, using the $O(n^3)$ algorithms, which can be implemented on a two-dimensional array of $n(n+1)/2$ processors with a nearest neighbor interconnection. The overall of communication complexity, even with such simple interconnection structure, is limited to $O(n)$ and no additional data alignment between the computation of the different algorithms is required, which further reduces the overhead in the parallel computation. A new algorithm for computation of the inertia matrix is developed which, though not efficient for serial processing, achieves the best performance for parallel computing in terms of both computation and communication complexity while demanding simple architectural features for its implementation. The parallel algorithm for computing the inertia matrix achieves the time lower bound of $O(\log_2 n)+O(1)$ on the processor array. Synchronous data-flow parallel algorithms are also developed for factorization of the inertia matrix and the solution of the resulting triangular systems on the processor array.

This paper is organized as follows. In Section II, parallelism and time and processors bounds in the computation of the forward dynamics are investigated. In Section III, parallel algorithm for computation of the inertia matrix is developed. In Section IV, parallel computation of the bias vector and the linear system solution are briefly discussed. Finally, some concluding remarks are made in Section V.

II. PARALLELISM IN FORWARD DYNAMICS COMPUTATION

A. Time and Processor Bounds in the Computation

The analysis of time and processors bounds in parallel computation of a given problem is of fundamental theoretical importance. It can determine the inherent parallelism in the problem and the bound on the number of processors required for exploiting maximum parallelism and achieving the time lower bound in the computation. However, besides the theoretical importance, it can also provide, as is the case for forward dynamics problems, useful insights into devising more practical and efficient parallel algorithms (in the sense of both computation time and number of processors) for the problem.

Let P denote the class of problems that can be solved sequentially in a time bounded by a polynomial of the input size, n. Also, let NC (for "Nick's Class" [18]) stand for the class of problems that can be solved in parallel in a time of $O(\log_2^k n)$, for some constant k, with a number of processors bounded by a polynomial of n. One open question regarding the complexity of parallel algorithms is whether $P = NC$, which is thought to be very unlikely [19]. It is clear that $NC \subseteq P$. For k = 1, the time of $O(\log_2 n)+O(1)$ represents the natural time lower bound in the computation. However, most of the kinematic and dynamic problems in robotics belong to the class of NC [8]. Furthermore, it is possible to devise parallel algorithms which achieve the time lower bound of $O(\log_2 n)+O(1)$ in solving these problems [8,14,16,17]. In the following, we study the time and processors bounds in the computation of the forward dynamics by different algorithms.

Using the N-E formulation, the bias vector can be computed in a time of $O(\log_2 n)+O(1)$ with $O(n)$ processors [15]-[16]. This implies that the time and processors bounds in the forward dynamics computation are determined by those in the solution of Eq. (3). Note that, with $O(n)$ processors, the integration of the computed joint accelerations can be performed in a time of $O(1)$.

The solution of Eq. (3) by the $O(n)$ algorithms results in a set of first-order nonlinear recurrences which can be represented (at an abstarct level) as

$$X_i = C_i + \phi_2(X_{i+1})/\phi_1(X_{i+1}) = C_i + \phi(X_{i+1}) \qquad (4)$$

where C_i is constant, ϕ_1 and ϕ_2 are polynomials of first and second degree, and deg ϕ = max (deg ϕ_1, deg ϕ_2) = 2. It is well-known that, regardless of the number of processors, the computation of nonlinear recurrences of the form of Eq. (4) and with deg ϕ>1 can be speeded up only by a constant factor [20]-[21]. This is due to the fact that the data dependency in nonlinear

recurrences and, particularly, those containing division, is stronger than in linear recurrences [22]. Hence, the parallelism in the $O(n)$ algorithms is bounded, that is, their parallelization leads to the $O(n)$ algorithms which are faster than the serial algorithm only by a constant factor. Note that a rather simple model was used for presentation of the nonlinear recurrences of the $O(n)$ algorithms while they are far more complex than those usually studied in literature, e.g., in [21]-[22] (see [8] for a more detailed discussion).

For the conjugate gradient algorithms in [7], [10], the computation of each iteration, as is shown in [15], can be done in a time of $O(\log_2 n)$ with n processors, leading to the $O(n\log_2 n)$ parallel algorithms. This implies that the parallelism in conjugate gradient algorithm is unbounded. Asymptotically, however, the parallel conjugate gradient algorithms are slower than the best serial algorithms, the $O(n)$ algorithms, for the solution of the problem.

The inertia matrix can be computed in $O(\log n)+O(1)$ steps with $O(n^2)$ processors [8], [11], [13]. The implication of this result is that it further reduces the analysis of the time and processors bounds in the forward dynamics problem to that in a more generic problem, the linear system solution. Csanky has shown that the linear system can be solved in $O(\log_2^2 n)$ steps with $O(n^4)$ processors [23] . This implies that the forward dynamics problem belongs to the class of *NC*. Note that, using Cramer's rule, the linear system solution can be computed in $O(\log n)$ steps with $O(n!)$ processors [20]. But such a result has neither theoretical nor practical importance.

However, Csanky's algorithm is unpractical since, besides using too many processors, it is numerically unstable [25]. The best stable algorithms for linear system solution achieve a time of $O(n)$ with $O(n^2)$ processors [24]-[25]. Hence, parallelization of the $O(n^3)$ algorithms results in the stable $O(n)$ parallel algorithms with $O(n^2)$ processors, which indicates an unbounded parallelism.

The above analysis shows that the forward dynamics problem belongs to the class of *NC* and that the best known upper bounds on the time and processors are $O(\log_2^2 n)$ and $O(n^4)$, respectively. Practically, however, the fastest (and stable) parallel algorithm for its computation is of $O(n)$. With respect to these results the main question is, given the fact that both the serial $O(n)$ and $O(n^3)$ algorithms result in the $O(n)$ parallel algorithms, which one is more efficient for parallelization?

Let $\alpha_1 n+\beta_1$ denote the polynomial complexity of the serial $O(n)$ algorithms. There is a limited parallelism in both coarse grain and fine grain (in matrix-vector operation) forms in these algorithms [8]. Exploitation of this parallelism leads to the parallel algorithms with polynomial complexity as $\alpha_2 n+\beta_2$ where, due to the limited parallelism, α_1 is reduced to α_2 only by a small factor. Furthermore, exploitation of both coarse and fine grain parallelism requires additional architectural complexity. For the $O(n^3)$

algorithms, the polynomial complexity of the resulting parallel $O(n)$ algorithm is of the form $\alpha_3 n+\gamma_3\lceil\log_2 n\rceil+\beta_3$ where α_3 is smaller than α_1 by more than two orders-of-magnitude. As a result, while the algorithm is asymptotically faster than the serial $O(n)$ algorithms and their parallel versions by a high constant factor, it is also more efficient for small n. The price to be paid for this efficiency, of course, is an architecture with $O(n^2)$ processors. However, the efficiency of the parallel algorithm and the suitability of the architecture for VLSI and WSI implementation strongly support the choice of $O(n^3)$ algorithms for parallel computing.

III. PARALLEL COMPUTATION OF INERTIA MATRIX

A. Basic Algorithms for computation of inertia matrix

From Eq. (3) the elements of the inertia matrix can be computed as

$$a_{ij} = a_{ji} = \Gamma_j \tag{5}$$

for the condition given by

$$\ddot{Q}_i = 1 \text{ and } \ddot{Q}_{k\neq i} = 0 \qquad \text{For } k = 1, 2, \ldots, n \tag{6}$$

Two physical interpretations can be thought for the above condition, with each interpretation leading to a distinct class of algorithms as

1) The first i-1 links do not have any motion, that is, they are static, and the accelerations and the forces/torques of the last n-i+1 links result from the unit acceleration of link i. This interpretation leads to the first class of algorithms, designated as the class of Newton-Euler Based (NEB) algorithms, in which the diagonal and lower off-diagonal elements of the inertia matrix are computed. In [7] an algorithm of this class is presented, designated as the Original NEB (ONEB) algorithm, which computes the inertia matrix by successive applications of the N-E formulation.

2) The last n-i+1 links can be considered as a single composite rigid system, since they do not have any relative motion, which is accelerating in space, leading to the exertion of forces and moments on the first i-1 static links. This interpretation leads to the second class of algorithms, designated as the class of Composite-Rigid Body (CRB) algorithms, in which the diagonal and upper off-diagonal elements of the inertia matrix are computed. In [7] an algorithm of this class, designated as the Original CRB (OCRB) algorithm, is presented in which the center of mass and the first and the second moment of mass with respect to the center of mass of a set of composite systems are computed.

The comparative study in [7] shows that the OCRB algorithm achieves a significantly greater efficiency over the ONEB algorithm. In [8]-[9], we have developed an algorithm, designated as the Variant of CRB (VCRB) algorithm, which avoids the redundancies in the OCRB algorithm and represents the most efficient algorithm (to date) for computing the inertia matrix. Note that, however, due to the symmetry of the problem, both interpretations and hence both classes of algorithms should lead to the same results and computational

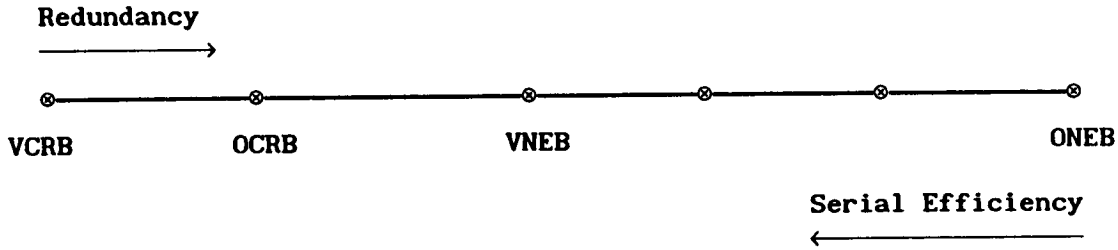

Fig. 1. Comparison of different algorithms for computation of inertia matrix

efficiency. In [8]-[9], we have shown that, by introducing or reducing the redundancy in the computation, the algorithms of the two classes can be transformed to one another and, particularly, to the most efficient one, the VCRB algorithm. Figure 1 shows the relative serial efficiency of and redundancy in different algorithms.

Although the results presented in Fig. 1 answer the question of the serial efficiency of different algorithms, it does not indicate which algorithm provides the most suitable features for parallelization. For serial processing, removing any redundancy increases the computational efficiency. For parallel processing, however, depending on its impact on the data dependency in the computation, this may increase or decrease the efficiency. The fact that arbitrary algorithms can be developed by introducing or removing different types of redundancy in the computation represents an additional degree-of-freedom that can be exploited to derive an algorithm which, though perhaps not efficient for serial computing, is the most suitable for parallelization. In [8], [12], we have shown that the NEB algorithms are more suitable for parallel computing than the CRB algorithms. In fact, they not only achieve a better computational complexity (in the parallel sense) but also require a less complex communication and synchronization mechanism. This better efficiency for parallelization mainly results from the fact that the evaluation of the columns of the inertia matrix by the NEB algorithms is order independent and hence can be done in parallel.

B. A Variant of Newton-Euler Based (VNEB) Algorithm

Four different types of redundancy can be recognized in the ONEB algorithm, which can be eliminated, respectively, by [8], [9], [13]:

1) Choosing a more suitable coordinate frame for projection of the equations.
2) Optimizing the N-E formulation for the condition given by Eq. (6).
3) Using a more efficient variant of the optimized N-E formulation.
4) Introducing a two-dimensional recursion in the computation.

Note that the first redundancy resides in the *extrinsic* equations and results from the choice of coordinate frame for projection of the *intrinsic* equations while the second, the third, and the fourth redundancies reside in the intrinsic equations and are inherent in the formulation. As stated before, by removing all redundancies in the intrinsic equations, the ONEB algorithm can be transformed to the VCRB algorithm. However, removing any type of redundancy in the NEB algorithms, as far as the order independence property

is preserved, will also increase the efficiency of their parallel versions. In this regard, only the removal of the fourth redundancy, which leads to the introduction of a two-dimensional recursion in the computation, results in the loss of the order independence property of the algorithm. In the following, a Variant of the NEB algorithm, designated as the VNEB algorithm, is presented which is developed by removing the first three redundancies.

The derivation of the Variant of N-E Based (VNEB) algorithm is fully discussed in [8], [9], [12], [13]. Here, for the sake of completeness, a brief description of the algorithm is given. The algorithm is presented by the intrinsic equations. In this paper, according to the Gibbs notation, vectors are underlined once and tensors (tensors of order 2) twice. Also, in order to simplify and, particularly, unify the derivation of the serial and parallel algorithms, a set of notations, given in Table I and Fig. 3, are used. The VNEB algorithm is then written as

For $i = 1, 2, \ldots, n$

  For $j = i, i+1, \ldots, n$

$$\dot{\underline{\omega}}(j,i) = \underline{Z}(i) \tag{7}$$

$$\dot{\underline{V}}(j,i) = \dot{\underline{V}}(j-1,i) + \dot{\underline{\omega}}(j,i)\times\underline{P}(j,j-1) = \dot{\underline{\omega}}(j,i)\times\underline{P}(j,i) \tag{8}$$

$$\underline{F}(j+1,j,i) = M(j)\dot{\underline{V}}(j,i) + \dot{\underline{\omega}}(j,i)\times\underline{H}(j) \tag{9}$$

$$\underline{N}(j+1,j,i) = \underline{K}(j)\dot{\underline{\omega}}(j,i) \tag{10}$$

  For $j = n, n-1, \ldots, i$

$$\underline{F}(n+1,j,i) = \underline{F}(j+1,j,i) + \underline{F}(n+1,j+1,i) \tag{11}$$

$$\underline{N}(n+1,j,i) = \underline{N}(j+1,j,i) + \underline{N}(n+1,j+1,i) + \underline{P}(j+1,j)\times\underline{F}(n+1,j+1,i) \tag{12}$$

$$\text{with } \underline{F}(n+1,n+1,i) = \underline{N}(n+1,n+1,i) = 0$$

$$a_{ji} = \underline{Z}(j).\underline{N}(n+1,j+1,i) \tag{13}$$

C. Parallel Algorithm for Computation of Inertia Matrix

The serial computational complexity of evaluating the inertia matrix is of $O(n^2)$. No serial algorithm can achieve a better asymptotic complexity since, given n inputs (joint positions), the evaluation of the $O(n^2)$ outputs, the elements of the inertia matrix, requires $O(n^2)$ distinct steps in the computation. Based on the VCRB algorithm, we have already shown that the inertia matrix can be computed in $O(\log_2 n)+O(1)$ steps with $O(n^2)$ processors [8], [11]. It is interesting to note that not only the same bounds on time and processors can be much more easily derived by parallelization of the VNEB algorithm but also the resulting parallel algorithm, compared to the parallel VCRB algorithm, reduces the coefficients on the polynomial complexity.

For the implementation of the parallel algorithm achieving the time lower bound, we consider a two-dimensional array of $n(n+1)/2$ processor-memory modules represented as PR_{ij}, for $i = 1, 2, \ldots, n$ and $j = i, i+1, \ldots, n$

(Fig. 3 shows the array for n = 6). For the parallel algorithm, the equations are projected onto the EE coordinate frame, coordinate frame n+1. An n DOF all revolute joints manipulator is considered for which the joint variables are the joint angles, that is, $Q_j = \theta_j$. It is assumed that the joint variables θ_j (or $S\theta_j$ and $C\theta_j$) and constant parameters $S\alpha_j$, $C\alpha_j$, ${}^{j+1}P(j+1,j)$, ${}^{j+1}H(j)$, ${}^{j+1}K(j)$, and $M(j)$ reside in the memory of all processors of Row j. The analysis of the mapping the algorithm onto the architecture of Fig. 3 is fully presented in [12]. Here, due to the lack of space, we only give the a brief description of the algorithm in terms of its computational steps and cost.

For the parallel algorithm, the ith column of the inertia matrix is computed by the processors of the ith column of the processor array. The fact that $\dot{\omega}(j,i) = Z(i)$ for $j = i, i+1, \ldots, n$, implies that global communication of $Z(i)$ among the processors of the ith column is required. This requirement can be avoided by introducing two recurrences as

$$\dot{\omega}(j,i) = \dot{\omega}(j-1,i) = Z(i) \tag{14}$$

$$P(j,i) = P(j-1,i) + P(j,j-1) \tag{15}$$

Equation (14) does not need any computation while, for the parallel algorithm, the computation of Eq. (15) is required. By computing Eqs. (14)-(15) as a set of coupled recurrences, the terms $\omega(j-1,i)$ can be considered as the data associated with Eq. (15). Using such a scheme increases the communication complexity of parallel evaluation of Eq. (15) but will result in the global distribution of $Z(i)$ among the processors of the ith column. The computation of the parallel algorithm is then performed as follows.

Step 1:
1) Parallel compute $R(j+1,j)$ by all processors of the jth row.

For $j = 1, 2, \ldots, n$

  For $i = 1, 2, \ldots, j$

    PR_{ji} : $R(j+1,j)$ (16)

2) Parallel compute $R(n+1,j)$ by processors of the ith column.

For $i = 1, 2, \ldots, n$
  For $j = i, i+1, \ldots, n$
    For $\eta = 1$ step 1 until $\lceil \log_2(n+1-j) \rceil$, Do (17)

$$R(j+2^{\eta},j) = R(n+1,j) \qquad j+2^{\eta} > j+2^{\eta-1} \geq n+1$$

$$R(j+2^{\eta},j) = R(n+1,j) = R(n+1,j+2^{\eta-1})R(j+2^{\eta-1},j) \qquad j+2^{\eta} \geq n+1 > j+2^{\eta-1}$$

$$R(j+2^{\eta},j) = R(j+2^{\eta},j+2^{\eta-1})R(j+2^{\eta-1},j) \qquad n+1 > j+2^{\eta} > j+2^{\eta-1}$$

    End_Do

3) Rotate $R(n+1,j)$ by processors of Row j to the processors of Row j-1.
For $j = 1, 2, \ldots, n$
  For $i = 1, 2, \ldots, j$
    PR_{ji}: $R(n+1,j+1)$ (18)

with $R(n+1,n+1) = U$ (Unit Matrix)

Note that, as the result of the above data transfer, both the terms $R(n+1,j)$ and $R(n+1,j+1)$ reside in the memory of all the processors of the jth row.

4) Parallel compute ${}^{n+1}Z(j)$, ${}^{n+1}P(j+1,j)$, ${}^{n+1}H(j)$, and ${}^{n+1}K(j)$ by all the processors of the jth row.

For $j = 1, 2, \ldots, n$

For $i = 1, 2, \ldots, j$

a) PR_{ji} : ${}^{n+1}Z(j) = R(n+1,j)\,{}^{j}Z(j)$ (19)

with ${}^{j}Z(j) = [0\ 0\ 1]^t$

b) PR_{ji}: ${}^{n+1}P(j+1,j) = R(n+1,j+1)\,{}^{j+1}P(j+1,j)$ (20)

with ${}^{j+1}P(j+1,j) = [a_i\ \ d_i S\alpha_i\ \ d_i C\alpha_i]^t$

c) PR_{ji}: ${}^{n+1}H(j) = R(n+1,j+1)\,{}^{j+1}H(j)$ (21)

d) PR_{ji}: ${}^{n+1}K(j) = R(n+1,j+1)\,{}^{j+1}K(j)R(j+1,n+1)$ (22)

Note that for the processors of the nth row Eqs. (21)-(22) do not need any computation since the terms ${}^{n+1}H(n)$ and ${}^{n+1}K(n)$ are given constant parameters. As the result of the computation of Step 1, all the vectors and the tensors are projected onto the coordinate frame n+1. In the following, the absence of superscripts denotes that the computations are performed in this frame.

Step 2:

1) Parallel compute $P(j,i)$ and $\dot{\omega}(j,i)$ by processors of the ith column.

For $i = 1, 2, \ldots, n$

For $j = i, i+1, \ldots, n$

For $\eta = 1$ step 1 until $\lceil \log_2(n+1-j) \rceil$, Do

$$\dot{\omega}(j+2^{\eta},i) = \dot{\omega}(j-2^{\eta},i) = Z(i) \tag{23}$$

$$P(j+2^{\eta},j) = P(i,j) \qquad j+2^{\eta} > j+2^{\eta-1} \geq i$$

$$P(j+2^{\eta},j) = P(i,j) = P(i,j+2^{\eta-1}) + P(j+2^{\eta-1},j) \qquad j+2^{\eta} \geq i > j+2^{\eta-1} \tag{24}$$

$$P(j+2^{\eta},j) = P(j+2^{\eta},j+2^{\eta-1}) + P(j+2^{\eta-1},j) \qquad i > j+2^{\eta} > j+2^{\eta-1}$$

End_Do

2) Parallel compute $\dot{V}(j,i)$, $F(j+1,j,i)$, $N(j+1,j,i)$ by processors of the ith column.

For $i = 1, 2, \ldots, n$

For $j = i, i+1, \ldots n$

PR_{ji}: $\dot{V}(j,i) = \dot{\omega}(j,i) \times P(j,i)$ (25)

$$PR_{ji}:\ F(j+1,j,i) = \dot{\omega}(j,i)xH(j) + M(j)\dot{V}(j,i) \tag{26}$$

$$PR_{ji}:\ N(j+1,j,i) = K(j)\dot{\omega}(j,i) + H(j)x\dot{V}(j,i) \tag{27}$$

Step 3:

1) Parallel compute F(n+1, j, i) and N(n+1, j, i) by processors of the ith column.

For i = 1, 2, ..., n

For j = i, i+1, ..., n

For η = 1 step 1 until $\lceil \log_2(n+1-j) \rceil$, Do (28)

$F(j+2^{\eta}, j, i) = F(n+1, j, i)$ $\quad j+2^{\eta} > j+2^{\eta-1} \geq n+1$

$F(j+2^{\eta}, j, i) = F(n+1, j, i) = F(j+2^{\eta}, j+2^{\eta-1}, i)+F(j+2^{\eta-1}, j, i)$ $\quad j+2^{\eta} > n+1 > j+2^{\eta-1}$

$F(j+2^{\eta}, j, i) = F(j+2^{\eta}, j+2^{\eta-1}, i)+F(j+2^{\eta-1}, j, i)$ $\quad n+1 > j+2^{\eta} > j+2^{\eta-1}$

End_Do

For η = 1 step 1 until $\lceil \log_2(n+1-j) \rceil$, Do (29)

$N(j+2^{\eta}, j, i) = N(n+1, j, i)$ $\quad j+2^{\eta} > j+2^{\eta-1} \geq n+1$

$N(j+2^{\eta}, j, i) = N(n+1, j, i) = N(n+1, j+2^{\eta-1}, i)+N(j+2^{\eta-1}, j, i)+$

$P(j+2^{\eta-1}, j)xF(n+1, j+2^{\eta-1}, i)$ $\quad j+2^{\eta} \geq n+1 > j+2^{\eta-1}$

$N(j+2^{\eta}, j, i) = N(j+2^{\eta}, j+2^{\eta-1}, i)+N(j+2^{\eta-1}, j, i)+$

$P(j+2^{\eta-1}, j)xF(j+2^{\eta}, j+2^{\eta-1}, i)$ $\quad n+1 > j+2^{\eta} > j+2^{\eta-1}$

End_Do

2) Parallel compute a_{ji} by PR_{ji}.

For i = 1, 2, ..., n

For j = i, i+1, ...n

$$PR_{ji}:\ a_{ji} = Z(j).N(n+1, j, i) \tag{30}$$

As stated before, the time lower bound in, as well as the computational cost of, parallel evaluation of the inertia matrix, using the VNEB algorithm, is determined by that of the parallel evaluation of the first column of the inertia matrix. In other words, the computational cost of the n recurrences in Eqs. (17), (23), (24), (28), and (29) is determined by that of the largest ones, i.e., for i = 1, which are of size n. Furthermore, the computational cost of all the $O(n^2)$ terms in Eqs. (16), (19)-(22), (25)-(27), and (30) is determined by the cost of one term since for each column n terms are computed in parallel and the computation for n columns, as will be discussed later, is overlapped. Let *m* and *a* denote the cost of multiplication and addition, respectively. The computational cost of the parallel algorithm is then evaluated as follows:

Step 1: The cost of Eq. (16) is $4m$; The cost of Eq. (17) is $(27m+18a)\lceil \log_2 n \rceil$; Eq. (18) represents a simple data rotation; Eq. (19) does not need any computation; The cost of Eqs. (20), (21), and (22) is $(9m+6a)$, $(9m+6a)$, and $(54m+36a)$, respectively. The cost of this step is $(27m+18a)\lceil \log_2 n \rceil+(76m+48a)$.

Step 2: Eq. (23) does not need any computation; The cost of Eq. (24) is $(3a)\lceil \log_2 n \rceil$; The cost of Eqs. (25), (26), and (26) is $(6m+3a)$, $(9m+6a)$, and $(15m+12a)$, respectively. The cost of this step is $(3a)\lceil \log_2 n \rceil+(30m+21a)$.

Step 3: The cost of Eqs. (28) and (29) is $(3a)\lceil \log_2 n \rceil$ and $(6m+9a)\lceil \log_2 n \rceil$, respectively; The cost of Eq. (30) is $(3m+2a)$. The cost of this step is $(6m+12a)\lceil \log_2 n \rceil+(3m+2a)$.

Adding the cost of Steps 1-3, the computation cost of the algorithm is obtained as $(33m+33a)\lceil \log_2 n \rceil+(109m+71a)$. As stated before, mapping the developed parallel algorithm onto the processor array is fully presented in [12] where it is shown that, even using a simple nearest neighbor interconnection structure, a communication complexity of $O(n)$ can be achieved. Also, the mechanisms for global and local synchronization for the processor array are presented. Figure 4 shows the resulting distribution of the elements of the inertia matrix among the processors.

IV. PARALLEL COMPUTATION OF N-E FORMULATION AND SOLUTION OF LINEAR SYSTEM

As stated before, the bias vector can be computed by evaluating the N-E formulation while setting the vector of joint accelerations to zero. We use the parallel algorithm presented in [15] for computing the bias vector. This computation is performed by the processors of the first column with the equations being projected onto the frame n+1. Therefore, the results of the computation of the first step except Eqs. (21)-(22) can be used while, similar to the terms ${}^{n+1}K(j)$ in Eq. (22), the terms ${}^{n+1}J(j)$, for $j = 1, 2, \ldots, n$, are also needed to be computed by the processors of the first column. The cost of evaluation of the vector Γ is then obtained as $15a\lceil \log_2 n \rceil+(141m+101a)$. With the nearest neighbor interconnection among the processors of the first column a communication complexity of $O(n)$ is achieved. In order to achieve the proper sequencing of the computation of the inertia matrix, the bias vector, and the linear system solution, a data-driven mechanism can be employed. That is, the processors of all columns, except the processors of the first column, by completion of the computation of their corresponding column of the inertia matrix enter the wait state while the processors of the first column start the computation of the bias vector and the vector Γ. The activity of the processors of column 2 through column n in linear system solution is triggered by receiving the corresponding data and by the completion of the computation of the vector Γ.

Figure 4 shows the organization of the data resulting from the computation of the inertia matrix and the vector Γ. Given this data organization, we have developed synchronous data-flow parallel algorithms for the full solution of Eq. (3), that is, for decomposition of the inertia matrix, using the square-root-free variant of the Cholesky factorization, and the solution of the

resulting lower and upper triangular linear systems, which are presented in detail in [12]. The computation cost of the solution of Eq. (3) is obtained as $(n-1)(3m+2a)+(1m+1a)$, where the cost of division and multiplication is taken to be the same. A communication complexity of $O(n)$ is also achieved.

Adding the cost of evaluation of the inertia matrix, the bias vector and the vector Γ, and the linear system solution, the computational cost of the forward dynamics problem is obtained as $(3m+2a)n+(33m+48a)\lceil \log_2 n \rceil+(238m+171a)$. The computational cost of the best serial $O(n)$ algorithm, the Articulated-Body algorithm [3]-[4], is evaluated as $(380m+302a)n-(198m+173a)$ [15]. If the time of multiplication and addition is taken to be the same, then, for $n = 6$, the developed parallel algorithms achieve a speedup of 6 compared to the best serial $O(n)$ algorithm. Note, however, that as n increases, e.g., for redundant manipulators, the speedup also significantly increases. To see this, let us write the computational cost of the serial $O(n)$ algorithm and the parallel algorithms as $682n+O(1)$ and $5n+O(\log_2 n)+O(1)$, with the time of multiplication and addition taken to be the same. It can be seen that, asymptotically, the parallel algorithms achieve a speedup of more than two orders-of-magnitude. For small n, the computational cost of the parallel algorithms is dominated by the $\lceil \log_2 n \rceil$-dependent and constant terms on the polynomial complexity. Hence, for small n, the computational complexity of the developed parallel algorithms can be practically considered as $O(\log_2 n)+O(1)$.

V. CONCLUSION

We developed a set of parallel algorithms for computing the forward dynamics problem. These algorithms exploit a high degree of parallelism in the problem and achieve a significant speedup in the computation. Furthermore, they can be efficiently implemented on a two-dimensional array of processors with a nearest neighbor interconnection. This architecture is particularly suitable for practical implementation using VLSI and WSI technologies. Due to their simple architectural requirements, these algorithms, with some modifications, can also be efficiently implemented on rather more general-purpose architectures, e.g., a two-dimensional array of Transputers.

A key factor in our approach to the parallel computation of the forward dynamics is the minimization of the resulting overhead. The overall communication complexity is of $O(n)$. The overhead is further minimized since there is no need for any data alignment between the computation of different algorithms and the intermediate data resulting from the different algorithms are generated and consumed within the array. Also, the final result of the computation, that is, the vector of joint accelerations, is computed by the processors of the first column. Therefore, they can be output using the same channels for inputting the data to the array (Fig. 5). This is particularly critical for VLSI and WSI implementation since, by using only n bidirectional Input/Output channels, the number of required pins is kept small.

ACKNOWLEDGEMENT

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration (NASA).

References

[1] M.H. Milman and G. Rodriguez,"Cooperative Dual Arm Manipulator Issues and Task Approach," *Jet Propulsion Lab., Eng. Memorandum 347*, Nov. 1987.

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

[3] R. Featherstone,"The Calculation of Robot Dynamics Using Articulated-Body Inertia," *Int. J. Robotics Research*, Vol.2(2), 1983.

[4] R. Featherstone, *Robot Dynamics Algorithms*. Ph.D. Dissertation, Univ. of Edinburgh, 1984.

[5] G. Rodriguez,"Kalman Filtering, Smooting and Recursive Robot Arm Forward and Inverse Dynamics," *IEEE J. Robotics and Automation*, Vol. RA-5, Dec. 1987. Also in Jet Propulsion Lab. Publication 86-48, Dec. 1986.

[6] G. Rodriguez and K. Kreutz,"Recursive Mass Matrix Factorization and Inversion: An Operator Approach to Open- and Closed-Chain Multibody Dynamics," *Jet Propulsion Lab. Publication* 88-11, May 1988.

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

[8] A. Fijany, *Parallel Algorithms and Architectures in Robotics*. Ph.D. Dissertation, Univ. of Paris XI (Orsay), Sept. 1988.

[9] A. Fijany and A.K. Bejczy,"An Efficient Algorithm for Computation of the Manipulator Inertia Matrix," To appear in J. of Robotic Systems, Feb. 1990.

[10] A. Fijany and R.E. Scheid,"Efficient Conjugate Gradient Algorithms for Computation of the Manipulator Forward Dynamics," *Proc. of NASA Conf. on Space Telerobotics*, Pasadena, CA, Jan. 1989.

[11] A. Fijany and A.K. Bejczy,"A Class of Parallel Algorithms for Computation of the Manipulator Inertia Matrix," *IEEE Trans. Robotics and Automation*, Vol. RA-5, No. 5, pp. 600-615, Oct. 1989.

[12] A. Fijany and A.K. Bejczy," Parallel Algorithms and Architecture for Computation of the Manipulator Forward Dynamics," Submitted to *IEEE Trans. Robotics and Automation*.

[13] A. Fijany,"A New Class of Parallel and Pipeline Algorithms for Computation of the Manipulator Inertia Matrix," In preparation.

[14] A. Fijany and J.G. Pontnau,"Parallel Computation of the Jacobian for Robot Manipulators," Proc. IASTED Int. Conf. on Robotics & Automation, Santa Barbara, May 1987.

[15] C.S.G. Lee and P.R. Chang,"Efficient Parallel Algorithms for Robot Forward Dynamics Computation," *IEEE Trans. Syst. Man Cybern.*, Vol. SMC-18, no. 2, pp. 238-251, March/April 1988.

[16] C.S.G. Lee and P.R. Chang,"Efficient Parallel Algorithm for Robot Inverse Dynamics Computations," *IEEE Trans. Syst. Man Cybern.*, Vol. SMC-16(4), pp. 532-542, July/Aug. 1986.

[17] L.H. Lathrop,"Parallelism in Manipulator Dynamics," *Int. J. Robotics Res.*, Vol. 4(2), Summer 1985.

[18] S.A. Cook,"An Overview of Computational Complexity," *Com. ACM*, Vol. 26, June 1983.

[19] J.S. Vitter and R.A. Simons,"New Classes of Parallel Complexity: A Study of Unification and other Complete Problems for *P*," IEEE Trans. Computer, Vol. C-35(5), May 1986.

[20] H.T. Kung,"New Algorithms and Lower Bounds for the Parallel Evaluation of

Certain Rational Expressions and Recurrences," *J. of ACM*, Vol. 23, No. 2, pp. 252-261, April 1976.
[21] J. Miklosko and V.E. Kotov (Eds.), *Algorithms, Software and Hardware of Parallel Computers*. New York: Springer-Verlag, 1984.
[22] L. Hyafil and H.T. Kung,"The Complexity of Parallel Evaluation of Linear Recurrences," *J. ACM*, Vol. 24, No. 3, pp. 513-521, July 1977.
[23] L. Csanky,"Fast Parallel Matrix Inversion Algorithms," *SIAM J. of Computing*, Vol. 5, No. 4, pp. 618-623, Dec. 1976.
[24] A.H. Sameh and D.J. Kuck,"On Stable Parallel Linear System Solvers," *J. of ACM*, Vol. 25, No. 1, pp. 81-91, Jan. 1978.
[25] A. Bojanczyk, R.P. Brent, and H.T. Kung,"Numerically Stable Solution of Dense Systems of Linear Equations Using Mesh-Connected Processors," *SIAM J. of Stat. Comput.*, Vol. 5, No. 1, March 1984.

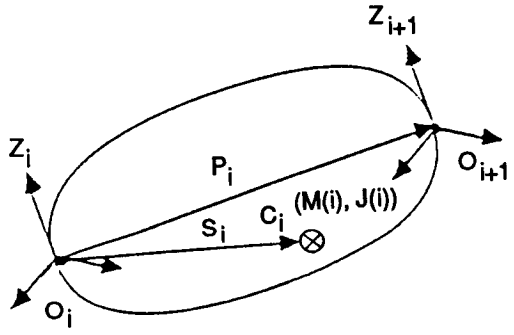

Fig. 2. Link, Frames, and Kinematic and Dynamic Parameters.

a_i, d_i, & α_i	Length, Distance, and Twist Angle of link i, respectively.
Q_i, $\dot{Q}_i$, & $\ddot{Q}_i$	Position, Velocity, and Acceleration of joint i, respectively.
M(i)	Mass of link i.
J(i)	Second moment of mass of link i about its center of mass (C_i).
H(i)	First moment of mass of link i about point O_i.
K(i)	Second moment of mass of link i about point O_i.
Z(i)	Axis of joint i
P(i,j)	Position vector from point O_j to point O_i.
R(i,j)	A 3x3 matrix representing the orientation of coordinate frame j with respect to coordinate frame i.
$\dot{\omega}$(i,j)	Angular acceleration of link i resulting from the unit acceleration of joint j.
$\dot{V}$(i,j)	Linear acceleration of link i (point O_i) resulting from the unit acceleration of joint j.
F(k+1,i,j)	Force exerted on point O_i due to the acceleration of links i through k, i.e., the links contained between points O_i and O_{k+1}, resulting from the unit acceleration of joint j.
N(k+1,i,j)	Moment exerted on point O_i due to the acceleration of link i through k, resulting from the unit acceleration of joint j.

Table I. Notation used in derivation of serial and parallel algorithms

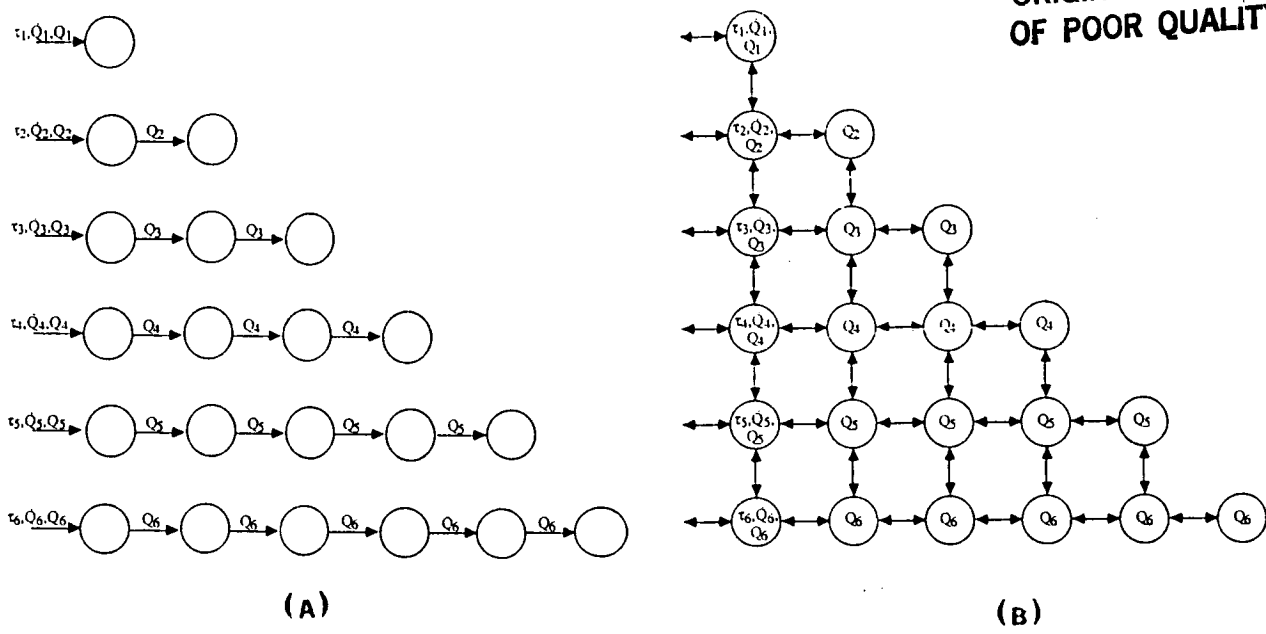

FIGURE 3. A TWO-DIMENSIONAL PROCESSOR ARRAY (A) DATA INPUT TO PROCESSOR ARRAY, (B) DISTRIBUTION OF INPUT DATA AMONG PROCESSORS.

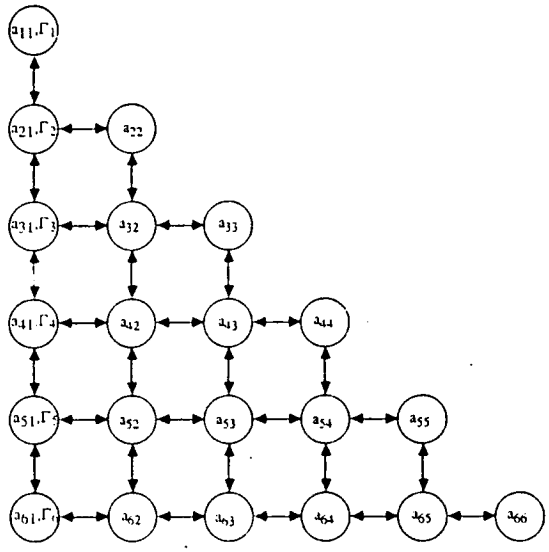

FIGURE 4. ORGANIZATION OF DATA RESULTING FROM COMPUTATION OF THE INERTIA MATRIX AND THE BIAS VECTOR.

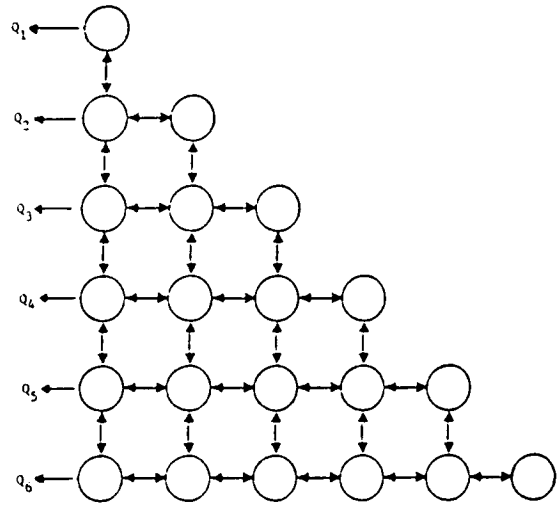

FIGURE 5. DATA OUTPUT FROM PROCESSOR ARRAY.