International Journal of Parallel Programming, Vol. 30, No. 1, February 2002 (© 2002)

Parallel Approaches for Singular Value Decomposition as Applied to Robotic Manipulator Jacobians[1]

Tracy D. Braun,[2,6] Renard Ulrey,[3]
Anthony A. Maciejewski,[4] and Howard Jay Siegel[5]

Received July 2000; revised February 2001

The system of equations that govern kinematically redundant robotic manipulators is commonly solved by finding the singular value decomposition (SVD) of the corresponding Jacobian matrix. This can require a considerable amount of time to compute, thus a parallel SVD algorithm reducing execution time is sought. The approach employed here lends itself to parallelization by using Givens rotations and information from previous decompositions. The key contribution of this research is the presentation and implementation of parallel SVD algorithms to compute the SVD for a set of Jacobians that represent various different joint failure scenarios. Results from implementation of the algorithm on a MasPar MP-1, an IBM SP2, and the PASM prototype parallel computers are compared. Specific issues considered for each implementation

[1] This research was supported in part by Sandia National Laboratories under contract numbers 18-4379B and AL-3001; and by Rome Laboratory under contract F30602-94-C-0022. T. D. Braun was also supported by a Purdue Benjamin Meisner Fellowship. The MasPar MP-1 was provided and supported in part by the National Science Foundation under grant number CDA-9015696.

[2] NOEMIX, 1425 Russ Blvd. Ste. T-110, San Diego, California 92101. E-mail: tdbraun@noemix.com

[3] AMCC Corporation, 4715 Innovation Dr., Fort Collins, Colorado 80525. E-mail: rulrey@amcc.com

[4] Electrical and Computer Engineering Department, Colorado State University, Fort Collins, Colorado 80523. E-mail: aam@engr.colostate.edu

[5] Electrical and Computer Engineering Department and Computer Science Department, Colorado State University, Fort Collins, Colorado 80523. E-mail: hj@colostate.edu

[6] To whom correspondence should be addressed.

include: how data is mapped to the processing elements, the effect that increasing the number of processing elements has on execution time, the type of parallel architecture used, and trade-offs between modes of parallelism.

KEY WORDS: IBM SP2; redundant manipulator; MasPar MP-1; robotic control; singular value decomposition.

1. INTRODUCTION

The singular value decomposition (SVD) of a matrix is a fundamental matrix decomposition that provides information useful for a wide range of applications (e.g., feature extraction, data reduction, and low rank approximation[1]). In many applications, rapid computation of the SVD is necessary, e.g., when the SVD is used for solving the systems of equations for real-time motion control of robotic manipulators. Consider the kinematics of a robotic manipulator with $\underline{n}$ joints operating in $\underline{m}$ dimensions. Jacobians of size $m = 6$ and $n = 7$ were chosen for this study because they represent the minimum size for a redundant manipulator operating in three dimensions and manipulators of this size are commercially available. However, the algorithms presented are completely general and can be used for arbitrary values of m and n. Let $\underline{\dot{x}} \in \mathbb{R}^m$ specify the manipulator's end-effector velocity, $\underline{\dot{\theta}} \in \mathbb{R}^n$ denote the joint velocities of the manipulator, and $\underline{J} \in \mathbb{R}^{m \times n}$ be the manipulator Jacobian matrix. The kinematics of the robotic manipulator can then be represented by the equation $\dot{x} = J\dot{\theta}$. The ability to compute the SVD of J in real time allows for evaluation of proximity to singularities and dexterity optimization.[2]

In general, techniques for computing the SVD of an arbitrary matrix involve iterating an unknown number of times until a data-dependent convergence criterion is met. Therefore, the number of operations required is not known *a priori* and guaranteeing real-time computation of the SVD is difficult. However, for this application the current Jacobian matrix, $J(t)$, can be regarded as a perturbation of the previous Jacobian matrix, i.e., $J(t) = J(t - \Delta t) + \Delta J(t)$. Knowledge of this previous state can be used to decrease computational complexity during calculation of the current SVD.[3, 4]

This article presents a technique based on research performed by Maciejewski and Klein[3] and Roberts and Maciejewski[5] for real-time calculation of the SVD of a Jacobian for a manipulator experiencing locked-joint failures. The performance of this technique is analyzed on three different parallel architectures: an SIMD (single instruction stream, multiple data stream) MasPar MP-1,[6] an MIMD (multiple instruction stream, multiple data stream) IBM SP2,[7] and the mixed-mode PASM (partitionable SIMD/MIMD) prototype.[8, 9] Specific issues considered for each implementation include how data is mapped to the processing elements, the effect that increasing the number of processing elements has on execution time, the type of parallel architecture used, and trade-offs between modes of parallelism. Case studies, such as the one presented here, are a necessary step in developing software tools for mapping an application task onto a single parallel machine, and for mapping an application task onto a heterogeneous suite of parallel machines, where a choice among different types of machines is possible.

The remainder of this article is arranged as follows. The SVD procedure from Maciejewski and Klein(3) is reviewed in Section 2. Section 3 describes an SVD technique for a set of Jacobians representing various different joint failure scenarios. Section 4 examines different mappings of this technique onto parallel machines. The results obtained from the MasPar MP-1, IBM SP2, and PASM prototype are summarized in Sections 5–7, respectively. Finally, Section 8 summarizes the results of this application study.

2. BACKGROUND INFORMATION

2.1. Kinematically Redundant Manipulators

Robotic manipulators play a key role in structured industrial settings as well as in remote and/or hazardous environments such as in space or undersea exploration. Failures in these robots can have significant consequences, ranging from economic impact to potentially catastrophic incidents. As mentioned in Section 1, the kinematics of a manipulator with n joints operating in m dimensions is typically described by the equation

$$\dot{x} = J\dot{\theta} \tag{2.1}$$

where $\dot{x} \in \mathbb{R}^m$ specifies the manipulator's end-effector velocity, $\dot{\theta} \in \mathbb{R}^n$ denotes the joint velocities of the manipulator, and $J \in \mathbb{R}^{m\times n}$ represents the manipulator Jacobian matrix.

One method used to increase fault tolerance, and prevent the consequences just mentioned, is to use kinematically redundant manipulators. Kinematically redundant robotic manipulators have a greater number of independently controlled joints than are necessary to achieve the desired degree of motion, i.e., $m < n$. The available redundancy in these manipulators allows for optimization of some secondary criterion once the requirements of the specified end-effector velocity are met. Section 3 will discuss one such criterion, fault tolerance. Other secondary criteria that have been investigated include obstacle avoidance,(10) singularity avoidance,(11) and dexterity optimization.(2)

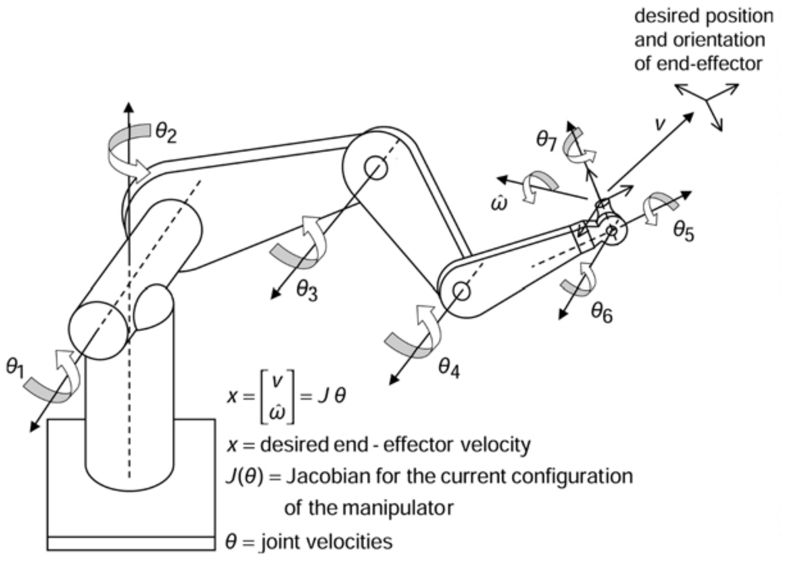
desired position
and orientation
of end-effector
v
θ_7
θ_2
$\hat{\omega}$
θ_5
θ_3
θ_6
θ_1
θ_4
$x = \begin{bmatrix} v \\ \hat{\omega} \end{bmatrix} = J\,\theta$
x = desired end - effector velocity
$J(\theta)$ = Jacobian for the current configuration of the manipulator
θ = joint velocities

Fig. 1. Figure showing the relationship represented by $\dot{x} = J\dot{\theta}$ for a kinematically redundant manipulator with $n = 7$ joints operating in $m = 6$ dimensions (3 translational and 3 rotational).

Figure 1 shows an example manipulator with $n = 7$ joints operating in $m = 6$ (three linear and three rotational) dimensions. In Fig. 1, each θ_i represents the velocity for joint i. The manipulator is maneuvering the end-effector to the desired position and orientation, shown in the upper-right of Fig. 1.

2.2. The SVD and Pseudoinverse

The underlying mathematical background behind the SVD and Givens rotations is presented in the rest of this section. The equations derived will form the basis of the computations performed on the parallel architectures, as explicated in subsequent sections.

For a matrix $J \in \mathbb{R}^{m \times n}$, let $\underline{U} \in \mathbb{R}^{m \times m}$ denote an orthogonal matrix of output singular vectors, $\underline{V} \in \mathbb{R}^{n \times n}$ represent an orthogonal matrix of input singular vectors, and $\underline{D}$ be a nonnegative diagonal matrix. Then, the SVD of J is defined as the matrix factorization

$$J = UDV^T = \sum_{i=1}^{m} \sigma_i \hat{u}_i \hat{v}_i^T \tag{2.2}$$

where D has the form

$$D \in \mathbb{R}^{m\times n} = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \ddots & \vdots & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & 0 & \vdots & & \vdots \\ 0 & \cdots & 0 & \sigma_m & 0 & \cdots & 0 \end{bmatrix} \tag{2.3}$$

and $\underline{\hat{u}_i}$ and $\underline{\hat{v}_i}$ represent the ith column of U and V, respectively. The singular values, $\underline{\sigma_i}$, are ordered so that $\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant \sigma_m \geqslant 0$. It is assumed in Eq. (2.3), and for the remainder of this paper, that $m \leqslant n$.

Once the SVD has been computed, it can easily be manipulated to form the *pseudoinverse*, $\underline{J}^\dagger$, of the matrix J. The pseudoinverse is given by the equation

$$J^\dagger = VD^\dagger U^T = \sum_{i=1}^{m} \frac{1}{\sigma_i} \hat{v}_i \hat{u}_i^T \tag{2.4}$$

where $J^\dagger \in \mathbb{R}^{n\times m}$ and $\underline{D}^\dagger$ is defined as

$$D^\dagger \in \mathbb{R}^{n\times m} = \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_2} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \frac{1}{\sigma_m} \\ 0 & \cdots & & 0 \\ \vdots & & & \vdots \\ 0 & \cdots & & 0 \end{bmatrix} \tag{2.5}$$

with the ith diagonal term equal to 0 if $\sigma_i = 0$. The pseudoinverse can then be used to find a solution to Eq. (2.1). That is, $\dot{\theta}$ can be found via $\dot{\theta} = J^\dagger \dot{x}$, giving the minimum joint velocities to move the end-effector as close as physically possible to the manner specified by $\dot{x}$. In addition, the $n-m$ portion of V that provides an orthonormal basis for the null space of J allows one to characterize **all** solutions of Eq. (2.1).

An efficient algorithm for computing the SVD of a Jacobian matrix is therefore useful for the control of redundant manipulators. The most common technique for computing the SVD of a matrix is the Golub–Reinsch algorithm.(12) However, when attempting to parallelize this method, there are two drawbacks. First, it is not straightforward to incorporate into this method information from the SVD of a perturbed matrix to update the SVD. Secondly, this method is relatively sequential in nature. Therefore, algorithms more receptive to parallelization are desirable.

2.3. Givens Rotations

While there have been several parallel SVD algorithms written and implemented on various machine architectures,(13–17) they do not exploit any particular characteristic of the matrix being decomposed. For this work, the fact that the current Jacobian matrix is a perturbation of the previous Jacobian matrix can be exploited.

The techniques studied here are based on a methodology presented in Maciejewski and Klein.(3) This algorithm uses Givens rotations(18, 19) to orthogonalize two columns of a matrix. *Givens rotations* are orthogonal transformations that can be represented as a matrix multiplication of the form

$$Q_{ij} = \begin{array}{cc} \left[\begin{array}{ccccccccccc} 1 & & & \cdot & & & & \cdot & & & \\ & \ddots & & \cdot & & & & \cdot & & & \\ & & 1 & \cdot & & & & \cdot & & & \\ \cdot & \cdot & \cdot & \cos(\phi) & \cdot & \cdot & \cdot & -\sin(\phi) & \cdot & \cdot & \cdot \\ & & & \cdot & 1 & & & \cdot & & & \\ & & & \cdot & & \ddots & & \cdot & & & \\ & & & \cdot & & & 1 & \cdot & & & \\ \cdot & \cdot & \cdot & \sin(\phi) & \cdot & \cdot & \cdot & \cos(\phi) & \cdot & \cdot & \cdot \\ & & & \cdot & & & & \cdot & 1 & & \\ & & & \cdot & & & & \cdot & & \ddots & \\ & & & \cdot & & & & \cdot & & & 1 \end{array}\right] & \begin{array}{c} \\ \\ \\ i \\ \\ \\ \\ j \\ \\ \\ \\ \end{array} \\ \begin{array}{ccccccccccc} & & & i & & & & j & & & \end{array} & \end{array} \tag{2.6}$$

where $\underline{Q}_{ij}$ orthogonalizes columns i and j. All other elements not shown off the main diagonal in Eq. (2.6) are zero. The matrix Q_{ij} can be interpreted geometrically as a rotation of ϕ radians in the i–j plane. Notice that post-multiplication by Q_{ij} will only affect columns i and j of a matrix.

2.4. Using Givens Rotations to Find the SVD

Consider an orthogonal matrix V, generated by successive Givens rotations. (The actual construction of this V matrix is discussed shortly.) Assume that this V matrix results in

$$JV = B \tag{2.7}$$

where $\underline{B} \in \mathbb{R}^{m \times n}$, and the columns of B, denoted $\underline{b}_i$, are orthogonal. The matrix B can then be decomposed into two matrices, an orthogonal matrix (U) and a diagonal matrix (D) resulting in

$$B = UD \tag{2.8}$$

This is accomplished by letting the columns of U, denoted $\underline{\hat{u}}_i$, be equal to the normalized versions of the columns of B, i.e.,

$$\hat{u}_i = \frac{b_i}{\|b_i\|} \tag{2.9}$$

and then defining the diagonal elements of D to be equal to the norm of the columns of B, i.e.,

$$\sigma_i = \|b_i\|, \qquad \text{where} \quad \|b_i\| = \sqrt{b_i^T b_i} \tag{2.10}$$

Using Eq. (2.8) in Eq. (2.7) and solving for J results in the SVD of J, as given in Eq. (2.2).

The critical step in the above procedure is determining the orthogonal matrix V that will orthogonalize the columns of J in Eq. (2.7). This V matrix is constructed as a product of Givens rotations, each of which is designed to orthogonalize two columns. Consider the ith and jth columns of an *arbitrary* matrix $\underline{A}$. Post-multiplication by a Givens rotation results in two new columns, a_i' and a_j', given by

$$a_i' = a_i \cos(\phi) + a_j \sin(\phi) \tag{2.11}$$

$$a_j' = a_j \cos(\phi) - a_i \sin(\phi) \tag{2.12}$$

The constraint that these columns be orthogonal results in $\cos(\phi)$ and $\sin(\phi)$ terms that are calculated using formulas given in Nash.[20] These formulas are based on the following terms:

$$p = a_i^T a_j \tag{2.13}$$

$$q = a_i^T a_i - a_j^T a_j \tag{2.14}$$

$$c = \sqrt{4p^2 + q^2} \tag{2.15}$$

For cases when $q \geqslant 0$

$$\cos(\phi) = \sqrt{\frac{c+q}{2c}} \qquad \text{and} \qquad \sin(\phi) = \frac{p}{c \cdot \cos(\phi)} \tag{2.16}$$

and when $q < 0$

$$\sin(\phi) = \operatorname{sgn}(p)\sqrt{\frac{c-q}{2c}} \qquad \text{and} \qquad \cos(\phi) = \frac{p}{c \cdot \sin(\phi)} \tag{2.17}$$

where

$$\operatorname{sgn}(p) = \begin{cases} 1 & \text{if} \quad p \geqslant 0 \\ -1 & \text{if} \quad p < 0 \end{cases} \tag{2.18}$$

The two sets of formulas [Eqs. (2.16) and (2.17)] are given so that ill-conditioned equations resulting from the subtraction of nearly equal numbers can always be avoided. [For example, if during the computation of the Givens rotation $q \approx -c$, then $\cos(\phi) \approx 0$ and there would be a serious loss in precision when calculating $\sin(\phi)$ using Eq. (2.16).]

The preceding discussion describes a single Givens rotation that will orthogonalize two columns of a given matrix. For a matrix with n columns, $n(n-1)/2$ rotations are required to orthogonalize each possible pair of columns. This set of $n(n-1)/2$ rotations is referred to as a *sweep*.[12] Multiple sweeps are generally required to obtain a fully orthogonal matrix. The matrix V in Eq. (2.7) can therefore be computed based on J as the series of sweeps such that

$$V = \prod_{\#\,\text{sweeps}} \left(\prod_{i=1}^{n-1} \prod_{j=i+1}^{n} Q_{ij} \right) \tag{2.19}$$

Unfortunately, the number of sweeps required to orthogonalize the columns of J is usually not known *a priori*. However, this problem can be circumvented by considering the current Jacobian matrix to be a perturbation of the previous Jacobian, i.e.,

$$J(t) = J(t - \Delta t) + \Delta J(t) \tag{2.20}$$

Using this information, it was shown that a good approximation for the current SVD could be obtained from the previous SVD information in a single sweep if ΔJ is small.(3) That is, using

$$U(t)\, D(t) = B(t) \approx J(t)\, V(t - \Delta t) \tag{2.21}$$

and applying one sweep of Givens rotations, the current SVD can be found. Note that the accuracy of this approximation depends on the size of Δt, i.e., the control cycle time of the robot. Therefore, in this work, Eq. (2.19) is calculated using

$$V(t) = V(t - \Delta t) \left(\prod_{i=1}^{n-1} \prod_{j=i+1}^{n} Q_{ij} \right) \tag{2.22}$$

Thus, the previous V matrix is updated using only a single sweep. Because it is known *a priori* that only one sweep of rotations will be performed, the computational expense associated with iterating to convergence is eliminated. Note that the initial V matrix can either be computed or assumed to be known based on the initial starting position of the manipulator.

2.5. Summary

This section reviewed the SVD, the pseudoinverse, and an SVD algorithm that uses Givens rotations. The penalties of convergence checking and uncertainty in the number of iterations are removed by this methodology. This approach assumes that SVD information from the previous state is available, and considers the current Jacobian matrix to be a perturbation of this previous state [Eq. (2.20)].

The approach described begins with an initial estimate of the current B matrix [Eq. (2.21)]. Givens rotations [described by Eqs. (2.6), (2.11)–(2.18)] are then applied to complete the orthogonalization of B. These same Givens rotations are then applied to V [Eq. (2.22)]. The result of these rotations is Eq. (2.7). From Eq. (2.7), the current SVD is easily derivable via Eqs. (2.8)–(2.10).

3. ALGORITHM DESCRIPTIONS

3.1. Motivation

A robotic manipulator can have joint failures. The calculations described by the equations in Section 2 assume no joint failures, and thus determine the *fault-free SVD*. However, the goal of this study is not only to find the fault-free SVD, but also to find the SVD of these Jacobians should a single joint fault occur in the manipulator. It is assumed that such a single joint fault can be detected by the manipulator and the joint locked into position. Let the columns of the Jacobian matrix be denoted $\underline{j_i}$. Therefore, the Jacobian for the post-fault manipulator, denoted ${}^f\underline{J}$, becomes ${}^fJ = [\, j_1 \;\; j_2 \cdots j_{f-1} \;\; \mathbf{0} \;\; j_{f+1} \cdots j_n]$, where joint $\underline{f}$ has failed and the column j_f is replaced with $j_f = \mathbf{0}$. The SVD of each ${}^fJ$ is referred to here as the *post-fault SVD*. The post-fault SVDs are computed for the manipulator even when no faults have occurred because this information can be used in several ways, including trajectory planning for maintaining maximum failure tolerance.(5, 21)

The next technique described for finding the fault-free SVD and post-fault SVDs was implemented on three different parallel machines: the MasPar MP-1, the IBM SP2, and the PASM prototype. The study of parallel approaches is justified because the algorithm is applicable for the real-time control of arbitrarily large arms and arm systems and can be applied to combinations of multiple joint failures.

3.2. Single Sweep SVD Approximation

Figure 2 shows the basic single sweep SVD algorithm. In step 1, the previous V matrix and the current J matrix are used to calculate a good initial estimate for the current B matrix. This is done using Eq. (2.21).

```
step 1: calculate initial estimate for $B$ from $J$ and previous $V$, using Eq. (2.21)
step 2: for all column pairs $(i, j)$ of $B$ do
            /* one sweep */
            calculate $p$, $q$, and $c$ using Eqs. (2.13)–(2.15).
            calculate $\cos(\phi)$ and $\sin(\phi)$, using Eq. (2.16) or (2.17)
            perform rotation on columns $i$ and $j$ of $B$, similar to Eqs. (2.11) and
            (2.12)
            perform rotation on columns $i$ and $j$ of $V$, similar to Eqs. (2.11) and
            (2.12)
        end for
step 3: calculate $D$ from $B$, using Eq. (2.10)
        calculate $U$ from $B$ and $D$, using Eq. (2.9)
```

Fig. 2. High-level single sweep SVD algorithm using Givens rotations.

Step 2 performs the single sweep of Givens rotations. First, based on the selected columns of the current B, values for p, q, and c are calculated using Eqs. (2.13)–(2.15). Next, $\cos(\phi)$ and $\sin(\phi)$ are calculated using Eqs. (2.16) or (2.17). Then, a Givens rotation is performed on columns i and j of B. This is done using b_i and b_j for a_i and a_j, respectively, in Eqs. (2.11) and (2.12). Now, the V matrix must also be updated (to maintain Eq. (2.7)). Therefore, a Givens rotation is also performed on columns i and j of V, denoted $\underline{\hat{v}_i}$ and $\underline{\hat{v}_j}$. This rotation is done using $\hat{v}_i$ and $\hat{v}_j$ for a_i and a_j, respectively, in Eqs. (2.11) and (2.12). Note that both rotations use the same $\sin(\phi)$ and $\cos(\phi)$ values.

It is assumed in step 2 that the use of the previous decomposition information allows the algorithm to converge in a single sweep. Step 3 then provides the current singular values through the straightforward computations in Eqs. (2.9) and (2.10).

3.3. Fault-free and Post-fault SVD Approximations

The post-fault approximation technique assumes that the decomposition information available from the previous time period includes post-fault matrices. That is, to compute the current post-fault singular values, ${}^fD(t)$, for the current post-fault Jacobian, ${}^fJ(t)$, the previous post-fault matrix, ${}^fV(t-\Delta t)$, is used in Eqs. (2.21) and (2.22). Thus, the post-fault SVD approximation technique can be represented by the equations

$$ {}^fJ(t)\,{}^fV(t-\Delta t) \approx {}^fU(t)\,{}^fD(t) \tag{3.1}$$

$$ {}^fV(t) = {}^fV(t-\Delta t)\left(\prod_{i=1}^{n-1}\prod_{j=i+1}^{n} Q_{ij}\right) \tag{3.2}$$

To compute the fault-free and post-fault SVD approximations, the algorithm from Fig. 2 is simply performed $n+1$ times, as shown in Fig. 3. Using this notation, $f=0$ represents the calculation of the fault-free SVD, i.e., the SVD for the Jacobian with no columns zeroed out, and $f=1,\ldots,n$ represent the n post-fault Jacobians with column $1,\ldots,n$ zeroed out,

```
for all $f \in \{0, 1, \ldots, n\}$ do
    /* failure in joint $f$ (where $f = 0$ represents no fault) */
    perform single sweep SVD algorithm on ${}^fJ$
end for
```

Fig. 3. To compute the fault-free SVD and all post-fault SVDs, the algorithm in Fig. 3 is executed $n+1$ times.

respectively. Given that computing ${}^fJ(t)$ is simply replacing j_f with $j_f = \mathbf{0}$, and the assumption that ${}^fV(t - \Delta t)$ is available from the previous time period, each iteration of the for all f loop in Fig. 3 is independent of previous iterations. Thus, on a parallel machine with enough processors, the entire algorithm from Fig. 3 could be performed such that each processor would only have to perform one instance of the algorithm from Fig. 2 (e.g., processor i could perform the algorithm from Fig. 2 for ${}^fJ$ with $f = i, 0 \leqslant i \leqslant n$).

3.4. Generation of Test Data

To evaluate the performance of the proposed SVD algorithm, a set of test Jacobian matrices was generated. The input to the SVD algorithm is the current Jacobian matrix $J(t)$ (each ${}^fJ(t)$ is generated within the algorithm itself). Because it is assumed that the SVD of the previous Jacobian, $J(t - \Delta t)$, is available from the previous control cycle time, these values had to be computed *a priori*.

To accurately represent all possible scenarios, the 6×7 Jacobian matrices were randomly generated, uniformly distributed over all possible Jacobians for rotary jointed manipulators. Each column of these Jacobians is of the form

$$\mathbf{j}_i = \begin{bmatrix} \mathbf{v}_i \\ \hat{\omega}_i \end{bmatrix} \tag{3.3}$$

where $\underline{\hat{\omega}_i}$ is of unit-length, and pointing uniformly over all directions. Given $\overline{\hat{\omega}}_i$, the vectors $\underline{\mathbf{v}_i}$ have directions uniformly distributed over the subspace orthogonal to $\overline{\hat{\omega}}_i$, with a length that is uniformly distributed over $[0, 2]$. This distribution was intended to represent a reasonably normalized Jacobian that has accounted for the disparity in units between linear and rotational velocities.(21) This file represented the current Jacobian matrices, $J(t)$, for which the new SVD was desired.

Next, the Denavit–Hartenburg parameters(22) for each Jacobian matrix were found. The configuration of the manipulator, given by θ, was then perturbed by a small amount, $\underline{\|\Delta\theta\|}$, simulating movement by the manipulator. For the experimental timing results in subsequent sections, the value $\|\Delta\theta\| = 0.57$ degrees (0.01 radians) was used. The Jacobian corresponding to this new, perturbed configuration was used to represent the "previous" Jacobian matrix, $J(t - \Delta t)$. Then, the SVD for each previous Jacobian matrix and each previous single joint, post-fault Jacobian matrix was computed using MATLAB.(23)

Recall Fig. 1, which illustrates an example manipulator with $n = 7$ joints operating in $m = 6$ (three linear and three rotational) dimensions. In Fig. 1, the vector $\mathbf{v}$ represents the linear velocity required to maneuver the end-effector of the manipulator to the desired position. The vector $\hat{\omega}$ represents the rotational velocity required to rotate the end-effector of the manipulator to the desired orientation.

4. PARALLEL MAPPINGS CONSIDERED

4.1. Overview

Distributed memory parallel architectures generally consist of a number of *processing elements* (*PEs*). A PE is a combination of a processor and a memory module, allowing the processor fast access to the local memory. This section focuses on techniques to efficiently distribute the data and computations of the single sweep SVD algorithm among the PEs of a parallel architecture, with the goal of decreasing overall execution time. These techniques are easily extended to the fault-free and post-fault SVD algorithm. Specifically, three data mapping techniques are described: 1CPP, 2CPP, and column segmentation.

4.2. One Column Per PE (1CPP)

From the algorithm in Fig. 2, notice that no inter-PE communication is required in step 1 if each PE has one column of V, a copy of the entire Jacobian matrix, and one column of B. In step 3, single-column operations can also be performed simultaneously on n PEs. The columns of U and the singular values of D are computed from the corresponding columns of B. Again, no inter-PE communications are required.

Step 2, however, cannot be performed without inter-PE communications for this one column per PE (*1CPP*) distribution. Consider the n columns currently held on n PEs, one per PE. For a given pairing of PEs, there are $n/2$ PE pairs, and hence $n/2$ column pairs. Let the parallel execution of a Givens rotation on $n/2$ column pairs by the PEs be defined as a *rotation step*. Then, a minimum of $n-1$ rotation steps must be performed to generate all $n(n-1)/2$ possible column pairings that constitute one sweep. During each rotation step (step 2), the 1CPP approach used here performs several inter-PE communications. Columns of B are exchanged and columns of V are exchanged [to form new column pairs, used in Eqs. (2.11) and (2.12), and to compute p in Eq. (2.13)]. The PEs also exchange $b_i^T b_i$ to calculate q [Eq. (2.12)]. [Note: An alternative would be to let two PEs calculate both $b_i^T b_i$ and $b_j^T b_j$ locally (i.e., redundantly), for a given i and j. For the machines considered, the exchange approach taken was based on the system architectures.]

The 1CPP communication pattern that was employed is shown in Fig. 4. This column transfer method formed all possible column pairings using only $n-1$ column transfers. This was implemented as follows. Assume there are n PEs, numbered from 0 to $n-1$. PE i always contains the most recent version of column $i+1$ (columns are numbered 1 to n). In the kth rotation step, $1 \leqslant k < n$, PE i exchanges its column of data with PE $i \oplus k$, where $\oplus$ is the bit-wise exclusive-or (for all i, $0 \leqslant i < n$).

4.3. Two Column Per PE (2CPP)

A two column per PE (*2CPP*) approach that distributes pairs of columns of V and B to $n/2$ PEs was also implemented. This 2CPP approach reduces the frequency and complexity of the inter-PE communications (because each PE will hold a pair of columns to be orthogonalized). For the 2CPP approach, step 1 and step 3 are performed concurrently without any inter-PE communications, similar to the 1CPP approach.

Using the 2CPP approach, the first rotation step of step 2 can be performed without any inter-PE communications. In contrast to the 1CPP method, the 2CPP method only requires the exchange of columns b_i and $\hat{v}_i$ so that each PE can obtain a new column pair for orthogonalization. (There is no need to exchange $b_i^T b_i$ and $b_j^T b_j$ because both are located on the same PE, so q can be calculated locally without any redundant computations.)

Although it is obvious that step 1 and step 3 of the single sweep SVD algorithm will take twice as long to compute using the 2CPP distribution versus the 1CPP distribution, the 2CPP implementation does not require twice as much total time to execute. This is due to the fact that the highest percentage of the total execution time for the single sweep SVD algorithm is spent performing step 2, where 2CPP has the advantage of fewer communications.

The 2CPP communication technique implemented is shown in Fig. 5. Initially, all PEs being used are in a single communicating subgroup. Let each PE contain two columns, $\underline{x}$ and $\underline{y}$. All possible column pairs are formed by repeating the following process. In the first phase, all y columns are shifted right one PE at a time through all other PEs in the subgroup. In the second phase, each subgroup is split into two new subgroups, the left subgroup and the right subgroup. The PEs in the left subgroup exchange their y column with an x column from the right subgroup. This two phase process is repeated with the new subgroups until all $n-2$ column transfers have been performed. Different 2CPP procedures can be found in Chuang and Chen,[15] Maciejewski and Reagin,[4] and Schimmel and Luk.[17]

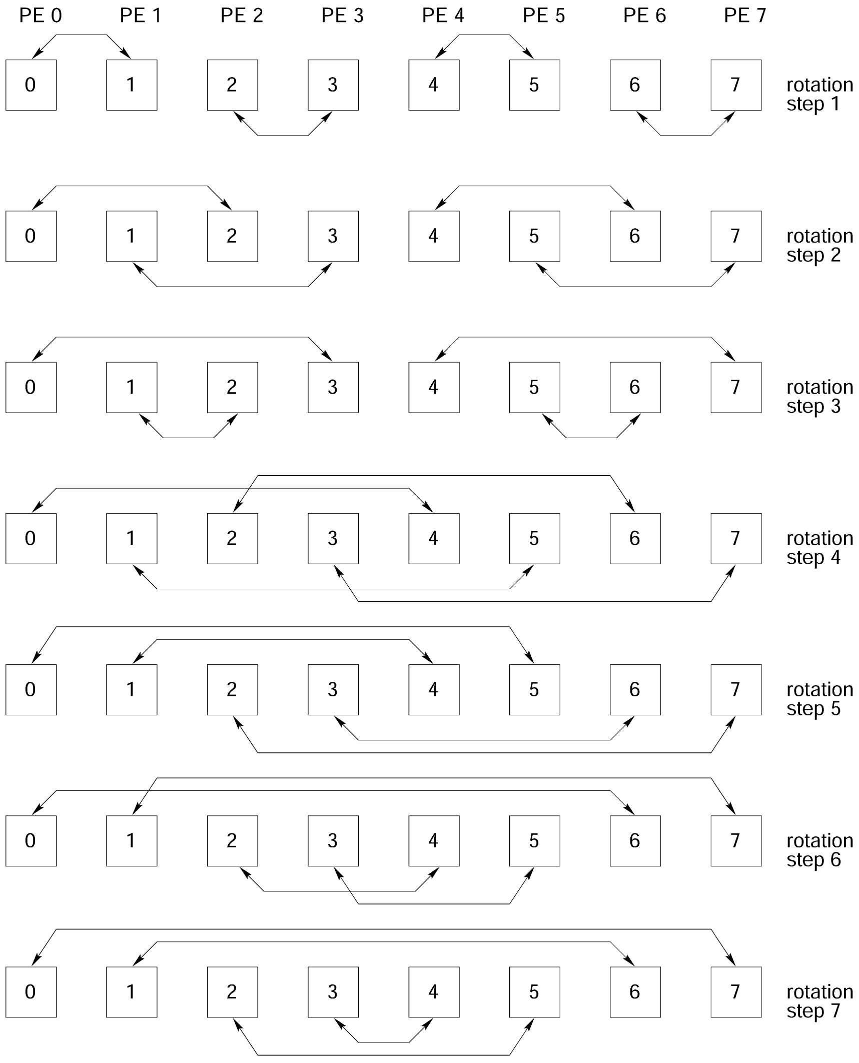

Fig. 4. Method for performing inter-PE column transfers for one sweep in the 1CPP algorithm mapping.

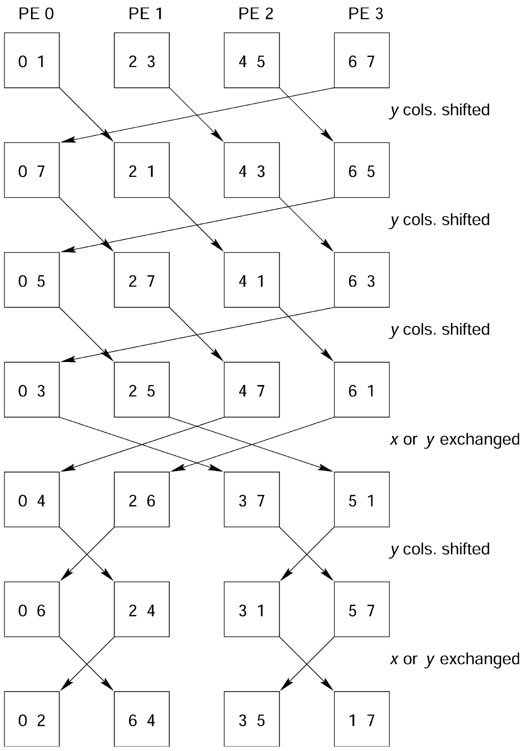

Fig. 5. Method for performing inter-PE column transfers for one sweep in the 2CPP algorithm mapping.

4.4. Column Segmentation

One goal of this study was to extract as much parallelism as possible from both the algorithm and the target machines to reduce execution time. The technique described here divides each column vector of the B and V matrices into $\underline{r}$ segments, where r is a power of two. This variable r represents the *column segmentation* of the data (and operations) among PEs and increases the total number of PEs used by a factor of r. Values of $r \in \{1, 2, 4, 8\}$ were implemented. When $r > 1$, each PE only operates on a column segment (column segments are numbered $0 \cdots r-1$). The goal of this column segmentation was to decrease execution time by decreasing the number of computations per PE, while incurring a minimum number of additional communications.

This segmentation of the column data does not interfere with the inter-PE column transfers for the 1CPP and 2CPP methods. Column transfers simply take place between PEs containing the same segment number. Inter-PE communication also occurs among PEs containing different segments of the same column (e.g., to add up partial sums).

Figure 6 outlines how column segmentation affects the matrix multiply during step 1 of the single sweep SVD algorithm (Fig. 2). Step 1 of the SVD algorithm finds an initial estimate for the B matrix by multiplying the current Jacobian by the previous V matrix, based on Eq. (2.21). Each PE has a copy of the entire J matrix, but columns of V (and the resulting columns of B) are distributed among multiple PEs. The figure shows the multiplication of J by $\hat{v}_i$ to find column b_i when the columns are divided into $r = 4$ segments.

The diagonally cross-hatched segment of $\hat{v}_i$ in Fig. 6 is the only segment of $\hat{v}_i$ resident on the PE under consideration. Therefore, this PE is responsible for deriving the corresponding segment of b_i that is also shown with diagonal cross-hatching. To fully derive this segment of b_i, the entire column of $\hat{v}_i$ must be multiplied by the section of J shown between the two dashed lines. However, because only the diagonally cross-hatched segment of $\hat{v}_i$ is resident on the PE, the only information that is obtainable by this PE is a partial sum of the entire column b_i. This partial sum for the entire column b_i is calculated by multiplying the resident segment of $\hat{v}_i$ by the corresponding columns of J. These columns of J, and the resulting partial sum for b_i, are shown by the jagged cross-hatching in Fig. 6. (Segments with no cross-hatching of any kind in Fig. 6 are unused or unavailable on this PE.)

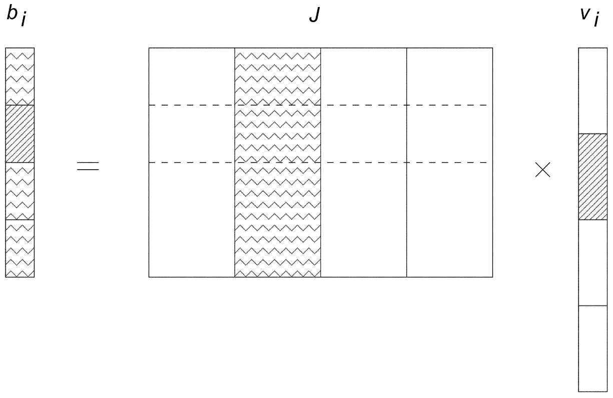

Fig. 6. Diagram of matrix multiply to find the partial sum of $b_i = J \times \hat{v}_i$ resident on this PE, with column segmentation $r = 4$.

After all r PEs that contain segments of $\hat{v}_i$ perform the partial matrix multiplication, each of these PEs contains a unique partial sum of the entire column b_i. For any PE to obtain the final result for its segment of b_i, that PE must combine its partial sum of the segment with the corresponding partial sums of the segments from the $r-1$ other PEs. That is, for the PE in Fig. 6 to obtain the final result for the diagonally cross-hatched segment of b_i, it must obtain the corresponding segment of b_i from the other $r-1$ PEs that have a segment of $\hat{v}_i$, and add the partial sums together. Likewise, the PE in Fig. 6 will send its $r-1$ segments of b_i with jagged cross-hatching to the corresponding PEs combining those partial sums.

Several methods were investigated for performing this combining of partial sums. First, standard recursive doubling[24] was considered. In standard recursive doubling, the r PEs containing segments of b_i are partitioned into two halves. Each PE in one partition sends its partial sum to a PE in the other partition. The sending PEs then become disabled. PEs remaining enabled add this partial sum to their own, and the process is repeated. The result is eventually found in a single PE. This method would require $\log_2(r) \times r$ inter-PE communication steps to find the final sum for each of the r segments. Each block of data transferred would have size $\lceil \frac{m}{r} \rceil$, and each enabled PE would perform $\lceil \frac{m}{r} \rceil$ additions after each communication. All n groups (of r PEs each) can do this simultaneously.

Next, full recursive doubling[24] was implemented. Typically, in this variation of recursive doubling, each of the r PEs sharing a column remains enabled, and performs the send-receive-add sequence. The result will then end up on all PEs involved in the recursive doubling. Therefore, only $\log_2(r)$ such recursive doubling steps are required. However, because each PE is computing a different final sum, each transfer consists of m column elements, and each PE must perform m additions after each communication. Again, all n groups (of r PEs each) can do this simultaneously.

A third method, called *segment combining* was also investigated. This method requires fewer communication steps than standard recursive doubling, and smaller blocks of data being transferred than full recursive doubling.

Segment combining is illustrated in Fig. 7. In Fig. 7, the arrows indicate source/destination positions of column segments, and are meant to wrap around the diagram from right to left. Segment combining requires $r-1$ communication steps. There are $r=4$ column segments, so three combining steps are required. The diagonally cross-hatched sections show the segments being combined by each PE. PE $(i+n)$ in Fig. 7 is meant to represent the same PE that was considered in Fig. 6. That is, the column b_i with diagonal and jagged cross-hatching in PE $(i+n)$ in Fig. 7 corresponds to the same column and cross-hatching from Fig. 6.

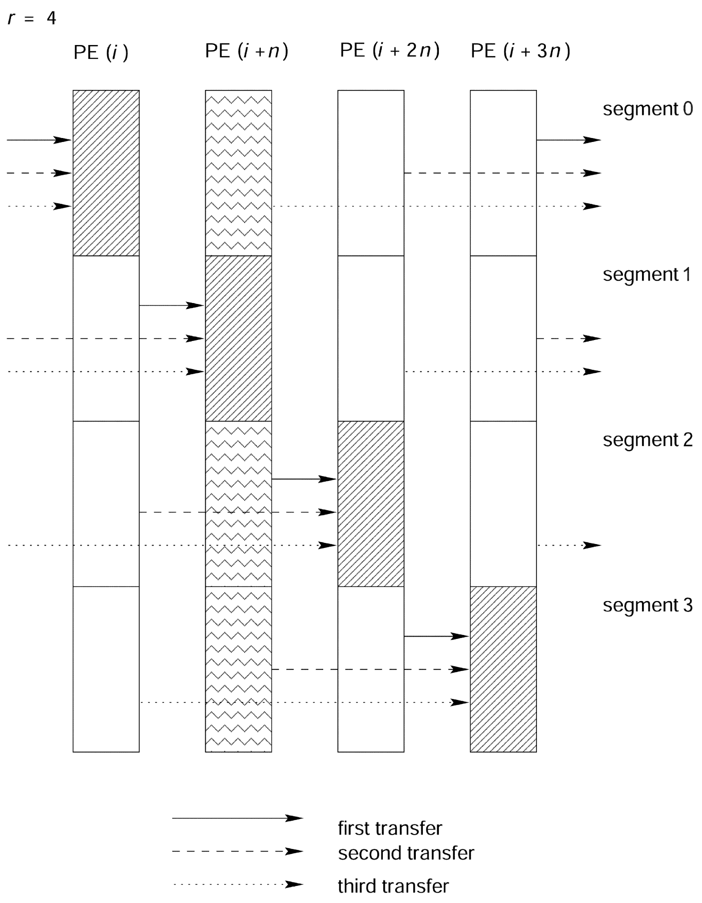

Fig. 7. Diagram of segment combining method for a single column with $r = 4$. PE $(i + jn)$ accumulates the values for segment j of b_i.

Recall that each PE contains a unique partial sum of the entire column b_i. Each PE is responsible for computing the final result for only its segment of column b_i. To do this, each PE must obtain the corresponding segment of column b_i from the other $r - 1$ PEs. Let $\underline{z}$ denote the current communication step, $1 \leqslant z \leqslant r - 1$. During each communication step of segment combining, each PE sends a different segment of b_i to a different destination PE. The segment that each PE sends and the destination for that segment is based on modulo r addition.

For example, in Fig. 7, let, $r = 4$, $z = 1$, and $n = 8$. Also, for the sake of explanation, assume there are only $rn = 32$ PEs, numbered from 0 to 31.

In general, PE k will operate on segment $j = \lfloor k/n \rfloor$ of column $(k \bmod n)$. During the first communication step (i.e., $z = 1$), PE k sends its copy of segment $((j+z) \bmod r) = ((j+1) \bmod 4$ to PE $(k + (zn) \bmod (rn)) = (k + (8) \bmod 32)$. This is repeated for $z = 2$ and $z = 3$. After which, every PE will have received every partial sum that it needs.

Thus, column elements are combined by making each PE sequentially transfer the partial sums only of the column segments for which it is not responsible. Each segment transfer is made to the PE that is responsible for the corresponding segment in its final result. To perform concurrent transfers, all PEs will have a unique destination PE (i.e., transfer the partial sum for a unique segment) during each of the $r-1$ combining steps. This is done concurrently for all columns, in groups of r PEs each.

Therefore, segment combining requires $r-1$ communication steps, data transfers of size $\lceil \frac{m}{r} \rceil$, and each PE only performs $\lceil \frac{m}{r} \rceil$ additions after each communication. For the cases considered in this study ($m = 6$ and $n = 7$), segment combining generally performed the best for step 1 of the single sweep SVD algorithm, and so it was used. (For larger values of m and n, the other approaches may perform better.) The reason it outperformed the more popular recursive doubling techniques was because a unique final sum was being computed on each PE. In situations where the same final sum was required on each PE with data from the same column (e.g., to compute p, q, and c), full recursive doubling performed better and was used. Thus, all implementations here used segment combining for step 1, and full recursive doubling for steps 2 and 3.

5. SIMD ARCHITECTURE EXPERIMENTS

5.1. Algorithm Implementation Details

This section presents the parallel implementations of the fault-free and post-fault SVD algorithm on an SIMD architecture.[25] SIMD machines consist of a collection of PEs, a central control unit (*CU*), and an interconnection network. The CU broadcasts instructions to all PEs, forming a single instruction stream. Each instruction is performed synchronously on all enabled PEs.

The SIMD machine used in this study was a MasPar MP-1 system[6] with 16,384 PEs located at Purdue University. The MP-1 provides two different high-speed PE interconnection networks, the X-Net and the global router. The *X-Net* connects a PE to its eight nearest neighbors and provides fast communications for PEs in close proximity. The *global router* is a multistage interconnection network that connects groups of 16 PEs and is faster for communications between PEs that are further apart. Data transfers using the X-net between nearest neighbors can achieve 18 GB/s for a 16,384 PE machine, and 1300 MB/s using the global router.

There are four different implementations of the fault-free and post-fault SVD algorithm for the MP-1, based on two different design options: data distribution (1CPP or 2CPP) and interconnection network selection (X-Net or global router). All implementations were written in MPL, a variant of C developed by MasPar for the MP-1. Each implementation also utilized column segmentation for $r \in \{1, 2, 4, 8\}$. Thus, at most $(n \times r \times (n+1)) = 512$ PEs are required.

To take advantage of as much parallelism as possible in the global router implementations, only one PE per global router group was used to reduce contention. The PEs used were spread out as much as possible, to reduce contention at the lower levels of the multistage global router. If $r > 1$, the additional column segments were distributed to PEs within groups with separate connections to the global router.

In contrast, implementations using the X-Net selected a collection of PEs that were all adjacent, to keep inter-PE distances short and inter-PE communications fast. For example, if $r = 1$ and $n = 7$, PEs 0 to 6 were used for one matrix. If $r = 2$ and $n = 7$, PEs 0 to 6 hold segment 0, and PEs 128 to 134 hold segment 1 (there are 128 PEs per row in the MP-1).

For step 1 of Fig. 2, each PE contains the entire J matrix, and only a segment (for $r > 1$) of each column of V, so matrix multiplications are performed as concurrent vector–vector multiplications. This creates an $m \times 1$ vector of partial sums on each PE. These partial sums were then added using segment combining to form the unique solution for each PE.

Step 2 performs one sweep of rotations on the columns of B and V. To do this, all possible combinations of pairs of columns of B must be formed. The same must also be done for V. This makes step 2 the most communication-intensive step, especially for the 1CPP distribution. The 1CPP method requires $n-1$ B column transfers, $n-1$ V column transfers, and $n-1$ scalar transfers to calculate q. The 2CPP approach only requires $n-2$ B column transfers and $n-2$ V column transfers, one after each rotation step. Both approaches also require additional communications for combining column segments, performed here by full recursive doubling because the same sum (and not a different sum for each segment) is required on each PE with data from the same column. The full recursive doubling is done in $\log_2(r)$ steps, in groups of r PEs for all groups concurrently.

Step 3 of the SVD algorithm normalizes the columns of the B matrix to obtain the columns of the U matrix, as well as the singular values, according to Eqs. (2.9) and (2.10). Full recursive doubling is repeated for obtaining the final results.

5.2. Timing Results

Experimental timing results for the 1CPP, 2CPP, X-net, and global router implementations were obtained. Only instructions directly relating to computation or communication in the algorithms were timed. Procedures such as file I/O were not timed to reduce possible disruption by events beyond the control of the programmer, e.g., operating system interrupts.

The timings shown represent the average time to calculate the SVDs of all eight ${}^fJ$ matrices corresponding to one Jacobian matrix. This average is taken over 1000 different randomly generated matrices. Even though the MP-1 is an SIMD machine, meaning it operates synchronously and there should be no variation in timings, averages were still taken because of data conditional execution of statements within the code.

Figure 8 shows a direct comparison between the 1CPP and 2CPP distribution execution times on the MP-1 for the global router fault-free and post-fault SVD calculation. The execution times are grouped in terms of the number of active PEs, which was always set to be a power of two. (In some cases, columns were padded with zeros, because $m = 6$ is not a power of two. This does not affect the final outcome of the singular values.)

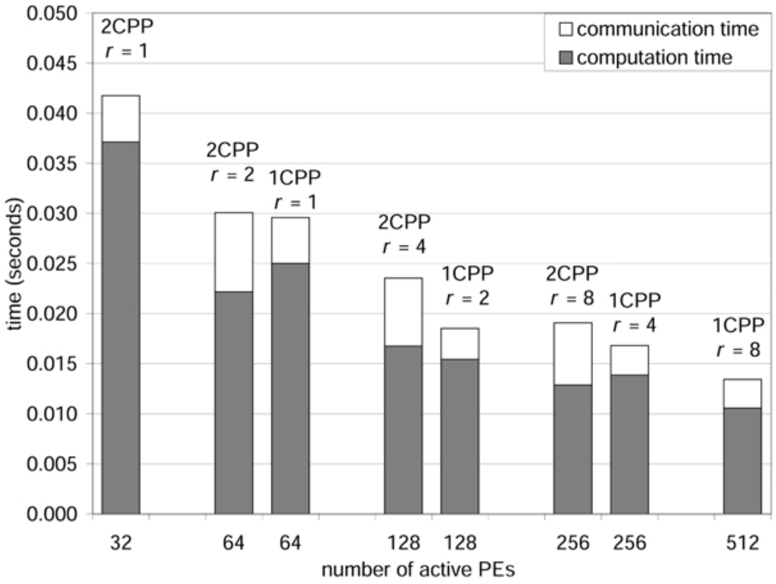

Fig. 8. Comparison between 1CPP and 2CPP average execution times for the MasPar MP-1 global router implementation, in terms of number of active PEs.

From Fig. 8, when comparing cases that have an equal number of active PEs, the 2CPP method has a higher degree of column segmentation and thus more column combining operations. The 2CPP communication pattern also requires more PE enabling and disabling statements than the 1CPP method (e.g., computing new subgroups, and checking for the x or y column). Thus, the communication time for the 2CPP method is greater than the 1CPP method for an equivalent number of PEs.

Examining the communication times for the 1CPP method as r increases, the communications times do decrease slightly in each case. Therefore, column segmentation was beneficial, and the time saved transferring fewer elements between columns was greater than the overhead of performing combining operations within columns. The largest decrease was from 1CPP, $r = 1$ to 1CPP, $r = 2$, i.e., going from no column segmentation to a minimum amount of column segmentation. As $r > 2$, more column combining was required, and communication times improved less and less.

Examining the communication times for the 2CPP method, one notices a heavy initial penalty from column segmentation. The 2CPP $r = 1$ case has the smallest communication time of all four 2CPP cases, but $r = 2$ case has the highest. For $r = 2$, the overhead involved with performing the masking and column combining communications with two column segments during each rotation step was much more costly than the benefit of exchanging a smaller single column segment between rotation steps. As r increases from 2 to 4 to 8, the communication times do decrease slightly. While there is a slight increase in the overhead for column combining masking and communications, it is balanced by the benefit of the smaller column segments in all communications. However, the decrease never gets back down to the $r = 1$ level, where there was no combining overhead at all.

Comparing the communication times between the 1CPP and 2CPP methods, the 2CPP $r = 1$ case is slightly less than the corresponding 1CPP $r = 1$ case. This is as expected, given the fewer number of communications for the 2CPP case with no column segmentation. As r increases, 2CPP has the higher communication times than the corresponding 1CPP case. This is because segment combining and recursive doubling for two columns (1) involves twice as much data than the 1CPP case (two columns versus one column); and (2) occurs more often (several places within Fig. 2) than the exchange of columns (after each rotation step).

Examining just the computation times (without communication times), the 2CPP technique should be faster than the 1CPP method. The 2CPP method is more conducive to the pair-wise operations of the rotation steps because it avoids some of the redundant calculations the 1CPP method must perform. The exception to this observation occurs for 128 PEs because the

Table I. Summary of the Best Results from Each of the Three Architectures

		No. of PEs	r	Computation time (s)	Communication time (s)	Total time (s)
MasPar MP-1 (SIMD)	1CPP	512	8	0.01056	0.00286	0.01342
MasPar MP-1 (SIMD)	2CPP	256	8	0.01285	0.00623	0.01908
IBM SP2 (MIMD)	4CPP	16	1	0.00065	0.00122	0.00187
IBM SP2 (MIMD)	8CPP	8	1	0.00134	0	0.00134
PASM (mixed-mode)	1CPP	16	4	7.701	0.092	7.793
PASM (mixed-mode)	2CPP	8	4	14.071	0.195	14.266

data was padded with zeros so that the number of active PEs was a power of two (simplifying recursive doubling).

For the 1CPP case, there is a decrease in total execution times as a result of increasing the number of PEs. However, execution times improved less per PE as the number of active PEs went from 256 to 512. This implies that the use of column segmentation was beneficial, but provides diminishing returns (per PE) as the columns were segmented into smaller and smaller pieces. A similar trend occurs for the 2CPP case.

A comparison of the X-Net and global router results revealed that the global router implementations achieve the faster execution times. The computation times between the X-Net and global router implementations were nearly equivalent, as one would expect. However, given the communication patterns and matrix sizes of this application, and the ability to select the enabled PEs, the global router implementations provided better performance than the X-Net implementations in each case.

In all cases (global router, X-Net, and various values of r), the differences in computation times between 1CPP and 2CPP were not enough to overcome the differences in communication times, and the 1CPP technique had a faster total execution time. The global router, $r = 8$, 1CPP technique had the fastest total execution time on the MasPar MP-1. In all of the cases examined on the MasPar MP-1, increasing the number of processors improved execution times. Table I summarizes the best results achieved by the MasPar MP-1 (with global router) for comparison with the other architectures. Detailed results for individual cases can be found in Braun.[26]

6. MIMD ARCHITECTURE EXPERIMENTS

6.1. Algorithm Implementation Details

The parallel implementation of the fault-free and post-fault SVD algorithm on an MIMD architecture[25] is described in this section. Each

PE in an MIMD machine stores its own set of instructions and data in its local memory module. This allows for asynchronous, multiple threads of control, because PEs may contain unique sets of instructions.

The IBM SP2 is a scalable distributed memory MIMD parallel supercomputer.(7) The interconnection network in the SP2 is a multistage interconnection network based on the *SP2 High-Performance Switch*.(7) Message passing for this study used a C-based implementation of the *Message Passing Interface* (*MPI*).(27) Simulation results were obtained using only the thin node type of processor,(7) with submachine sizes, $\underline{s}$, of $s \in \{1, 2, 4, 8, 16\}$.

Both the 1CPP and 2CPP methods were implemented on the SP2. The biggest difference between the MP-1 and the SP2 implementations was the use of MPI on the SP2. Another difference between the MP-1 and SP2 implementations was the number of PEs available. The number of thin nodes on the SP2 that was used for the experiments in Eq. (2.16), was fewer than $(n \times r \times (n+1))$, so for most cases, the outer loop of the fault-free and post-fault SVD algorithm (Fig. 3) cannot be performed concurrently with all values of f, as it was on the MP-1.

For the computational sections of the algorithms, the MP-1 and SP2 implementations were very similar. The same computations are performed, they are just performed asynchronously on the SP2. In contrast, the communication sections of the algorithm implementations differ greatly between the MP-1 and SP2. The SP2 performs asynchronous inter-PE communications, which require blocking until the required data has been received.

Because of the limited number of PEs available, and the high MPI overheads observed, two additional techniques were added to the MIMD portion of the study, namely an eight column per PE (*8CPP*) and a four column per PE (*4CPP*) method. The 8CPP method executes entirely on one PE with no inter-PE communications. The 4CPP method executes on two PEs, each holding four columns of B and four columns of V. The 4CPP method performs two exchanges, each containing two columns of B and two columns of V. The 8CPP and 4CPP techniques did not use column segmentation.

6.2. Timing Results

Because of their asynchronous operation, there is usually a large variance in timing information from MIMD machines. The timings recorded, as before, represent the time to calculate the SVDs of all eight ${}^{f}J$ matrices corresponding to one J matrix, taken as the average time over 1000 different matrices.

Comparing the 8CPP, 4CPP, 2CPP, and 1CPP distributions of the fault-free and post-fault SVD algorithm in Fig. 9, the communication times dominated the total execution time of the algorithms on the SP2. This is because of the large overhead associated with MPI communications.(28) When using MPI, the time required for setup and initialization of each communication is relatively large. If only small sets of data are being transferred, the overhead can easily require more time than the actual transfer of data.

Consider the communication times for 1CPP, $r = 1$ and $r = 2$. Notice that the communication time has nearly quadrupled for $r = 2$. This is partially because half as many SVDs are able to be calculated concurrently during each iteration of the for all loop of Fig. 3. For $r = 1$, two SVD are computed concurrently (16 PEs/8 columns per matrix = 2 concurrent SVDs). For $r = 2$, only one SVD is computed at a time. This by itself doubles the number of inter-PE column exchanges needed. Also, when $r = 2$, segment combining and recursive doubling communications must be performed for each column operation. This more than doubles the total number of communications performed. The fact that some of these increased number of communications involve half as much data as the $r = 1$ case is masked by the large overhead of each communication. Thus, the communication time for $r = 2$ is approximately four times that of the $r = 1$ case.

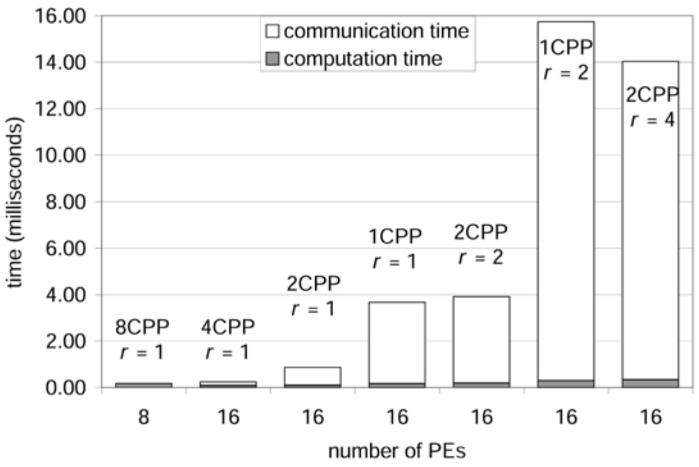

Fig. 9. Comparison among 1CPP, 2CPP, 4CPP, and 8CPP average execution times for the IBM SP2 implementations.

The large overhead also accounts for the change in performance between the 1CPP and 2CPP methods. The combining operations for the 2CPP $r = 2$ case nullify the advantage of fewer column exchanges as compared to the 1CPP $r = 1$ case, and thus 2CPP does slightly worse. For 1CPP $r = 2$, combining operations are now introduced into the 1CPP method, and the combination of penalties from having more communications and some combining operations outweighs the additional penalties of increasing r from two to four for the 2CPP case. The best case on the IBM SP2 was the 8CPP case, which used just one matrix per PE, and only half of the available PEs.

Comparing timing results between the IBM SP2 and the MasPar MP-1 (see Table I), the relative strengths of each machine become apparent. The MP-1 is a well balanced machine with computation and communication instructions requiring about the same amount of time to execute. In contrast, the SP2 has superior computational speed but relatively slow communications when using MPI. In most cases, communication times on the MP-1 using the global router were equal to or less than the corresponding SP2 implementations using MPI. The advantage in computation time goes to the SP2 which defeated the MP-1 in every instance. This can largely be attributed to the SP2 being a newer machine and having better processor technology available at its time of design and construction. In general, the total execution times for the SP2 were less than the MP-1.

7. MIXED-MODE ARCHITECTURE EXPERIMENTS

7.1. Algorithm Implementation Details

The PASM (partitionable SIMD/MIMD) parallel processing system[8, 9] was the third parallel architecture used to implement the single sweep SVD algorithm and the fault-free and post-fault SVD algorithm. PASM, designed at Purdue University, supports *mixed-mode* parallelism, i.e., it can operate in either SIMD or MIMD mode of parallelism, and can switch modes at instruction level granularity with generally negligible overhead. A small-scale 30-processor PASM prototype has been built with 16 PEs in the computational engine. For inter-PE communications, PASM uses a partitionable circuit-switched multistage cube interconnection network. The network can be used in both SIMD and MIMD modes.

PASM is capable of employing hardware-based barrier synchronization in MIMD mode, called Barrier MIMD *BMIMD*. Each PE executes its code independently until it arrives at a synchronization point called a *barrier*. Each PE waits at the barrier until all PEs indicate they have reached it, then they continue execution simultaneously. One use for this is to synchronize inter-PE transfers performed in MIMD mode.

The PASM implementations use a combination of C and ELP programs. ELP is a custom language developed for use on PASM. ELP implements a subset of the C instruction set, and includes custom instructions for inter-PE communications and switching operation modes (SIMD or MIMD).

7.2. Mixed-Mode Analysis

To extend the mixed-mode analysis, matrices of size 4×4, 4×8, and 8×8 were used. This allowed timing data to be recorded while using different numbers of PEs on the 16 PE PASM prototype. Both the 2CPP and 1CPP implementations were executed with matrices of these three sizes. Only results from the fault-free single sweep SVD algorithm (Fig. 2) will be discussed. Unless otherwise stated, experimental timing data being presented represents the average execution times of an algorithm run on 256 different Jacobian matrices of the given size.

The experimental data is normalized to the average execution time of the SVD algorithm when decomposing a 4×4 matrix with a single PE. This is to show the gains from the different parallel implementations examined. The raw execution speeds are not the focus of the mixed-mode analysis. [For completeness, the actual execution times are given in Table I.] Instead, the goal is to look at how the reconfigurability of a mixed-mode architecture like PASM can exploit different properties of the problem to obtain better performance over any other mode of execution. The absolute execution times were very high compared to the SP2 due to the type of processors used in the PASM prototype.

The 2CPP and 1CPP algorithms were performed on the PASM prototype using SIMD, MIMD, BMIMD, and mixed-mode modes of parallelism. To determine the most effective mode mappings, each of the algorithms were divided into several code fragments. The fastest execution mode for each code fragment was then determined.

The SIMD and MIMD modes of parallelism each have several advantages and disadvantages.[29] An advantage of SIMD over MIMD is the ability to overlap execution of instructions on the CU and on the PEs (*CU/PE overlap*). For the SVD implementations, this overlap occurs when the CU performs the overhead associated with loops implementing the equations noted in Fig. 2, while the PEs execute the loop bodies. Another advantage of SIMD is that the implicit synchronization after every instruction broadcast from the CU to the PEs implies that synchronization is not required during communication, as was required for the IBM SP2 and PASM MIMD implementations.

An advantage of MIMD over SIMD is the ability to execute the clauses of data conditional statements without underutilizing PEs, i.e., in SIMD the data conditional "then" and "else" clauses must be broadcast to the PEs serially and some PEs are idle during the execution of a particular clause. Another advantage of MIMD mode over SIMD mode on PASM was for intensive floating point calculations, MIMD mode was faster. Due to the way that the processors in the PASM prototype implemented floating point operations, SIMD mode required extra overhead (as compared to MIMD mode) to synchronize suboperations. A MIMD disadvantage is that sender/receiver synchronization is required before inter-PE communication can take place. On PASM, sending and receiving PEs must be synchronized for every value sent through the network in MIMD mode. In the BMIMD implementations, all operations are executed in MIMD with the exception that a barrier is executed once for every network setting. After the barrier, all required data transfers can be done in SIMD mode, with less overhead than MIMD network transfers.

Mixed-mode implementations incorporate advantages of both the SIMD and MIMD mode implementations while trying to avoid the disadvantages of each. Various mode combinations were considered for the different program fragments of both the 2CPP and 1CPP approaches. The following is an analysis of the implementations that resulted in the smallest execution times for each of SIMD, MIMD, BMIMD, and mixed-mode.

Figure 10 shows how the 1CPP single sweep SVD algorithm was divided into code fragments. The figure also states the fastest mode of parallelism for each 1CPP code fragment, and the reason(s) for choosing that mode of parallelism. These were the modes used in the mixed-mode implementation.

Fragments 1 and 2 constitute step 1 of the SVD algorithm. Fragment 1 is a nested loop calculation of the partial sum of one column of the B matrix, where each of m column elements are determined from n/r matrix elements of J and V. This fragment is implemented in SIMD mode to maximize the advantage of CU/PE overlap. Fragment 2 is a set of transfers in a loop that combines the partial sums of segments of b_i. Fragment 2 is also implemented in SIMD to utilize both CU/PE overlap and implicit network transfer synchronization.

Code fragments 3 through 8 constitute step 2 of the SVD algorithm. Inter-rotation step column segment transfers are handled by code fragment 3. Fragment 3 is performed $n-1$ times during the execution of 1CPP, once for each column transfer called for by the algorithm in Fig. 2. SIMD mode is used to take advantage of both CU/PE overlap and transfer efficiency.

Fragment 4 calculates two partial sum values within a loop executed in SIMD mode. Fragment 5 combines these partial sums via recursive doubling

algorithm step	code fragment	code fragment description	fastest mode	reason
1	1	derive partial sum of column b_i	SIMD	a
	2	combine segments of b_i	SIMD	a, b
2	3	exchange segment of b_i and $\hat{v}_i$ with current partner	SIMD	a, b
	4	derive partial sums of $b_i^T b_i$ and p	SIMD	a
	5	combine $b_i^T b_i$ and p partial sums	SIMD	a, b
	6	exchange $b_i^T b_i$ value with current partners	SIMD	b
	7	calculate q, c, $\sin(\phi)$, and $\cos(\phi)$	MIMD	d
	8	rotate b_i and $\hat{v}_i$	SIMD	a
3	9	derive partial sum of $b_i^T b_i$	SIMD	a
	10	combine $b_i^T b_i$ partial sums	SIMD	a, b
	11	calculate σ_i, and conditional calculation of $\hat{u}_i$	MIMD	c

reason key

a:	CU/PE overlap of loop control with operations
b:	implicit "send" and "receive" synchronization
c:	"then" and "else" code execution not serialized
d:	faster floating point performance

Fig. 10. Fastest mode of parallelism for each code fragment in the 1CPP single sweep SVD algorithm.

transfer operations in a loop. Again, this loop is executed in SIMD to take advantage of both CU/PE overlap and implicit transfer synchronization. After the partial sums are combined, code fragment 6 performs an inter-PE communication so that partner PEs can exchange their value of $b_i^T b_i$. This fragment is done in SIMD mode to exploit the implicit network synchronization.

Code fragment 7 calculates the values q, c, $\sin(\phi)$, and $\cos(\phi)$. This is an in-line block of code requiring no loops, so MIMD mode is used to make use of its faster floating point operations. Fragment 8 performs the rotation operation on two column segments of B and of V. Rotating column segments of B requires m/r iterations of a tight loop, and rotating column segments of V requires n/r loop iterations. These loops are again performed in SIMD mode to take advantage of CU/PE overlap.

Code fragments 9 through 11 constitute step 3 of the SVD algorithm. The calculation of the partial sum for the new $b_i^T b_i$ value occurs in fragment 9 as another tight SIMD loop. Fragment 10 performs transfer operations in a loop to combine the single $b_i^T b_i$ term in each PE in SIMD mode. These transfers are performed in SIMD mode to take advantage of both CU/PE overlap and implicit transfer synchronization. Finally, code fragment 11 finds the square-root of this final value to obtain σ_i. A conditional computation of $\hat{u}_i$ is also performed. If σ is nonzero, the division operation is executed. Otherwise, the corresponding column of U is replaced with zeroes. Fragment 11 is therefore performed in MIMD mode to take advantage of parallel "then" and "else" clause execution.

7.3. Timing Results

Figure 11 shows the execution times of the 1CPP implementation when it is run in different modes of parallelism on the PASM prototype computer. The data shown in the figure are the results of 4×4 matrix SVD. [Results were similar for the other matrix sizes.[30]] Confidence intervals were calculated for the data presented. Average execution times were determined for each data point, each from 256 different random matrix SVDs. The confidence interval provides a 95% probability that the average

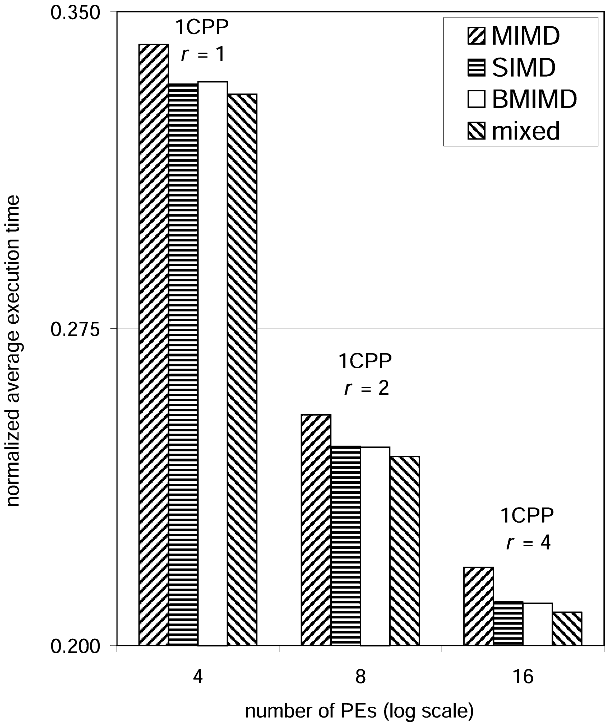

Fig. 11. 1CPP single sweep SVD algorithm execution time comparison for different modes of parallelism on the PASM prototype with a 4×4 matrix.

execution times displayed are within ± 0.0017 on the scale used in the figures. The figure shows that the minimum execution time can be obtained when using mixed-mode parallelism. Results from the 2CPP implementation were similar.

In Fig. 11, it is obvious that the advantage of strictly SIMD operation over MIMD operation increases as the number of PEs increases. It is also obvious that SIMD and BMIMD execution provide similar execution times, meaning that the greatest advantage that SIMD has over MIMD for the SVD algorithm is implicit network transfer synchronization.

The execution times displayed in Fig. 11 also show that the advantage mixed-mode parallelism has over strictly SIMD operation increases as the number of PEs increases. Examining Fig. 10 shows that the mixed-mode code fragments performed in MIMD generally do not operate on column segments, and therefore their performance is generally independent of the number of PEs, i.e., the value of r. Thus, the MIMD code fragment execution times become a larger fraction of the overall execution times as more PEs are used. As the overall execution times decrease, the MIMD advantage of those code fragments becomes more prominent. More detailed analysis and results from the PASM prototype implementations can be found in Ulrey.(30)

In summary, based on the PASM prototype implementation, the following observations can be made for the SVD algorithm. SIMD mode is better than MIMD, and is comparable to BMIMD, for the reasons given. A mixed-mode approach based on the analyses shown in Fig. 10 performed the best, although only 6% better than SIMD/BMIMD and 15% better than MIMD (for 16 PEs).

8. CONCLUSIONS

The system of equations used for the kinematic control of robotic manipulators is frequently represented by a Jacobian matrix. One method for solving this system of equations is based on computing the SVD of the Jacobian matrix. This study uses a technique developed by Maciejewski and Klein(5) that exploits the well-behaved nature of the SVD to calculate the SVD in a single sweep. This technique has been applied to the computation of the SVD of the full, pre-fault Jacobian matrix and the set of single locked-joint, post-fault Jacobian matrices. These procedures can provide a basis for the real-time control of kinematically redundant manipulators and also provide fault tolerance information useful for real-time singularity avoidance and error recovery.

Experiments were conducted for the fault-free and post-fault SVD computations on commercial SIMD and MIMD architectures, the MasPar

MP-1 and IBM SP2. A mixed-mode analysis was also performed on the PASM prototype. For these experiments, data layout, different numbers of PEs, and different modes of parallelism were compared. The fastest overall execution times on the MP-1 were from the 1CPP, global router method. The 8CPP method provided the fastest results on the SP2 because of the high overhead involved with communications. Mixed-mode parallelism gave the best results on the PASM prototype, demonstrating the advantages of mixed-mode processing. Increased column segmentation was an effective method for reducing computation times on the MP-1 and PASM but not on the SP2.

Studies such as this, which compare machines with different network and processor technologies, are useful not only for analysis of the application, but also for profiling each architecture, and pointing out weaknesses which could be addressed in future implementations. All of the methods studied here can be extended to larger analyses, including multiple joint failures, systems of multiple arms, or computation of several fault tolerance measures, all of which would require high levels of parallelism to accomplish in real time.

ACKNOWLEDGMENTS

The authors gratefully acknowledge useful discussions with James Armstrong, Rick Born, Rohit Gupta, Muthucumaru Maheswaran, Robert Palmer, Pierre Pero, Mitchell D. Theys, Lee Wang, Daniel Watson, and Bill Whitson. We also thank the anonymous reviewers for their comments. Preliminary versions of portions of this research appeared in the *Eighth International Parallel Processing Symposium*, and the *Merged 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*.

REFERENCES

1. M. S. Moonen and B. R. L. de Moor (ed.), *SVD and Signal Processing, III: Algorithms, Architectures, and Applications*, Elsevier Science, New York (1995).
2. C. A. Klein and B. E. Blaho, Dexterity Measures for the Design and Control of Kinematically Redundant Manipulators, *Int'l. J. Robotics Res.*, **6**(2):72–83 (1987).
3. A. A. Maciejewski and C. A. Klein, The Singular Value Decomposition: Computation and Applications to Robotics, *The Int'l. J. Robotics Res.*, **8**(6):63–79 (December 1989).
4. A. A. Maciejewski and J. M. Reagin, A Parallel Algorithm and Architecture for the Control of Kinematically Redundant Manipulators, *IEEE Trans. Robotics Automation*, **10**(4):405–414 (August 1994).
5. R. G. Roberts and A. A. Maciejewski, A Local Measure of Fault Tolerance for Kinematically Redundant Manipulators, *IEEE Trans. Robotics Automotion*, **12**(4):543–552 (August 1996).

6. T. Blank, The MasPar MP-1 Architecture, *IEEE Compcon*, pp. 20–24 (February 1990).
7. C. B. Stunkel, D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Grice, P. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stuckel, M. Tsao, and P. R. Varker, The SP2 High-Performance Switch, *IBM Syst. J.*, **34**(2):185–204 (1995).
8. N. Giolmas, D. W. Watson, D. M. Chelberg, P. V. Henstock, Ho Yi, H. J. Siegel, Aspects of Computational Mode and Data Distribution for Parallel Range Image Segmentation, *Parallel Computing*, **25**(5):449–523 (May 1999).
9. M. Tan, J. M. Siegel, and H. J. Siegel, Parallel Implementations of Block-Based Motion Vector Estimation for Video Compression on Four Parallel Processing Systems, *IJPP*, **27**(3):195–225 (June 1999).
10. A. A. Maciejewski and C. A. Klein, Obstacle Avoidance for Kinematically Redundant Manipulators in Dynamically Varying Environments, *Int'l. J. Robotics Res.*, **4**(3):109–117 (1985).
11. R. V. Mayorga, N. Milano, and A. K. C. Wong, A Fast Procedure for Manipulator Inverse Kinematics Computation and Singularities Prevention, *J. Robotic Syst.*, Vol. 9, No. 8 (1992).
12. G. H. Golub and C. F. Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, Maryland (1996).
13. R. P. Brent, F. T. Luk, and C. Van Loan, Computation of the Singular Value Decomposition Using Mesh-Connected Processors, *J. VLSI Computer Syst.*, **1**(3):242–270 (1985).
14. D. Chan, Implementation and Evaluation of a Parallel Singular Value Decomposition Algorithm for Direction of Arrival Analysis, Master's Thesis, Electrical and Computer Engineering Department, Worcester Polytechnic Institute, Massachusetts (May 1995).
15. H. Chuang and L. Chen, Efficient Computation of the Singular Value Decomposition on a Cube Connected SIMD Machine, *Supercomputing*, pp. 276–282 (November 1989).
16. F. T. Luk, A Triangular Processor Array for Computing Singular Values, *Linear Algebra Appl.*, **77**(5):259–273 (1986).
17. D. E. Schimmel and F. T. Luk, A New Systolic Array for the Singular Value Decomposition, in *Advanced Research in VLSI*, C. E. Leiserson (ed.), MIT Press, Cambridge, Massachusetts, pp. 205–217 (1986).
18. M. R. Hestenes, Inversion of Matrices by Biorthogonalization and Related Results, *J. Soc. Industr. Appl. Math.*, **6**(1):51–90 (1958).
19. J. C. Nash, A One-Sided Transformation Method for the Singular Value Decomposition and Algebraic Eigenproblem, *Computer J.*, **18**(1):174–76 (February 1975).
20. J. C. Nash, *Compact Numberical Methods for Computers: Linear Algebra and Function Minimization*, A. Hilger, Bristol, United Kingdom (1979).
21. K. N. Groom, A. A. Maciejewski, and V. Balakrishnan, Real-Time Failure Tolerant Control of Kinematically Redundant Manipulators, *IEEE Trans. Robotics and Automation*, **15**(6):1109–1116 (December 1999).
22. K. S. Fu, R. C. Gonzalez, and C. S. G. Lee, *Robotics: Control, Sensing, Vision, and Intelligence*, McGraw–Hill, New York (1987).
23. D. Hanselman and B. Littlefield, *Mastering MATLAB: A Comprehensive Tutorial and Reference*, Prentice Hall, Upper Saddle River, New Jersey (1996).
24. H. S. Stone, Parallel Computers, *Introduction to Computer Architecture*, H. S. Stone (ed.), Science Research Associates, Inc., Chicago, Illinois (1975).
25. M. J. Flynn, Very High-Speed Computing Systems, *Proc. IEEE*, **54**(12):1901–1909 (December 1966).
26. T. D. Braun, Parallel Algorithms for Singular Value Decomposition as Applied to Failure Tolerant Manipulators, Master's Thesis, School of Electrical and Computer Engineering, Purdue University, West Lafayette, Indiana (December 1997).

27. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*, MIT Press, Cambridge, Massachusetts (1995).
28. Z. Xu and K. Hwang, Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2, *IEEE Parallel and Distributed Technology*, **4**(1):9–23 (1996).
29. H. J. Siegel, J. B. Armstrong, and D. W. Watson, Mapping Computer Vision Related Tasks Onto Reconfigurable Parallel Processing Systems, *Computer*, **25**(2):54–63 (February 1992).
30. R. R. Ulrey, Parallel Algorithms for Singular Value Decomposition and a Design Alternatives Study for a Network Interface Unit for PASM, Master's Thesis, School of Electrical and Computer Engineering, Purdue University, West Lafayette, Indiana (December 1993).