Parallel Algorithms for Singular Value Decomposition

Renard R. Ulrey†, Anthony A. Maciejewski‡, and Howard Jay Siegel‡

†NCR Corporation, 2001 Danfield Ct., Ft. Collins, CO 80525 USA
‡Parallel Processing Laboratory, School of Electrical Engineering
Purdue University, West Lafayette, IN 47907-1285 USA

Abstract

In motion rate control applications, it is faster and easier to solve the equations involved if the singular value decomposition (SVD) of the Jacobian matrix is first determined. A parallel SVD algorithm with minimum execution time is desired. One approach using Givens rotations lends itself to parallelization, reduces the iterative nature of the algorithm, and efficiently handles rectangular matrices. This research focuses on the minimization of the SVD execution time when using this approach. Specific issues addressed include considerations of data mapping, effects of the number of processors used on execution time, impacts of the interconnection network on performance, and trade-offs between modes of parallelism. Results are verified by experimental data collected on the PASM parallel machine prototype.

1: Introduction

Decreasing the execution time of computerized tasks is the focus of a tremendous amount of study. The use of parallel computer systems is one method to help decrease these times. The performance of a parallel system, however, is dependent on the algorithm implementation and the parallel machine characteristics. Performance optimization is therefore complicated, due to the wide variety of algorithm characteristics [7] and the rapidly growing variety of parallel machines that have been built or proposed. Thus, the study of mapping algorithms onto parallel machines is an important research area.

The singular value decomposition (SVD) of matrices has been extensively used in control applications, e.g., during the computational analysis of robotic manipulators [8, 22]. The decomposition aids the computational solution of system equations such as the motion rate control formula $\dot{x} = J\dot{\theta}$, where $\underline{\dot{x}} \in \mathbf{R}^M$ specifies the end effector velocity, $\underline{\dot{\theta}} \in \mathbf{R}^N$ specifies joint velocities, and $\underline{J} \in \mathbf{R}^{M \times N}$ is the Jacobian matrix [21]. For systems with many cooperating manipulators, the value of N can reach into the hundreds, resulting in a severe computational burden for achieving real-time control.

This research was supported in part by the National Science Foundation under grant CDA-9015696, by Sandia National Laboratories under contract 18-4379B, and by Rome Laboratory under contract F30602-94-C-0022.

In general, computation of the SVD of an arbitrary matrix is an iterative procedure, so the number of operations required to calculate it to within acceptable error limits is not known beforehand. The control of many systems, however, is based on equations involving the current Jacobian matrix, which can be regarded as a perturbation of the previous matrix, i.e., $J(t+\Delta t) = J(t) + \Delta J(t)$. It has been demonstrated that for these cases knowledge of the previous state can be used during the computation of the current SVD to decrease execution time [12]. This paper describes and analyzes two SVD algorithm implementations for these cases. Experimental data obtained on the PASM prototype parallel computer [1, 19] is provided that supports the conclusions of the algorithm analyses.

Section 2 provides background information about SVD, Givens rotations, and PASM. Descriptions of the two parallel SVD implementations being analyzed are presented in Section 3. Section 4 demonstrates an analysis approach to determine which implementation has the shorter execution time. The performances of SVD implementations on PASM are evaluated in Section 5.

2: Background information

The SVD of a matrix $J \in \mathbf{R}^{M \times N}$ is defined as the matrix factorization $J = UDV^T$, where $\underline{U} \in \mathbf{R}^{M \times M}$ and $\underline{V} \in \mathbf{R}^{N \times N}$ are orthogonal matrices of the singular vectors, and $\underline{D} \in \mathbf{R}^{M \times N}$ is a nonnegative diagonal matrix. The singular values of J, $\underline{\sigma}_i$, are ordered from largest to smallest along the diagonal of D. It is assumed here that $M \leq N$.

The Golub-Reinsch algorithm [6] is the standard technique for determining the SVD of a matrix. This method, however, has two unattractive aspects. The first is that the algorithm, as it is defined, cannot use knowledge of a previous matrix decomposition. The second is that the technique is relatively serial in nature, making more parallelizable algorithms desirable.

Several parallel SVD algorithms have been implemented for various machine architectures, including those proposed in [3, 4, 10, 11, 16]. These implementations also do not allow their iterative natures to be reduced. Algorithms being studied in this paper are based on a methodology presented in [12], which exclusively uses Givens rotations [6] to orthogonalize matrix columns.

Successive Givens rotations are used to generate the orthogonal matrix V that will result in $JV = B$, where the columns of $\underline{B} \in R^{M \cdot N}$ are orthogonal. A matrix with orthogonal columns can be written as the product of an orthogonal matrix U and a diagonal matrix D (i.e., $B = UD$) by letting the columns of U, u_i, equal normalized columns of B, b_i,

$$u_i = b_i / \|b_i\| \quad (\text{where } \|b_i\| = \sqrt{b_i^T b_i}), \tag{1}$$

and defining the diagonal elements of D to be equal to the norm of the columns of B

$$\sigma_i = \|b_i\|. \tag{2}$$

This results in the SVD of J.

The orthogonal matrix V that will orthogonalize the columns of J is formed as a product of Givens rotations, each of which orthogonalizes two columns. Considering the i-th and k-th columns of an arbitrary matrix A, a single Givens rotation results in new columns, a_i' and a_k', given by

$$a_i' = a_i \cos(\phi) + a_k \sin(\phi) \tag{3}$$

$$a_k' = a_k \cos(\phi) - a_i \sin(\phi). \tag{4}$$

The $\cos(\phi)$ and $\sin(\phi)$ terms necessary to achieve orthogonality are computed using the formulas in [14], which are based on the quantities

$$p = a_i^T a_k,\ q = a_i^T a_i - a_k^T a_k, \text{ and } c = \sqrt{4p^2 + q^2}. \tag{5}$$

Using these quantities, when $q \geq 0$

$$\cos(\phi) = \sqrt{(c+q)/(2c)} \text{ and } \sin(\phi) = p/(c \cdot \cos(\phi)). \tag{6}$$

When $q < 0$,

$$\sin(\phi) = \text{sgn}(p) \cdot \sqrt{(c-q)/(2c)} \text{ and } \cos(\phi) = p/(c \cdot \sin(\phi)), \tag{7}$$

where $\text{sgn}(p)$ equals 1 if $p \geq 0$ and -1 if $p < 0$. Two sets of formulas are given so that ill-conditioned equations resulting from the subtraction of nearly equal numbers can always be avoided.

To orthogonalize each possible pair of columns requires $N(N-1)/2$ rotations, referred to as a <u>sweep</u> [6]. The matrix V can be computed by iteratively forming the product of a set of sweeps and testing for convergence. While the number of sweeps required to orthogonalize the columns of J is not generally known beforehand, it was shown in [12] that by using the V matrix from the SVD of the previous J to find an initial estimate for B,

$$B(t+\Delta t) \simeq J(t+\Delta t) \times V(t), \tag{8}$$

one can obtain a good approximation to the new SVD using a single sweep if $\Delta J(t)$ is small. Therefore, in this work the current V matrix is calculated using $V(t+\Delta t) = V(t) \times \prod_{i=1}^{N-1} \prod_{k=i+1}^{N} G_{ik}$, where $\underline{G}_{ik}$ denotes the Givens rotation to orthogonalize columns i and k. Only a single sweep is performed to update the matrix V.

The PASM (<u>p</u>artitionable <u>S</u>IMD/<u>M</u>IMD) parallel processing system [1, 19] was used to implement these algorithms. PASM, designed at Purdue University, supports <u>mixed-mode</u> 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 <u>PEs</u> (processor/memory pairs) in the computational engine. For inter-PE communications, PASM uses a partitionable circuit-switched multistage cube interconnection network [18], also called an Omega [9]. The network can be used in both SIMD and MIMD modes.

PASM is capable of employing barrier synchronization [5] in MIMD mode, called Barrier MIMD (<u>BMIMD</u>). Each PE executes its code independently until it arrives at a synchronization point called a <u>barrier</u>. Then, each PE waits at the barrier until all PEs indicate they have reached it. One use for this is to synchronize inter-PE transfers performed in MIMD mode.

3: Data mapping

3.1: Overview

Based on the equations in Section 2, Fig. 1 gives an algorithm to calculate V, D, and U using Givens rotations. This algorithm assumes that the SVD of the Jacobian matrix from the previous control sample period has been computed. Thus, for step 1, the previous V matrix is available on the system. It is assumed that the algorithm then converges with a single sweep of rotations in step 2.

Referring to the parallel execution of a Givens rotation by all PEs as a <u>rotation step</u>, $N-1$ rotation steps must be performed on $N/2$ column pairs to form all $N(N-1)/2$ column pairs. With unique column pairs distributed among $N/2$ PEs, inter-PE communication is avoided within each rotation step. After the initial rotation step, however, an inter-PE communication is required before each remaining rotation step. This rotate-transfer-rotate sequence is required both to form all column pairs and to converge the B and V matrices to their single-sweep values. Newly updated columns are being transferred in each communication step.

```
step 1:
calculate initial estimate for B from J and previous V, using (8)
step 2:
for all column pairs (i,k) do                    /* one sweep */
   calculate p, q, and c using (5)
   calculate cos(ϕ) and sin(ϕ) using (6) or (7)
   perform rotation on columns i and k of B, similar to (3) and (4)
   perform rotation on columns i and k of V, similar to (3) and (4)
end for
step 3:
calculate D from B, using (2)
calculate U from B and D, using (1)
```

Fig. 1: High-level algorithm for finding SVD using Givens rotations.

As presented in Section 2, the calculations involved in this algorithm are straightforward. Of greater interest are ways to effectively map matrix elements to particular parallel machines, and the types of inter-PE communication these mappings dictate. Various implementations of column transfer operations have been devised, including those in [3, 4, 16]. Each of these methods map a unique column pair to each of N/2 PEs. The availability of a multistage cube network on PASM allows matrix data to be distributed across more PEs than allowed by implementations in [3, 4, 16], and thus increases the number of PEs that can perform useful work while still performing all necessary inter-PE communications in single transfer steps.

Two different methods for mapping matrices to PEs are presented. These implementations assume that $M=2^m \le N=2^n$ for the Jacobian matrix $J \in R^{M \cdot N}$. If the matrix does not have these dimensions, it can always be padded with zeroes. This section explains the data layout and communication patterns for each method. The algorithmic details for each are in Section 4.

3.2: Mappings being analyzed

A two columns per PE (2CPP) data mapping is the first to be considered. Assume that N/2 PEs are used, numbered from 0 to (N/2) − 1. Let S be the number of PEs in a communicating subgroup (S a power of two), and let i be the address of a PE that is transferring a column through the network. The 2CPP method uses the interconnection function $\underline{\text{Shift}_j}(i, S) = S\lfloor i/S \rfloor + ((i+j) \bmod S)$, where j is the Shift length, to determine the address of the destination PE. This function allows the destination PE address to remain within a current communicating subgroup of size S. The resulting communication patterns are conflict-free transfers on a multistage cube network [9, 18].

Let each PE contain columns x and y. All possible column pairs are formed by iteratively performing a two-step process. To begin, S = N/2 so that all PEs being used are in a communicating subgroup. In the first step, all y columns are shifted to all other PEs in the subgroup by applying the function $\text{Shift}_1(i, S)$ a total of S − 1 times. In the second step, the subgroup is split in two by exchanging x and y columns between subgroup halves using $\text{Shift}_{S/2}(i, S)$ and reducing S by a factor of two for the next iteration.

A model of the inter-rotation transfers for this algorithm is shown for N=8 in Fig. 2. Each row of blocks in the figure represents a rotation step where calculations are being performed in each of 8/2 PEs. Number pairs in the blocks denote the columns being updated/rotated in each PE. Arrows illustrate the inter-rotation column transfer steps. Beside each transfer step, the communication function used to achieve the interconnection pattern is specified, as well as the columns being exchanged (x is the left column number in each box, y is the right).

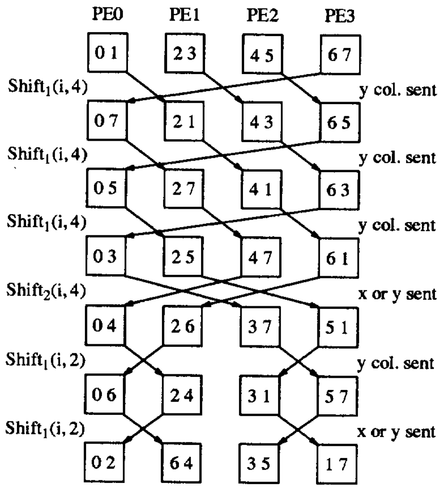

Fig. 2: Column transfer model for 2CPP mapping.

A second implementation was developed that allows all three algorithm steps to be implemented with a one column per PE (1CPP) distribution. Using the capabilities of the multistage cube network, a communication pattern was derived that forms all possible column pairs using N − 1 column transfers. Assume N PEs are used, numbered 0 to N − 1. PE i always contains the most recent version of column i. In the k-th rotation step, $1 \le k < N$, PE i exchanges data with PE $i \oplus k$, where $\oplus$ is the bit-wise exclusive-or. These transfers are conflict-free on the multistage cube [2].

Operations in step 2 require data from both columns of the pair being rotated. Therefore, the 1CPP mapping requires column transfers within each rotation step (intra-rotation transfers) rather than between rotation steps (inter-rotation transfers). A performance trade-off is immediately apparent with respect to the 2CPP method. Steps 1 and 3 can execute with half as many operations using the 1CPP mapping, but step 2 requires one additional column transfer to complete a full sweep. Later sections compare both the expected performance and observed performance between the two methods.

A model of the intra-rotation transfers for this implementation is shown in Fig. 3 for N = 8. Having the columns sequentially ordered after the decomposition may be an advantage for post-SVD operations.

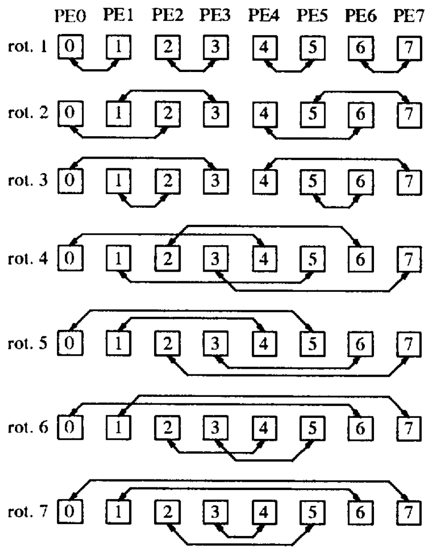

Fig. 3: Column transfer model for 1CPP mapping.

With the 2CPP and 1CPP column distribution models now formed, it is a goal of this study to further utilize parallelism in the SVD algorithm to possibly decrease execution times. The approach taken divides each column of the B and V matrices into $\underline{R} = 2^r$ segments. The total number of PEs that are used increases by a factor of R. For this study, $R \leq M \leq N$.

In the 2CPP mapping, because RN/2 total PEs are being used, $r + n - 1$ PE address bits can be used to fully define the column segment distribution. To map column segments onto PASM, PEs whose addresses agree in the $n - 1$ most significant bits contain different segments of the same column, and PEs with the same r least significant bits have corresponding segments of different columns. Similarly, for the 1CPP mapping, $r + n$ address bits define the column segment distribution among the RN PEs. PEs with the same n most significant bits contain segments of the same column, and PEs with the same r least significant bits have the same segment number.

These segment mappings allow the system network to still perform the column transfer communications as explained for both the 1CPP and 2CPP methods. These communications will occur between PEs that have the same segment number, i.e., agree in a given set of address bits. All PEs can also perform simultaneous communications between PEs containing different segments of the same column as a conflict-free transfer. The addresses of these PEs will agree in a different set of bits. This is due to the partitioning properties of the multistage cube [18].

Communication patterns between PEs that have different segments of the same column have not been discussed. The patterns that provide the fastest algorithm execution were found to be dependent on both the column mapping (1CPP or 2CPP) and the current operation being performed. The communication patterns providing the best performance are detailed in [20].

4: Performance analysis

4.1: Analysis overview

There are three goals of this analysis. The first is to demonstrate some considerations when examining algorithm performance. The second is to see whether a speedup of the SVD algorithm can always be expected when more PEs are used (this is not always the case, e.g., [15]). The third is to determine the conditions when one of the 1CPP and 2CPP implementations performs better.

An operation count analysis for the SVD implementations is the first step toward predicting the better algorithm mapping. The two main components of the SVD algorithm are considered to be computation and inter-PE communication. The number of floating-point operations (<u>FLOPs</u>) performed by each PE will be used as the measure of the amount of computation for this analysis.

In general, the time to perform FLOPs on a machine will depend on the operation to be performed, and possibly on the operands. For this analysis, it is assumed that all FLOPs and their associated address calculations take the same constant amount of time. It was shown in [15] that using an experimentally-derived average time as the execution time of each FLOP can provide good results.

The time it takes to set up a valid network configuration in SIMD mode on the PASM prototype is close to that to perform a floating-point (<u>FP</u>) data transfer. For this reason, and because the inter-PE transfers performed throughout the SVD algorithm involve different numbers of data items, the time spent performing communications in this analysis is represented by the total number of single data transfers (<u>DTs</u>) performed by each PE. Experimental results presented in Section 5 will show that this is a good approximation.

4.2: Operation counts

Various methods were derived to perform each of the three steps of the SVD algorithm with both the 2CPP and 1CPP approaches. A comparison of the operation counts for the different methods is detailed in [20]. The methods with the smallest complexities were implemented.

Because of the distribution of columns and column segments among PEs, many operations require the combining of the partial sums of calculations performed by single PEs. In most cases, some variation of recursive doubling (described in [17]) allowed execution with the fewest FP and DT operations. Other methods were also found to reduce the number of operations. The similarity of (6) and (7) is exploited so that both $\cos(\phi)$ and $\sin(\phi)$ are calculated using only 6 non-data-dependent FLOPs, regardless of the mode of parallelism being used. For the 1CPP approach, a method was developed for both SIMD and MIMD modes to perform column rotations on all PEs simultaneously, where half of the PEs rotate their own columns according to (3) and the other half according to (4). Again, the similarity of the equations is exploited. These methods are detailed in [20].

The complete complexity equations for both approaches are shown in Table 1. Because of the method chosen to perform the 2CPP approach, its total operation count has a special case when R = 1. For comparison purposes, the total number of FLOPs needed to perform the entire SVD algorithm using a single processor is also shown in the table.

4.3: Relation of no. of PEs to operation count

To find the number of PEs to use so that the fewest number of operations are performed, the derivative with respect to R of the equations shown in Table 1 are found and set to zero. Doing this for the FLOP count of the 2CPP implementation and rearranging the resulting equation reveals that the minimum number of FLOPs are performed when $R = (2N + (16NM - 8M - 2N)/(3N - 2))$. Similarly, for the 1CPP implementation, $R = (1.5N + (9NM - 5M - 1.5N)/(2N - 1))$. Recall that as R increases the number of PEs increases, and that for this study it is assumed $1 \le R \le M \le N$. Because $R > M$ in these two equations, the number of FLOPs used in each implementation continues to decrease as R (and the number of PEs) increases, up to the maximum allowed when R = M.

Setting the derivative of the DT count of the 2CPP approach to zero results in the mathematically optimal value of $R = ((N^2 - 2N + NM - 4M)/(3N - 2))$. In this equation, R may be less than M, depending on the values of N and M. Setting the derivative of the DT count of the 1CPP approach to zero results in the mathematically optimal value of $R = ((N^2 - N + NM - 2M)/(2N - 1))$. Again, R may be less than M, depending on the values of N and M. An examination of this equation, however, provides interesting results. Letting M = N, the equation reduces to $R = N - (2N/(2N - 1))$, so the optimal value of R will be between $N - 2$ and $N - 1$. Therefore, when using the 1CPP algorithm with M = N, the number of DTs will decrease as R increases from 1 to $M - 2$. Also, if M in the original equation is reduced to less than $N \ge 4$ by some power of two, the mathematically optimal value of R is larger than its assumed maximum value of $M \le N/2$, and the minimum number of DTs is always reached when the maximum number of PEs are used.

Floating-point Operation Count		
implementation	total	condition
2CPP (RN/2 PEs)	$(1/R)(6N^2 - 6N + 16NM - 8M) + r(3N - 2) + (9N - 10)$	$1 < R \le M$
	$6N^2 + 3N + 16NM - 8M - 9$	$R = 1$
1CPP (RN PEs)	$(1/R)(3N^2 - 3N + 9NM - 5M) + r(2N - 1) + (10N - 10)$	$1 \le R \le M$
one PE	$3N^3 + (3/2)N^2 + 8N^2M - 5NM - (9/2)N + M^2$	
Network Data Transfer Count		
implementation	total	condition
2CPP (RN/2 PEs)	$(1/R)(N^2 - 2N + NM - 4M) + r(3N - 2) + (N + 2M - 1)$	$1 < R \le M$
	$N^2 - N + NM - 2M - 2$	$R = 1$
1CPP (RN PEs)	$(1/R)(N^2 - N + NM - 2M) + r(2N - 1) + (N + M - 1)$	$1 \le R \le M$

Table 1: SVD algorithm operation count totals.

The possibility that the number of DTs performed by an algorithm may increase as the number of PEs increases means that there could be a case when the total algorithm execution time increases when more PEs are used. A method is presented in the next subsection for determining whether this is true for a given system and problem size.

4.4: Performance prediction

A method is adapted from [15] to predict the number of PEs to use that will minimize the execution time for the SVD algorithm. This method gives relative weights to the FP and DT operations by the determination of a communication ratio (CR). This ratio is used with the complexity equations in Table 1 to predict only whether performance improves as more PEs are used. Because the numbers of FP and DT operations do not account for the total execution time, machine-dependent data was collected to use for the prediction.

The CR is calculated in terms of average expected time to perform a DT over the average expected time to perform a FLOP (including memory access and array address calculation times). The units of measure for the CR are ((secs./DT)/(secs./FLOP)) = FLOPs/DT. Various methods can be used to determine the CR. The one chosen executes one implementation of the SVD algorithm on a small matrix, using the minimum number of PEs that the implementation allows. The 1CPP algorithm was arbitrarily selected to measure the CR, with four PEs being used to decompose a random 4×4 matrix. Although the PASM prototype can operate in different modes of parallelism, SIMD mode is used throughout this analysis for consistency. Hardware timers are used to measure the execution times of the operations being considered. Because the PASM prototype currently performs all FP calculations in software and has a relatively fast inter-PE communication network, its CR measured 0.119. It is assumed for this analysis that the CR does not vary with the number of PEs used.

Using the CR, the predicted performance (PP) of a machine running an SVD implementation is approximated by $PP = (\text{no. of FLOPs}) + CR \cdot (\text{no. of DTs})$, and is a function of both matrix size and the number of PEs. With this definition, PP will have units of number of FLOPs. Because the PP equations for the 2CPP and 1CPP approaches (PP_{2CPP} and PP_{1CPP}) do not consider many overhead operations, they do not provide absolute execution times, but they are reasonable estimates of relative execution times as R, N, and M are varied. Therefore, they can be analyzed to determine the number of PEs that will provide minimum execution time on a particular machine.

4.5: Implementation comparison

The operation counts of the 2CPP and 1CPP approaches are now compared. One comparison covers when the number of PEs equals the minimum common number that the two implementations can use (N PEs). A second comparison is for when the maximum common number of PEs are used (NM/2 PEs). These two cases are focused on because various numbers of PEs can be used, depending on the values N, M, and R. The third case directly compares PP_{2CPP} and PP_{1CPP}.

To compare the two implementations with N PEs, replacements are made for R and r in the equations of Table 1 which correspond to using N PEs with either approach. The 2CPP approach requires both fewer FLOPs and fewer DTs under the constraints that $M \geq 2$ and $N \geq 4$ (details in [20]). Because these constraints are met for all values of N and M of interest, the 2CPP implementation is expected to be the fastest (neglecting differences in overhead between the two approaches) when the minimum common number of PEs are used.

To compare the two approaches using NM/2 PEs, the same method is followed, with different values replacing R and r in the equations of Table 1. Analysis in [20] shows that the 1CPP implementation uses fewer FLOPs when NM/2 PEs are used, under the constraint that $M > 2$, which is true for all values of M of interest. It is also shown that the 1CPP implementation uses fewer DTs under the constraint $(M(N-1) \cdot (m+1) + M^2) > N^2$. This inequality is not true for all values of N and M, but it can easily be shown to be true when $M \leq N \leq M(m+1)$, which covers many cases. Thus, for this range of N, the 1CPP implementation is expected to be the fastest when the maximum common number of PEs are used. For N outside this range, the PP of the two implementations can be compared, taking into account the CR of the system.

PP_{2CPP} and PP_{1CPP} are directly compared for three matrix sizes of interest and a $CR = 0.119$ in Fig. 4. The figure shows that as the number of PEs used increases from N to NM/2, the execution times of both methods are expected to decrease, and the fastest implementation is predicted to change from the 2CPP approach to the 1CPP approach. It also shows that the 1CPP implementation with NM PEs is expected to provide the overall minimum execution time.

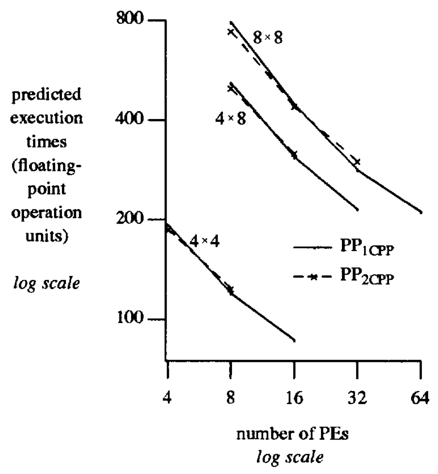

Fig. 4: Comparison of PP_{2CPP} and PP_{1CPP}; $CR = 0.119$.

5: Performance evaluation

5.1: Experimental algorithm performance

Matrices of size 4×4, 4×8, and 8×8 allow timing data to be recorded while using different numbers of PEs on the 16-PE prototype. Both the 2CPP and 1CPP implementations were executed on the PASM computer with matrices of these sizes. Jacobian matrix data consisted of randomly generated FP values within the range (−5, + 5). Algorithms were coded using a combination of a C language compiler, AWK scripts (for pre- and post-processing), and library routines for data-conditionals, network transfers, and data transfers from the control unit (CU) to the PEs. Matrix and column elements were stored in arrays. Values for M, N, and R were left as variables that could be updated before each execution so that several data points could be obtained easily. But, because M, N, and R were variables, all column and column segment operations involved loops that could not be unrolled.

The execution times were recorded for both the 2CPP and 1CPP implementations executed in SIMD mode on the PASM prototype. Matrices of each of the three dimensions specified were decomposed using both implementations, with all allowable numbers of PEs between one and 16, inclusive. The recorded data is plotted in [20].

A comparison of the 2CPP and 1CPP implementation execution times is illustrated in Fig. 5. The experimental timing data 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. For the 4×4 matrix case, it is apparent that the fastest implementation switches from the 2CPP approach to the 1CPP approach when going from four to eight PEs. Also, the 1CPP approach can use 16 PEs when working with a 4×4 matrix, whereas the 2CPP approach cannot. The data obtained for 4×4 matrices is as expected. Similarly, for the 4×8 matrix case, the fastest implementation switches from the 2CPP to the 1CPP approach when going from eight to 16 PEs. When working with an 8×8 matrix, the 1CPP implementation execution time approaches that of the 2CPP implementation when going from eight PEs to 16 PEs. All of these observations of Fig. 5 match exactly the comparison of the PPs shown in Fig. 4.

It was desired to determine how algorithm execution times are affected by the number of PEs when the CR is much higher than 0.119, as would be expected on a commercially available machine with FP coprocessors or digital signal processors. For this purpose, the 2CPP and 1CPP programs were changed to operate on integer data (the square-root FLOP, which is comparable to a FP division with an MC68881 FP coprocessor [13], was replaced by a single integer division). This was done for experimental timing studies only; FP operations are needed to get the desired accuracy for this application. The CR for this new code was determined to be 1.205. The PP for the two algorithms with this CR are analyzed in [20]. It is shown that for 4×4, 4×8, and 8×8 matrices, execution times are still expected to decrease as the number of PEs

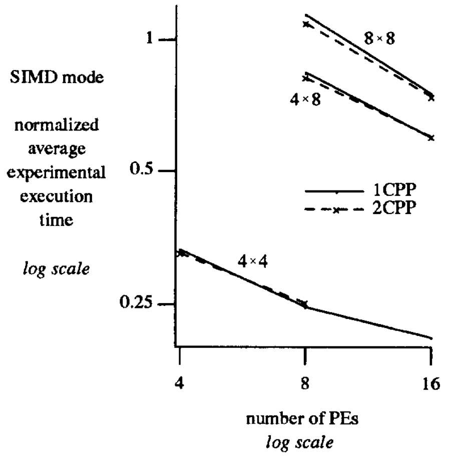

Fig. 5: Experimental performance of 2CPP vs. 1CPP.

increases. These predictions were verified by experimental execution times on the PASM machine [20].

5.2: Modes of parallelism

The 2CPP and 1CPP algorithms were performed on the PASM prototype using SIMD, MIMD, BMIMD, and mixed-mode parallelism. To determine the most effective mode mappings, both 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 [17]. An advantage of SIMD is the ability to utilize CU/PE overlap. For the SVD implementations, this overlap occurs when the CU performs the overhead associated with loops while the PEs execute the loop bodies. Another advantage of SIMD is that the implicit synchronization after every instruction broadcast to the PEs implies that explicit synchronization is not required during communication. A SIMD disadvantage is that data conditional "then" and "else" clauses must both be broadcast to the PEs.

An advantage of MIMD is the ability to execute the clauses of data conditional statements without underutilizing PEs. A block of instructions whose execution times are data-dependent will complete faster [17]. A MIMD disadvantage is that explicit 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 made as if the PEs were 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 shortest execution times of each method.

Table 2 shows how the 2CPP algorithm was divided into code fragments. Fragment 1 is a nested loop calculation of partial sums of two columns of the B matrix, where each of M column elements is 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 and b_j. Fragment 2 is also implemented in SIMD to utilize both CU/PE overlap and implicit network transfer synchronization.

alg. step	code frag.	code fragment description
1	1	derive partial sums of columns b_i and b_j
	2	combine segments of b_i and b_j
2	3	determine columns to transfer in $\text{Shifts}_{S/2}(i, S)$ op.
	4	transfer column segments via $\text{Shifts}_{S/2}(i, S)$ op.
	5	transfer column segments via $\text{Shift}_1(i, S)$ op.
	6	reorder left/right columns by their number
	7	derive partial sums of $b_i^T b_i$, $b_j^T b_j$, and p
	8	combine $b_i^T b_i$, $b_j^T b_j$, and p partial sums
	9	calculate q, c, $\sin(\phi)$, and $\cos(\phi)$
	10	rotate b_i, b_j, v_i, and v_j
3	11	derive partial sums of $b_i^T b_i$ and $b_j^T b_j$
	12	calculate σ_i and σ_j, $R = 1$
	13	determine $b^T b$ value to combine, $R \neq 1$
	14	combine $b^T b$ term, and partners exchange σ, $R \neq 1$
	15	conditional calculation of u_i and u_j

Table 2: Code fragmentation of 2CPP implementation for mixed-mode parallelism.

Inter-rotation column segment transfers are handled by code fragments 3 through 6. Fragments 3 and 4 are performed $n - 1$ times during the execution of 2CPP, once for each $\text{Shifts}_{S/2}(i, S)$ operation. Fragment 3 performs a short if-then-else conditional in MIMD mode to determine the column segments to be transferred. In fragment 4, a column segment of B and V is transferred by each PE. The matrix element transfers are performed in two loops. SIMD mode is used to take advantage of both CU/PE overlap and transfer efficiency. Code fragment 5 performs the $\text{Shift}_1(i, S)$ operation $N - n - 1$ times throughout execution of 2CPP. These transfers are again executed in SIMD mode for the advantages of CU/PE overlap and transfer synchronization. Fragment 6 is performed after each inter-rotation Shift (i.e., $N - 2$ times). It is an if-then-else operation executed in MIMD mode that reassigns i/j (left/right) column order.

Code fragments 7 through 10 are performed $N - 1$ times during the execution of the 2CPP algorithm; once for each rotation. Fragment 7 calculates three partial sum values within a loop executed in SIMD mode. Fragment 8 combines these partial sums via recursive doubling transfers that get performed r times in a loop. Again, this loop is executed in SIMD to take advantage of both CU/PE overlap and implicit transfer synchronization. Code fragment 9 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 because the instructions' execution times are data-dependent. Fragment 10 performs the rotation 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.

Calculation of partial sums of the values $b_i^T b_i$ and $b_j^T b_j$ occurs in fragment 11 as another tight SIMD loop. Fragment 12 is performed only in the special case when $R = 1$. In this case, the values $b_i^T b_i$ and $b_j^T b_j$ found in fragment 11 are final values, not partial sums, and two square-root FLOPs will yield σ_i and σ_j. Fragment 12 is performed in MIMD mode because instruction execution times are data-dependent. When $R \neq 1$, fragments 13 and 14 are performed to find the final σ_i and σ_j values. Fragment 13 is a short if-then-else operation performed in MIMD that determines the single $b^T b$ value a given PE will be calculating (i.e., for column i or for column j). Fragment 14 performs transfers in a loop to combine the single $b^T b$ term in each PE, then finds the square-root of this final value to obtain σ, and exchanges σ with its partner PE (so both have the final σ_i and σ_j values). These transfers are performed in SIMD mode to take advantage of both CU/PE overlap and implicit transfer synchronization.

The final code fragment, 15, calculates column segments of U by dividing elements of B by the corresponding σ value. If σ is nonzero, the division is executed. If σ is zero, the corresponding column of U is replaced with zeroes. Fragment 15 is therefore performed in MIMD mode to take advantage of parallel "then" and "else" clause execution.

Fig. 6 shows the execution times of the 2CPP implementation when run in different modes of parallelism on PASM. The data shown in the figure are the results of 4×4 matrix SVD. The minimum execution time was obtained using mixed-mode parallelism.

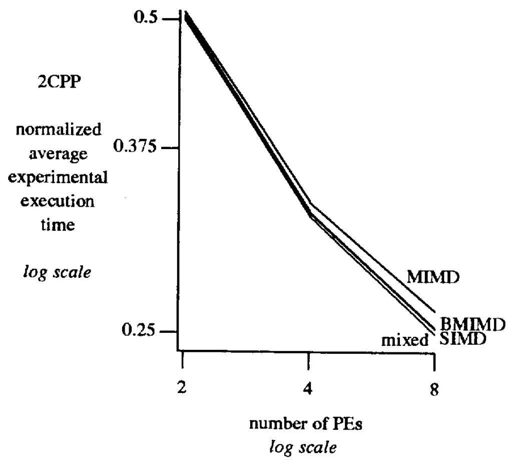

Fig. 6: 2CPP algorithm execution time comparison for different modes of parallelism; 4×4 matrix.

From the figure, it is obvious that the advantage of strictly SIMD mode over MIMD 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. Using the 2CPP DT count equation in Table 1, it can be determined that for a 4×4 matrix, the number of DTs increases as the number of PEs used increases. This means that network transfers become a larger portion of the operations performed, and that the SIMD advantage in transfer times becomes a greater asset.

The execution times displayed in Fig. 6 also show that the advantage mixed-mode parallelism has over strictly SIMD increases as the number of PEs increases. It is apparent from the code fragment analysis that the 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.

The 1CPP algorithm was also fragmented to determine the best combination of modes of parallelism for fastest mixed-mode execution (details in [20]). The 1CPP algorithm has fewer code fragments than the 2CPP approach, and each is analogous to one already presented in the 2CPP mixed-mode analysis. The observations made between the different modes of parallelism with the 2CPP approach held with the 1CPP approach.

6: Conclusions

Several methods for performing SVDs using column transformations have been previously developed. Many of these use rotation operations in an iterative construct to perform the decomposition. Those methods map a unique pair from N columns to each of N/2 PEs, and implement inter-PE communication patterns designed to accommodate their systems' interconnection network. This study presents a similar method, 2CPP, which utilizes the capabilities of a multistage cube network. Another method, 1CPP, was also developed, which maps one matrix column to each of N PEs. The method introduced here for dividing each matrix column into R segments provides the greatest impact on performance. It allows the use of R times the number of PEs previously used, by utilizing more of the inherent parallelism of the SVD algorithms. The approach works effectively with both the 2CPP and 1CPP mappings due to both the methods of data distribution and the capabilities of the multistage cube network.

The methods derived to implement one sweep of rotations (step 2 of the SVD algorithm) can also be applied to other SVD algorithms that iteratively perform multiple sweeps of column rotations. These algorithms would be useful when greater accuracy is needed in the decomposition, or when successive matrices being decomposed cannot be considered as small perturbations of previous matrices. Using more PEs by distributing column segments among PEs may decrease the execution times of these algorithms as well. The performance prediction method presented in Section 4 can be used for this determination.

The analysis presented in Section 4 and supported by experimental data in Section 5 provides the following results. First, the PP analysis presented in Section 4 can be used to determine the number of PEs to use in a system to achieve the minimum execution time of either the 2CPP or 1CPP implementation. Second, the execution times of both implementations depends on the size of the matrix being decomposed, the number of PEs being used, and the CR of the system executing the algorithm. Third, when increasing the number of PEs being used from N to NM/2, the fastest implementation generally changes from the 2CPP approach to the 1CPP approach.

Experimental data presented in Section 5 demonstrates that the mode of parallelism used can have an affect on the execution time of an algorithm. The results obtained show that an SIMD implementation of either the 2CPP or 1CPP SVD approach performs better than an MIMD implementation regardless of the number of PEs used. By using barriers to reduce the synchronization overhead involved in MIMD mode network transfers, the BMIMD implementations outperformed the MIMD implementations. Finally, a mixed-mode implementation can outperform SIMD, MIMD, and BMIMD implementations by using the advantages of each mode on different program fragments.

Acknowledgments: The authors gratefully acknowledge useful discussions with J. Armstrong, R. Born, R. Gupta, R. Palmer, P. Pero, L. Wang, and D. Watson.

References

[1] J. B. Armstrong, D. W. Watson, H. J. Siegel, "Software issues for the PASM parallel processing system," in *Software for Parallel Computation*, J. S. Kowalik, L. Grandinetti, eds., Springer-Verlag, Berlin, Germany, 1993, pp. 134-148.

[2] K. E. Batcher, "The multidimensional access memory in STARAN," *IEEE Trans. on Computers*, C-26(2), Feb. 1977, pp. 174-177.

[3] R. P. Brent, F. T. Luk, C. Van Loan, "Computation of the singular value decomposition using mesh-connected processors," *J. VLSI and Computer Systems*, 1(3), 1985, pp. 242-270.

[4] H. Chuang, L. Chen, "Efficient computation of the singular value decomposition on cube connected SIMD machine," *Supercomputing '89*, Nov. 1989, pp. 276-282.

[5] H. G. Dietz, T. Schwederski, M. T. O'Keefe, A. Zaafrani, "Static synchronization beyond VLIW," *Supercomputing '89*, Nov. 1989, pp. 416-425.

[6] G. H. Golub, C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD, 1983.

[7] L. H. Jamieson, "Characterizing parallel algorithms," in *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, R. J. Douglass, eds., MIT Press, Cambridge, MA, 1987, pp. 65-100.

[8] C. A. Klein, C. H. Huang, "Review of pseudoinverse control for use with kinematically redundant manipulators," *IEEE Trans. on Systems, Man, and Cybernetics*, 13(2), 1983, pp. 245-250.

[9] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. on Computers*, C-24(12), Dec. 1975, pp. 1145-1155.

[10] F. T. Luk, "Computing the singular-value decomposition on the ILLIAC IV," *ACM Trans. on Mathematical Software*, 6(4), Dec. 1980, pp. 524-539.

[11] F. T. Luk, "A triangular processor array for computing singular values," *Linear Algebra and its Applications*, 77(5), 1986, pp. 259-273.

[12] A. A. Maciejewski, C. A. Klein, "The singular value decomposition: Computation and applications to robotics," *Int'l J. Robotics Research*, 8(6), Dec. 1989, pp. 63-79.

[13] Motorola, Inc., *MC68881/MC68882 Floating-Point Coprocessor User's Manual*, MC68881UM/AD, Motorola, Inc., 1987.

[14] J. C. Nash, *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*, John Wiley & Sons, NY, 1979.

[15] G. Saghi, H. J. Siegel, J. L. Gray, "Predicting performance and selecting modes of parallelism: A case study using cyclic reduction on three parallel machines," *J. Parallel and Distributed Computing*, 19(3), Nov. 1993, pp. 219-233.

[16] D. E. Schimmel, F. T. Luk, "A new systolic array for the singular value decomposition," in *Advanced Research in VLSI*, C. E. Leiserson, ed., MIT Press, Cambridge, MA, 1986, pp. 205-217.

[17] H. J. Siegel, J. B. Armstrong, D. W. Watson, "Mapping computer vision related tasks onto reconfigurable parallel processing systems," *Computer*, 25(2), Feb. 1992, pp. 54-63.

[18] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies, Second Edition*, McGraw-Hill, New York, NY, 1990.

[19] H. J. Siegel, T. Schwederski, J. T. Kuehn, N. J. Davis IV, "An overview of the PASM parallel processing system," in *Computer Architecture*, D. D. Gajski, V. M. Milutinovic, H. J. Siegel, B. P. Furht, eds., IEEE Computer Society Press, Washington, DC, 1987, pp. 387-407.

[20] R. R. Ulrey, A. A. Maciejewski, H. J. Siegel, *Parallel Approaches for Singular Value Decomposition on Systems with a Multistage Network*, Tech. Rep. in preparation, EE School, Purdue.

[21] D. E. Whitney, "Resolved motion rate control of manipulators and human prostheses," *IEEE Trans. on Man-Machine Systems*, 10(2), 1969, pp. 47-53.

[22] T. Yoshikawa, "Manipulability of robotic mechanisms," *Int'l J. Robotics Research*, 4(2), 1985, pp. 3-9.