ELSEVIER

Parallel Computing 20 (1994) 1161–1173

PARALLEL COMPUTING

A parallel algorithm for computing the polar decomposition

Nicholas J. Higham *, Pythagoras Papadimitriou †

Department of Mathematics, University of Manchester, Manchester, M13 9PL, UK

Received 11 June 1993

Abstract

The polar decomposition $A = UH$ of a rectangular matrix A, where U is unitary and H is Hermitian positive semidefinite, is an important tool in various applications, including aerospace computations, factor analysis and signal processing. We consider a pth order iteration for computing U that involves p independent matrix inversions per step and which is hence very amenable to parallel computation. We show that scaling the iterates speeds convergence of the iteration but makes the iteration only conditionally stable, with the backward error typically $\kappa_2(A)$ times bigger than the unit roundoff. In our implementation of the iteration on the Kendall Square Research KSR1 virtual shared memory MIMD computer we take p to be the number of processors ($p \leq 16$ in our experiments). Our code is found to be significantly faster than two existing techniques for computing the polar decomposition: one a Newton iteration, the other based on the singular value decomposition.

Key words: Polar decomposition; Singular value decomposition; Numerical stability; LAPACK; Level 3 BLAS; Kendall Square Research KSR1 computer

1. Introduction

Any matrix $A \in \mathbf{C}^{m\times n}$ $(m \geq n)$ has a polar decomposition $A = UH$, where $U \in \mathbf{C}^{m\times n}$ has orthonormal columns and $H \in \mathbf{C}^{n\times n}$ is Hermitian and positive semidefinite. This is a generalization of the polar representation $z = re^{i\theta}$ for

* Corresponding author. Email: na.nhigham@na-net.ornl.gov The work of this author was supported by Science and Engineering Research Council grant GR/H52139, and by the EEC Esprit Basic Research Action Programme, Project 6634 (APPARC).

† This author was supported by an SERC Research Studentship.

SSDI 0167-8191(94)00016-4

complex numbers. The matrix H can be written as $H = (A^*A)^{1/2}$, where $G^{1/2}$ denotes the unique Hermitian positive semidefinite square root of a Hermitian positive semidefinite matrix G. If A has full rank then H is nonsingular and U is unique.

The polar decomposition is an important tool in various applications. These include aerospace computations [3,4], chemistry (Löwdin orthogonalization) [8], factor analysis [21], multidimensional scaling in statistics [10], signal processing [2], computation of block reflectors in numerical linear algebra [22], and mechanics [23]. The reason for the widespread use of the polar decomposition is that the unitary factor U solves two approximation problems. (By a unitary rectangular matrix we mean one with orthonormal columns; the term is usually applied only to square matrices.) For $A \in \mathbf{C}^{m\times n}$, U is a nearest unitary matrix to A. To be precise,

$$\| A - U \| = \min\{ \| A - Q \| : Q^*Q = I,\ Q \in \mathbf{C}^{m\times n}\}$$

for both the 2-norm and the Frobenius norm, and when $m = n$ the result is true for any unitarily invariant norm [6]. (Recall that $\| A \|_2$ is the square root of the largest eigenvalue of A^*A, $\| A \|_F = (\Sigma_{i,j} | a_{ij} |^2)^{1/2}$, and a norm is unitarily invariant if $\| UAV \| = \| A \|$ for all square, unitary U and V.) Furthermore, if A, $B \in \mathbf{C}^{m\times n}$ and $B^*A = UH$ is a polar decomposition, then

$$\| A - BU \|_F = \min\{ \| A - BQ \|_F : Q^*Q = I,\ Q \in \mathbf{C}^{n\times n}\}.$$

The latter minimization problem is called the orthogonal Procrustes problem; it arises in several of the applications mentioned above, most notably in factor analysis.

One way to compute the polar decomposition is via the singular value decomposition (SVD). If $A \in \mathbf{C}^{m\times n}$ has the SVD

$$A = P\begin{bmatrix} D \\ 0 \end{bmatrix} Q^*, \quad P \in \mathbf{C}^{m\times n}, \quad D = \operatorname{diag}(d_i) \in \mathbf{C}^{n\times n}, \quad Q \in \mathbf{C}^{n\times n},$$

then

$$U = P\begin{bmatrix} I \\ 0 \end{bmatrix} Q^*, \quad H = QDQ^*. \tag{1.1}$$

These formulae provide a numerically stable way to evaluate U and H, as long as the SVD is computed in a stable way. The standard way to compute the SVD is with the Golub-Reinsch SVD algorithm, as implemented in LAPACK [1]. Unfortunately, this algorithm is not very amenable to parallel computation. Moreover, the SVD approach to computing U and H has the drawback that it does not take advantage of situations in which the columns of A are nearly orthonormal (that is, $\| A - U \|$ is small). Near orthonormality arises in aerospace applications, for example, in which a direction cosine matrix whose columns drift from orthonormality is periodically orthonormalized by replacing it with its unitary polar factor. An alternative approach that does take advantage of a nearly unitary A is to use the Newton iteration [11]

$$Y_{k+1} = \tfrac{1}{2}(Y_k + Y_k^{-*}), \quad Y_0 = A \in \mathbf{C}^{n\times n}. \tag{1.2}$$

This is Newton's method applied to the equation $Y^TY = I$. For any nonsingular A, Y_k converges to U quadratically as $k \to \infty$, and if A is nearly unitary only a few iterations are needed. However, the only opportunity for exploiting parallelism is in the computation of the matrix inverse (this is one of the few instances in scientific computation where it really is necessary to compute a matrix inverse). The following iteration, which applies to rectangular matrices, is an attractive alternative for parallel computation:

$$X_{k+1} = \frac{1}{p} X_k \sum_{i=1}^{p} \frac{1}{\xi_i} \left(X_k^* X_k + \alpha_i^2 I \right)^{-1}, \quad X_0 = A \in \mathbf{C}^{m \times n}, \tag{1.3}$$

where

$$\xi_i = \tfrac{1}{2}\left(1 + \cos\frac{(2i-1)\pi}{2p}\right), \quad \alpha_i^2 = \frac{1}{\xi_i} - 1, \quad i = 1\colon p. \tag{1.4}$$

This iteration is derived by Higham [12] from a corresponding iteration of Pandey, Kenney and Laub [20] for the matrix sign function. For any positive integer p and any full rank $A \in \mathbf{C}^{m \times n}$ $(m \geq n)$ the iteration converges to the polar factor U of A with order of convergence $2p$. Iteration (1.3) is related to the Newton iteration (1.2) as follows: if $p = 1$, so that

$$X_{k+1} = 2X_k \left(X_k^* X_k + I \right)^{-1}, \tag{1.5}$$

then

$$X_k = Y_k \Rightarrow X_{k+1} = Y_{k+1}^{-*}.$$

If p is a power of 2, iteration (1.3) can be derived by combining $\log_2 p + 1$ steps of (1.5) and expressing the result in partial fraction form. These properties can be combined and expressed as follows: (1.2) and (1.3) satisfy

$$X_k = Y_k \Rightarrow X_{k+1} = Y_{k+\log_2 p+1}^{-*}. \tag{1.6}$$

The p matrix inverses in (1.3) can be evaluated in parallel, so the iteration has inherent high-level parallelism. However, when we work out approximate operation counts for the iterations (1.2) and (1.3) for $m = n$, taking into account (1.6), we find that iteration (1.3) has the greater operation count by a factor approximately $(3+p)/(2\log_2 p + 2)$, so the parallelism is obtained at the expense of extra floating point operations. In this work we develop a parallel implementation of (1.3) for the Kendall Square Research KSR1 computer and compare it with a parallel implementation of (1.2). In Section 2 we consider mathematical issues, including scaling to improve the speed of convergence, and numerical stability. In Section 3 we describe the implementation details and in Section 4 we give experimental results. Our conclusions are presented in Section 5.

2. Mathematical issues

Although the Newton iteration (1.2) is quadratically convergent, convergence can be slow initially. For real data with $n = 1$, the iteration is $y_{k+1} = \frac{1}{2}(y_k + y_k^{-1})$,

and if $y_0 = a \gg 1$ then in the early iterations $y_{k+1} \approx y_k/2$ and the error $y_k - 1$ is only approximately halved on each step. The speed of convergence can be improved by scaling the iterates $Y_k \leftarrow \gamma_k Y_k$, so that the iteration becomes

$$Y_{k+1} = \tfrac{1}{2}\left(\gamma_k Y_k + \gamma_k^{-1} Y_k^{-*}\right), \quad Y_0 = A \in \mathbf{C}^{n\times n}. \tag{2.1}$$

This idea has been investigated in [11], where it is shown that the scaling factor

$$\gamma_k = \left(\sigma_{\max}(Y_k)\sigma_{\min}(Y_k)\right)^{-1/2} \tag{2.2}$$

minimizes a bound on $\| Y_{k+1} - U \|_2$, where $\sigma_{\max}(Y_k)$ and $\sigma_{\min}(Y_k)$ are the largest and smallest singular values of Y_k, respectively. In practice, γ_k is too expensive to compute exactly, so we use an estimate of it, such as

$$\hat{\gamma}_k = \left(\frac{\| Y_k^{-1} \|_1 \| Y_k^{-1} \|_\infty}{\| Y_k \|_1 \| Y_k \|_\infty} \right)^{1/4}. \tag{2.3}$$

Kenney and Laub [18] give a thorough analysis of scaling the Newton iteration. A particularly interesting result is that with the optimal scaling parameter (2.2), $Y_k = U$ where k is the number of distinct singular values of A; that is, exact convergence is obtained in $k \le n$ iterations. Practical experience shows that even using the approximate scaling parameter (2.3), convergence is almost always obtained in ten iterations or less (to a convergence tolerance of 10^{-16} or greater).

Because of the relation (1.6) between the Newton iteration and (1.3), precisely the same comments about scaling apply to (1.3): scaling is vital to avoid unduly slow convergence (at least for small p) and the scaling parameter (2.2) is optimal in the sense that it minimizes an error bound. To be precise, the scaled version of (1.3) is

$$X_{k+1} = \frac{1}{p}\mu_k X_k \sum_{i=1}^{p} \frac{1}{\xi_i}\left(\mu_k^2 X_k^* X_k + \alpha_i^2 I\right)^{-1}, \quad X_0 = A \in \mathbf{C}^{m\times n}, \tag{2.4}$$

where μ_k is given by (2.2) or (2.3), with Y_k replaced by X_k. Note that, in view of (1.6), the effect of the scaling in (2.4) is the same as if we scaled only one in every $\log_2 p + 1$ Newton iterates. Thus the increased opportunity for parallelism brings with it a decreased opportunity to scale. Furthermore, the scaling parameter μ_k is nontrivial to compute since we do not form X_k^{-1} in the course of the iteration.

One advantage of (2.4) over the Newton iteration is that it is easier to test for convergence: because we form $X_k^* X_k$ during the iteration we can evaluate cheaply the upper bound in the inequalities [12, Lemma 5.1]

$$\frac{\| X_k^* X_k - I \|_F}{\| X_k \|_2 + 1} \le \| X_k - U \|_F \le \| X_k^* X_k - I \|_F \quad (A = UH).$$

The presence of $X_k^* X_k$ in (2.4) is also a disadvantage because of the condition squaring effect of forming the cross-product matrix and because there can be severe loss of information when $X_k^* X_k$ is evaluated in floating point arithmetic. These observations suggest that the iteration is potentially numerically unstable when A is ill-conditioned.

To examine the stability we need some way of measuring the quality of an approximate polar factor. We assume for the rest of this section that A is a square matrix (certain of the equalities below do not hold for rectangular A.) A natural definition of backward error of a unitary approximation V to U is the quantity

$$\beta(V) = \min\{ \| E \|_F : V \text{ is the unitary polar factor of } A + E\}$$
$$= \min\{ \| E \|_F : V^*(A+E) \text{ is Hermitian positive semidefinite}\}.$$

The following lemma shows how to evaluate $\beta(V)$, or at least a lower bound for it.

Lemma 2.1. *Let $A \in \mathbf{C}^{n\times n}$ and let $V \in \mathbf{C}^{n\times n}$ be unitary. Then*

$$\min\{ \| E \|_F : V^*(A+E) \text{ is Hermitian}\} = \tfrac{1}{2} \| A^*V - V^*A \|_F$$

and the minimum is achieved for $E_{\mathrm{opt}} = \frac{1}{2}(VA^*V - A)$. If $V^*(A + E_{\mathrm{opt}}) = \frac{1}{2}(A^*V + V^*A)$ is positive semidefinite then E_{opt} solves

$$\min\{ \| E \|_F : V^*(A+E) \text{ is Hermitian positive semidefinite}\}.$$

Proof. Let $V^*E \equiv G + K$, where $G = G^*$ and $K = -K^*$. The requirement that $V^*(A+E)$ is Hermitian can be written $2K = A^*V - V^*A$. Hence

$$\| E \|_F^2 = \| V^*E \|_F^2 = \| G \|_F^2 + \tfrac{1}{4} \| A^*V - V^*A \|_F^2.$$

Since G is arbitrary, the minimizing E is achieved when $G = 0$. This establishes the first part. The second part is immediate since the set of allowable E in the second minimization problem is contained in that for the first. □

The lemma implies that given a unitary approximation V to U, the best choice of H is $H = \frac{1}{2}(A^*V + V^*A)$, assuming this matrix is positive semidefinite. If it *is* positive semidefinite then $\beta(V) = \frac{1}{2} \| A^*V - V^*A \|_F$, and otherwise $\beta(V) > \frac{1}{2} \| A^*V - V^*A \|_F$. It is easy to show that $\frac{1}{2} \| A^*V - V^*A \|_F = \| A - VH \|_F$ (when A, $V \in \mathbf{C}^{n\times n}$), so $\beta(V)$ is greater than or equal to the residual of the approximate polar decomposition.

A computed approximation $\hat{U}$ to U will usually be unitary only to within roundoff, but with $V = \hat{U}$ all the equalities in the lemma are still true to within roundoff. Thus in practice it is natural to take for the computed Hermitian polar factor

$$\hat{H} = \frac{(\hat{U}^*A)^* + \hat{U}^*A}{2}.$$

This is the choice used in [11] and [13], but the justification given here is new.

The stability properties of iteration (2.4) are clearly illustrated by its performance on the 10×10 Vandermonde matrix with (i, j) element $((j-1)/(n-1))^{i-1}$. This matrix has 2-norm condition number $\kappa_2(A) = \| A \|_2 \| A^{-1} \|_2 = 1.52 \times 10^7$. We implemented the iteration in Matlab both with and without scaling. For the scaling we used (2.3) (with Y_k replaced by X_k), but only scaled while $\| X_k^* X_k - I \|_F > 10^{-2}$, for after this point convergence is rapid. In the unscaled iteration we

Table 1
Behaviour on a 10×10 Vandermonde matrix

Iteration	Scaling	Iters	$\beta(\hat{U})/\|A\|_F$	$\max_{i,k} \kappa_2(\mu_k^2 X_k^* X_k + \alpha_i^2 I)$
Newton	Unscaled	29	8.10E−12	–
	Scaled	8	4.10E−16	–
$p=1$	Unscaled	29	3.15E−16	2.00E+00
	Scaled	8	1.37E−09	9.21E+06
$p=2$	Unscaled	15	5.74E−16	6.83E+00
	Scaled	5	6.95E−09	5.37E+07
$p=4$	Unscaled	10	1.22E−15	2.63E+01
	Scaled	4	2.40E−09	2.33E+08
$p=8$	Unscaled	8	2.22E−15	1.04E+02
	Scaled	4	6.00E−09	9.49E+08
$p=16$	Unscaled	6	9.64E−15	4.15E+02
	Scaled	3	3.60E−09	3.82E+09

scaled the starting matrix X_0 to have unit Frobenius norm, for reasons explained below. The iterations were terminated when $\|X_k^* X_k - I\|_F \le nu$, where the unit roundoff $u \approx 1.1 \times 10^{-16}$. The results are summarized in Table 1, which also includes results for the Newton iteration (2.1), for comparison. In each case the computed $\hat{H}$ was positive semidefinite, so we were able to evaluate $\beta(\hat{U})$.

The most striking feature of the results is that the scaled iteration (2.4) is unstable for all p, the backward error being seven orders of magnitude larger than we would like. The reason for the instability is that some of the matrices $C_i = \mu_k^2 X_k^* X_k + \alpha_i^2 I$ are ill-conditioned; the fourth column of the table reports the maximum 2-norm condition number over all terms i and iterations k. These ill-conditioned terms arise when $\gamma_k X_k$ is ill-conditioned and has norm greatly exceeding 1. Since C_i is symmetric positive definite we can write down an expression for $\kappa_2(C_i)$ in terms of the singular values of X_k:

$$\kappa_2(C_i) = \frac{(\mu_k \sigma_{\max}(X_k))^2 + \alpha_i^2}{(\mu_k \sigma_{\min}(X_k))^2 + \alpha_i^2} \le \frac{(\mu_k \sigma_{\max}(X_k))^2}{\min_i \alpha_i^2} + 1. \tag{2.5}$$

For the scaled iteration the best bound that holds for all k can be shown to be $\|\mu_k X_k\|_2 \le \kappa_2(A)^{1/2}$ (with equality if $k = 0$) and so the best bound for $\kappa_2(C_i)$ is of order $\kappa_2(A)$. By a heuristic stability argument we would expect the backward error of the computed $\hat{U}$ to be proportional to $\kappa_2(A)$ when scaling is used; this prediction is borne out in Table 1 and in the results reported in Section 4.

If we do not use scaling then $\kappa_2(C_i)$ is nicely bounded, provided $\|X_0\|_2$ is not too large, because then all the X_k are not large in norm. This follows from the observation that with no scaling $\sigma_{\max}(X_k) \le 1$ for all $k \ge 1$, which is clear for iteration (1.5) and therefore follows for iteration (1.3). If $p = 16$ then $\min_i \alpha_i^2 = 2.41 \times 10^{-3}$ and (2.5) (with $\mu_k \equiv 1$) yields $\kappa_2(C_i) \le 416$ for all i and all $k \ge 1$; this is a sharp inequality, as can be seen from the $p = 16$ entry in Table 1.

The initial scaling $X_0 \leftarrow X_0/\|X_0\|_F$ ensures that $\sigma_{\max}(X_0) \le 1$. When X_0 is nearly unitary this scaling is unsatisfactory because it tends to increase $\|X_0 - U\|_F$,

since $\| Q \|_F = \sqrt{n}$ for unitary Q. Alternative initial scalings that do not have this drawback are $X_0 \leftarrow (\| X_0 \|_F / \| X_0^T X_0 \|_F) X_0$ (which solves the minimization problem $\min_\alpha \| I - (\alpha X_0)^* (\alpha X_0) \|_F$), $X_0 \leftarrow \sqrt{n} X_0 / \| X_0 \|_F$, or simply no scaling. The unit Frobenius norm scaling has the advantage of giving the smallest bound for $\kappa_2(C_i)$ when $k = 0$.

It is clear, then, that while scaling can greatly improve the speed of convergence, it is likely to degrade the stability when A is ill-conditioned (which is precisely the situation where scaling is most needed!). We emphasize that, by contrast, the scaled Newton iteration (2.1) is almost always stable in practice. However, as is clear from Table 1, the benefits of scaling become less pronounced as p increases, so in a parallel implementation with $p \geq 8$ (say) it may be acceptable not to scale. We defer a final assessment of the desirability of scaling until Section 5.

3. Implementation on the KSR1

The Kendall Square Research KSR1 is a virtual shared memory MIMD computer system. Unlike typical high-performance computers, which have large pools of main memory and small caches, the KSR1 has main memory consisting of large, communicating local caches, each capable of storing 32 megabytes. The programmer perceives the KSR1 memory system as a collection of processors connected to a shared memory. The shared memory is addressed by a 64-bit System Virtual Address (SVA). The contents of SVA locations are physically stored in a distributed fashion, in a collection of local caches. There is one local cache for each processor in the system. The *search engine* interconnects the local caches and provides routing and directory services for the collection of local caches. As a result, all the local caches behave logically as a single, shared address space. The combination of the local caches and the search engine is referred to as ALLCACHE memory.

The KSR1 installed in the Centre for Novel Computing at the University of Manchester is a 32-processor configuration with 1 gigabyte of memory and 10 gigabytes disk capacity. Its peak computational performance is 1.28 gigaflops. Each KSR1 processor is a superscalar 64-bit unit able to issue up to two instructions every 50 nanoseconds, giving a performance rating of 40 MIPS and a peak floating point performance of 40 megaflops. Our applications are written in KSR Fortran, an extension of Fortran 77 for the KSR1. In KSR Fortran low level parallelism is expressed with *pthreads*, while high level parallelism is expressed using *parallel regions, parallel sections*, and *tile families*. A *pthread* is a sequential flow of control within a process that cooperates with other pthreads to solve a problem. *Parallel regions* execute a single segment of code in parallel multiple times, and *parallel sections* execute multiple code segments in parallel. In our code we used all the above parallel constructs except parallel sections, which are not needed for our application. Loop parallelization in KSR Fortran is achieved by *tiling*. The tile directive causes the iteration space to be partitioned into rectilinear regions called

tiles, each of which contains enough loop iterations to create a reasonable amount of work for one processor. Efficient exploitation of tiling can greatly improve the performance of a code but tiling must be avoided for simple, small loops where the startup cost may outweigh the gain. Tiling can be accomplished in three ways: manually (via directives), semi-automatically (via directives) and fully automatically. In our codes we use semi-automatic tiling because it offers the user full control over tiling, while providing automatic correctness checking. More information on the KSR1 can be found in [9,14,15].

Our parallel algorithm for computing the polar decomposition has the following outline, which is based on the Matlab version mentioned above that we used for prototyping. We restrict our attention to real, square matrices.

Algorithm Parallel Polar.
Given a nonsingular matrix $A \in \mathbb{R}^{n\times n}$ and a convergence tolerance tol, this algorithm computes the polar decomposition $A = UH$. The algorithm uses iteration (2.4) and requires p processors.
$X := A$
if 'scaling is not required' and A is not nearly orthogonal, $X := X/\| X \|_F$,
end
Compute the coefficients ξ_i, α_i^2 $(i = 1{:}p)$ in (1.4).
repeat
(1) Compute $C := X^*X$ using parallel matrix multiply.
$\rho := \| C - I \|_F$
if $\rho \le$ tol, goto (5), end
if $\rho > 10^{-2}$ and 'scaling is required'
(2) Compute $W := X^{-1}$ by parallel matrix inversion.
$\mu := ((\| W \|_1 \| W \|_\infty)/(\| X \|_1 \| X \|_\infty))^{1/4}$
else
$\mu := 1$
end
Form $C_i := \mu^2 C + \alpha_i^2 I$ on each processor i $(i = 1{:}\mathrm{p})$.
(3) Compute $T_i := C_i^{-1}$ on each processor i $(i = 1{:}p)$.
$S := \sum_{i=1}^{p} T_i$
(4) Compute $X := (\mu/p)XS$, using parallel matrix multiply.
end repeat
(5) $U := X$
Compute $H_1 := U^*A$ using parallel matrix multiply.
$H := \frac{1}{2}(H_1^* + H_1)$ ('best H' by Lemma 2.1).

Our philosophy in implementing this algorithm was to use the best routines available to us on the KSR1 for the higher level linear algebra operations (matrix multiplication, matrix inversion, etc.) and not to produce our own implementations of these routines. Thus we used routines from LAPACK [1] and the BLAS as our building blocks. It would have been interesting to try alternative inversion algo-

rithms, such as the Newton iteration from [19], but this was beyond the scope of this work.

The matrices $C_i = \mu_k^2 X_k^* X_k + \alpha_i^2 I$ in (2.4) are symmetric positive definite. Therefore, for the inversions at step (3) of the algorithm we used the LAPACK routines SPOTRF and and SPOTRI. SPOTRF computes the Cholesky factorization of a real symmetric positive definite matrix and SPOTRI computes its inverse using the Cholesky factorization. The inversion of X at step (2) is necessary only when scaling is required. We do the inversion in parallel using the LAPACK routine SGETRF, which computes the LU factorization of a general matrix, and SGETRI, which computes its inverse using the LU factors.

We used the level 3 BLAS implementation supplied by Kendall Square Research in the KSRlib/BLAS Library [16]. This contains highly optimized xGEMM general matrix multiply routines. In our tests we found that SGEMM runs at 384 megaflops on 16 processors for a matrix of dimension 1024, which is over half the peak megaflop rate. When this work was begun a KSR1 implementation of LAPACK was not available to us, so we used the standard LAPACK distribution. We experimented with the block size in the LAPACK routines mentioned above (the block size is set in the environmental enquiry routine ILAENV) and found that a block size of 16 gives the best all-round performance on the KSR1. Therefore all our results are for a block size of 16. After our testing was complete we gained access to the KSRlib/LAPACK Library [17] and found that it gives almost identical performance to our own implementation.

We formed X^*X in step (1) of the algorithm using SGEMM because we found this to be faster in parallel, for the KSRlib/BLAS Library, than SSYRK (which takes advantage of symmetry but is not highly optimized in this library). Consequently, our implementation of Algorithm Parallel Polar does not fully exploit the symmetry of X^*X.

To obtain the best performance from the KSR1 we used the following two devices. When calling SGEMM with square matrices of order 1024 we set the extra, KSR1-specific parameter ps to 1, instead of the value 4 recommended in [16]; this gives a performance increase of about 27%. When dimensioning $n \times n$ arrays we set the trailing dimension to $n + 2$, which also leads to a performance improvement for $n = 1024$ because it results in better use of cache memory.

4. Experimental results

We compared the performance of three methods for computing the polar decomposition on the KSR1.

(1) The SVD method that first computes the SVD and then forms U and H according to (1.1). We compute the SVD using LAPACK's routine SGESVD using one processor. We found that the run time was larger when we used more than one processor with fully automatic parallelization by the compiler; this is apparently because the compiler parallelizes every DO loop and the startup costs for the small DO loops are significant.

Table 2
Algorithm Parallel Polar with 8 processors, $n = 1024$

$\kappa_2(A)$	Time	$\beta(\hat{U})/\|A\|_F$	Iters	Speedup	Scalings
1.01E0	204.82	4.3E−16	1	7.58	OFF
1.0E01	580.36	9.3E−16	3	7.62	OFF
1.0E04	1130.03	5.3E−15	6	7.69	OFF
1.0E08	1767.03	1.0E−14	9	7.74	OFF
1.0E12	2248.80	1.4E−14	12	7.78	OFF
1.01E0	286.34	4.3E−16	1	6.71	1
1.0E01	491.11	8.8E−16	2	6.59	1
1.0E04	775.92	3.1E−13	3	6.21	2
1.0E08	1085.83	2.4E−09	4	5.93	3
1.0E12	1124.59	2.0E−05	4	5.81	4

(2) Algorithm Parallel Polar for $p > 1$, both with and without scaling.

(3) The Newton iteration (1.2) with and without scaling, with the inversions done in parallel using LAPACK's `SGETRF` and `SGETRI` and with H computed as in Algorithm Parallel Polar. Scaling is done only while $\|Y_{k+1} - Y_k\|_F > 10^{-2}\|Y_k\|_F$.

We used real matrices of dimension up to 1024, and all the results presented here are for this maximum dimension. For each dimension we generated random matrices $A = PDQ^T$, where P and Q are random orthogonal matrices and $D = \mathrm{diag}(\sigma_i)$, with exponentially distributed singular values $\sigma_i = \alpha^i$ $(0 < \alpha < 1)$, so that $\kappa_2(A) = \alpha^{1-n}$. We chose a range of condition numbers $\kappa_2(A) = 1.01$, 10, 10^4, 10^8 and 10^{12}, to observe the effect of the condition number on the behaviour of the iterative methods. The matrices with $\kappa_2(A) = 1.01$ are such that $\|A - U\|_2 \approx 0.01$, so they are moderately close to being orthogonal. We used the convergence tolerance $\mathrm{tol} = nu$, where $u \approx 1.1 \times 10^{-16}$ is the unit roundoff for single precision arithmetic on the KSR1.

All the results reported were obtained using version 1.0 (March 1993) of the KSR Fortran compiler.

The timings for the SVD method are all between 9400 and 9500 seconds; as expected the condition number has little effect on the times.

Timings (in seconds) for Algorithm Parallel Polar with $p = 8$ and 16 processors are given in Tables 2 and 3. (We did not have access to all 32 processors of our KSR1 configuration). The speedup is defined as the time for iteration (2.4) with a given value of p implemented on a single processor divided by the time for Algorithm Parallel Polar implemented on p processors. The column headed 'scalings' indicates either that no scaling was attempted ('OFF'), or states the number of iterations on which scaling was used (which is the number of times the inversion step (2) of the algorithm is executed). Timings for the Newton iteration on 16 processors are given in Table 4; the speedup is the run time for the iteration with 16 processors divided by the run time for the same iteration with 1 processor.

We make several observations.

(1) Algorithm Parallel Polar with $p = 8$ or 16 is between 8 and 51 times faster

Table 3
Algorithm Parallel Polar with 16 processors, $n = 1024$

$\kappa_2(A)$	Time	$\beta(\hat{U})/\|A\|_F$	Iters	Speedup	Scalings
1.01E0	184.17	4.3E−16	1	15.15	OFF
1.0E01	425.26	8.1E−16	2	15.26	OFF
1.0E04	1025.48	8.3E−15	5	15.39	OFF
1.0E08	1320.96	1.9E−14	7	15.42	OFF
1.0E12	1868.04	2.6E−14	10	15.48	OFF
1.01E0	243.08	2.7E−16	1	13.28	1
1.0E01	554.07	8.2E−16	2	13.19	1
1.0E04	647.08	3.2E−13	3	12.79	2
1.0E08	717.82	2.3E−09	3	11.82	3
1.0E12	1144.05	2.0E−05	5	11.56	3

than the SVD approach, the greater speedups being for well-conditioned matrices. It is also between two and four times faster than the parallel implementation of the Newton iteration (1.2) when both iterations are scaled. The speed is a consequence of the high-level parallelism and the implementation's richness in level 3 BLAS operations. For the first entry in Table 3 ($p = 16$, $\kappa_2(A) = 1.01$) our code is running at a speed of about 140 megaflops.

(2) The speedups for algorithm Parallel Polar without scaling are close to optimal. When scaling is used, the inversions at step (2) of the algorithm degrade the speedup because matrix inversion using LAPACK's `SGETRF` and `SGETRI` is not highly parallel (we measured an execution rate of only 39.3 megaflops for inversion of a matrix of size 1024 on 16 processors). Similarly, the speedups for the Newton iteration are poor because the iteration is dominated by matrix inversions.

(3) The backward errors for Algorithm Parallel Polar are as predicted by the analysis in Section 2. The maximum backward error without scaling is of order 10^2u, which is consistent with the convergence tolerance $\text{tol} = nu =$

Table 4
Newton iteration with 16 processors, $n = 1024$

$\kappa_2(A)$	Time	$\beta(\hat{U})/\|A\|_F$	Iters	Speedup	Scalings
1.01E0	1862.22	3.4E−14	10	2.28	OFF
1.0E01	2356.81	5.4E−14	13	2.28	OFF
1.0E04	3739.51	2.6E−12	21	2.29	OFF
1.0E08	6262.17	6.8E−09	34	2.29	OFF
1.0E12	8500.59	3.2E−05	47	2.30	OFF
1.01E0	745.23	2.3E−14	4	2.30	1
1.0E01	1133.31	3.4E−14	6	2.30	4
1.0E04	1490.46	3.3E−14	8	2.31	5
1.0E08	1664.81	3.2E−14	9	2.31	6
1.0E12	1862.53	3.2E−14	10	2.32	7

$1024u$ and the bound $\max_{i,k}\kappa_2(C_i) \le 416$ mentioned in Section 2. The backward errors with scaling are all approximately $\kappa_2(A)u/5$.

(4) Algorithm Parallel Polar takes good advantage of well-conditioned or nearly orthogonal A, requiring only one iteration if A is sufficiently close to being orthogonal.

(5) Increasing p beyond 16 in iteration (2.4) produces diminishing improvements in the number of iterations, particularly for well-conditioned A. If more than 16 processors are available (and the KSR1 supports up to 1088 processors) the most promising way to improve our timing results is to use more than one processor to execute each of the parallel segments of Algorithm Parallel Polar. For example, the simultaneous Cholesky factorizations would be executed faster if we implemented the parallel algorithm of George, Heath and Liu for computing the Cholesky factorization on a shared memory computer [7].

5. Conclusions

Iteration (1.3) for computing the unitary polar factor of $A \in \mathbf{C}^{m\times n}$,

$$X_{k+1} = \frac{1}{p} X_k \sum_{i=1}^{p} \frac{1}{\xi_i} \left(X_k^* X_k + \alpha_i^2 I \right)^{-1}, \quad X_0 = A, \tag{5.1}$$

has inherent high-level parallelism that enables it to be implemented efficiently on the KSR1, with run times an order of magnitude smaller than are obtained using our implementation of the LAPACK SVD routine. With scaling, this iteration is two to four times as fast as the scaled Newton iteration (2.1) (which is applicable only to square matrices). We would expect the iteration to be successful on other shared memory parallel computers with a relatively small number of processors (indeed, this has already been demonstrated for the Cray Y-MP with 4 processors by Pandey, Kenney and Laub [20], who used the matrix sign function analogue of iteration (5.1)). As is often the case in numerical linear algebra, there is a tradeoff between speed and stability [5]: faster convergence is obtained when scaling is used, but this usually makes the backward error proportional to $\kappa_2(A)$. In some applications this is not an issue because A is known a priori to be well-conditioned or nearly unitary. In general, if high accuracy is important it is better to use the unscaled iteration, which is at worst about twice as slow as the scaled version for $p = 8$ or 16, but which is still at least as fast as the Newton iteration and which has excellent stability.

Acknowledgements

We thank Graham Riley of the Centre for Novel Computing at the University of Manchester for his advice on the use of the KSR1. We also thank Len Freeman, Des Higham and Rob Schreiber for suggesting improvements to the manuscript.

References

[1] E. Anderson, Z. Bai, C.H. Bischof, J.W. Demmel, J.J. Dongarra, J.J. Du Croz, A. Greenbaum, S.J. Hammarling, A. McKenney, S. Ostrouchov and D.C. Sorensen. LAPACK Users' Guide, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[2] K.S. Arun, A unitarily constrained total least squares problem in signal processing, *SIAM J. Matrix Anal. Appl.* 13(3) (1992) 729–745.

[3] I.Y. Bar-Itzhack, Iterative optimal orthogonalization of the strapdown matrix, *IEEE Trans. Aerospace and Electronic Syst.* AES-11(1) (1975) 30–37.

[4] Å. Björck and C. Bowie, An iterative algorithm for computing the best estimate of an orthogonal matrix, *SIAM J. Numer. Anal.* 8(2) (1971) 358–364.

[5] J.W. Demmel, Trading off parallelism and numerical stability, in: M.S. Moonen, G.H. Golub and B.L. De Moor, eds., *Linear Algebra for Large Scale and Real-Time Applications*, vol. 232 of *NATO ASI Series E* (Kluwer Academic Publishers, Dordrecht, 1993) 49–68.

[6] K. Fan and A.J. Hoffman, Some metric inequalities in the space of matrices, *Proc. Amer. Math. Soc.* 6 (1955) 111–116.

[7] A. George, M.T. Heath and J. Liu, Parallel Cholesky factorization on a shared-memory multiprocessor, *Linear Algebra and Appl.* 77 (1986) 165–187.

[8] J.A. Goldstein and M. Levy, Linear algebra and quantum chemistry, *Amer. Math. Monthly* 98(8) (1991) 710–718.

[9] A. Gottlieb, Architectures for parallel supercomputing, Technical report, Ultracomputer Research Laboratory, New York University, Dec. 1992.

[10] J.C. Gower, Multivariate analysis: Ordination, multidimensional scaling and allied topics, in: E.H. Lloyd, ed., *Statistics*, vol. VI of *Handbook of Applicable Mathematics* (John Wiley, Chichester, 1984) 727–781.

[11] N.J. Higham, Computing the polar decomposition – with applications, *SIAM J. Sci. Stat. Comput* 7(4) (Oct. 1986) 1160–1174.

[12] N.J. Higham, The matrix sign decomposition and its relation to the polar decomposition, Numerical Analysis Report No. 225, University of Manchester, England, April 1993, to appear in *Linear Algebra and Appl.*

[13] N.J. Higham and R.S. Schreiber, Fast polar decomposition of an arbitrary matrix, *SIAM J. Sci. Stat. Comput* 11(4) (July 1990) 648–655.

[14] Kendall Square Research Corporation, KSR Fortran Programming, Waltham, MA, 1991.

[15] Kendall Square Research Corporation, KSR Parallel Programming, Waltham, MA, 1991.

[16] Kendall Square Research Corporation, KSRlib/BLAS Library, Version 1.0, Installation Guide and Release Notes, Waltham, MA, 1993.

[17] Kendall Square Research Corporation, KSRlib/LAPACK Library, Version 1.0b BETA, Installation Guide and Release Notes, Waltham, MA, 1993.

[18] C. Kenney and A.J. Laub, On scaling Newton's method for polar decomposition and the matrix sign function, *SIAM J. Matrix Anal. Appl.* 13(3) (1992) 688–706.

[19] V. Pan and R. Schreiber, An improved Newton iteration for the generalized inverse of a matrix, with applications, *SIAM J. Sci. Stat. Comput.* 12(5) (1991) 1109–1130.

[20] P. Pandey, C. Kenney and A.J. Laub, A parallel algorithm for the matrix sign function, *Int. J. High Speed Comput.* 2(2) (1990) 181–191.

[21] P.H. Schönemann, A generalized solution of the orthogonal Procrustes problem, *Psychometrika* 31(1) (1966) 1–10.

[22] R.S. Schreiber and B.N. Parlett, Block reflectors: Theory and computation, *SIAM J. Numer. Anal.* 25(1) (1988) 189–205.

[23] I. Söderkvist and P.Å. Wedin, A practical condition number for a configuration of points defining a rigid body movement, Report UMINF-91.29, Institute of Information Processing, University of Umeå, Sweden, Dec. 1991.