Abstract. Two new algorithms for one-sided bidiagonalization are presented. The first is a block version which improves execution time by improving cache utilization from the use of BLAS 2.5 operations and more BLAS 3 operations. The second is adapted to parallel computation. When incorporated into singular value decomposition software, the second algorithm is faster than the corresponding ScaLAPACK routine in most cases. An error analysis is presented for the first algorithm. Numerical results and timings are presented for both algorithms.

Key words. singular value decomposition, bidiagonalization, block algorithm, parallel algorithm, numerical analysis

AMS subject classifications. 15A18, 65F30, 68W10

DOI. 10.1137/050636723

1. Introduction. There are two main types of algorithms for computing the complete singular value decomposition (SVD) of a matrix $A$: one-sided Jacobi methods [12] and algorithms based upon bidiagonalization. Recently there have been significant improvements in both types of methods [2], [3], [17], [18], [23], [24]; this work considers bidiagonalization-based algorithms. Such algorithms use orthogonal transformations to obtain a bidiagonal form and then apply a fast algorithm to obtain the SVD of the bidiagonal matrix [19], [24], [32].

Let $A \in \mathbb{R}^{m \times n}, m \geq n$ (for $m < n$, consider $A^T$). Then there exist a left orthogonal matrix $U \in \mathbb{R}^{m \times n}$ (i.e., $U^TU = In$), an orthogonal matrix $V \in \mathbb{R}^{n \times n}$ (i.e., $V^TV = VV^T = In$), and a bidiagonal matrix

$$B = \begin{bmatrix}
\psi_1 & \phi_2 \\
\psi_2 & \phi_3 \\
& \ddots \\
& & \psi_{n-1} & \phi_n \\
& & & \psi_n
\end{bmatrix},$$

such that

$$A = UBV^T.$$
p. 119] and Chan in [10] present a modification of the Golub–Kahan algorithm, which performs a QR factorization of the matrix $A$ before bidiagonalization, resulting in an algorithm with fewer operations if $m > 5/3n$ and $U$ is not explicitly formed. A bidiagonalization algorithm based on Givens rotations, which can attain higher accuracy, is given by Barlow [2]. Grosser and Lang in [23] propose a two-stage method, where in the first stage a matrix is reduced to a banded matrix and then in the second stage a banded matrix is bidiagonalized. This algorithm performs the majority of calculations in matrix-matrix products and can be faster than the ScaLAPACK bidiagonalization routine. Another different approach is described in [20], where the bidiagonal form is obtained by means of the Lanczos algorithm. This approach is fast but can be numerically unstable. Still there are many applications that do not suffer from this instability, and there is a simple procedure that corrects the possible loss of orthogonality; cf. [31].

Recently, in [28], [30], and [29] Ralha proposed a new approach for bidiagonalizing a matrix, the so-called one-sided bidiagonalization. The main idea is to implicitly tridiagonalize the matrix $A^T A$:

$$V^T A^T A V = T,$$

where $T$ is a tridiagonal matrix, by a one-sided orthogonal transformation of $A$

$$F = AV.$$

$V$ is computed as a product of Householder reflectors without explicitly forming $A^T A$. Then the Gram–Schmidt QR factorization of the matrix $F$ is performed to obtain

$$UB = F = AV,$$

where $B$ is upper triangular. In the case when $A$ has full rank, the matrix $B$ is the Cholesky factor of $T$, that is, $T = B^T B$, and hence $B$ must be bidiagonal. That property results in a fast bidiagonalization algorithm, which is much more suitable for parallel computing than the standard bidiagonalization algorithms. Here the modified Gram–Schmidt QR factorization of $F$ is computed by orthogonalizing the $k$th column of $F$ against the $(k-1)$th column of $U$ and normalizing. Unfortunately, this procedure is not always numerically stable and may lead to a matrix $U$ that is far from being left orthogonal in finite precision arithmetic.

To improve Ralha’s algorithm, Barlow, Bosner, and Drmač [3] proposed a modification, where one step of Gram–Schmidt orthogonalization and postmultiplication with one Householder reflector are performed simultaneously, resulting in a direct bidiagonalization algorithm. This algorithm produces exactly the same result as Ralha’s algorithm in exact arithmetic, but it turns out to be numerically stable in finite precision arithmetic.

The rest of this paper is organized as follows. The recently developed one-sided bidiagonalization from [3] is presented in section 2, together with a bound on its backward error. In section 3, a block version of the new one-sided bidiagonalization is introduced, and a detailed numerical analysis is given in Theorem 3.4. The results of numerical tests regarding efficiency of the block one-sided bidiagonalization are presented in section 4. Sections 5 and 6 deal with parallel versions of the new one-sided bidiagonalization and its efficiency.

2. The new stable one-sided bidiagonalization. The main difference between Ralha’s algorithm and the new stable algorithm proposed in [3] is that, in the
new bidiagonalization algorithm, transformations with Householder reflectors and the Gram–Schmidt orthogonalization are interlaced, whereas Ralha’s bidiagonalization separates these processes. The criteria for choosing Householder reflectors are also different.

Let $u_k$ be the $k$th column of the matrix $U$, and let $U_k = [u_1, \ldots, u_k]$ be a matrix containing the first $k$ columns of $U$. Further, let $V_k$ denote a Householder reflector such that

$$V_k = I - v_kv_k^T, \quad \text{where } \|v_k\|_2 = \sqrt{2}.$$ 

Then the new stable bidiagonalization can be described in its simplest form as follows:

- $A_0 = A$.
- For $k = 1, 2, \ldots$,
  - $u_k$ is produced from the $k$th column of $A_{k-1}$ by orthogonalization against $u_{k-1}$ (if $k > 1$) and normalization
    $$U_k = [u_1, \ldots, u_k], \quad k = 1, \ldots, n.$$ 
  - Householder reflector $V_k$ is chosen so that
    \begin{equation}
    U_k^T A_{k-1} V_k = B_k \in \mathbb{R}^{k \times n}, \quad B_k \text{ is bidiagonal,}
    \end{equation}

    \[\begin{array}{ccc}
    \ast & \ast & \ast \\
    \ast & \ast & \ast \\
    \ast & \ast & \ast \\
    \ast & \ast & \ast \\
    \ast & \ast & \ast \\
    \ast & \ast & \ast \\
    \ast & \ast & \ast \\
    \ast & \ast & \ast \\
    \end{array}\]

    \[\begin{array}{ccc}
    \ast & \ast & \ast \\
    \ast & \ast & \ast \\
    \ast & \ast & \ast \\
    \ast & \ast & \ast \\
    \ast & \ast & \ast \\
    \ast & \ast & \ast \\
    \ast & \ast & \ast \\
    \ast & \ast & \ast \\
    \end{array}\]

    and the matrix $A_{k-1}$ is postmultiplied with $V_k$
    $$A_k = A_{k-1} V_k, \quad k = 1, \ldots, n - 2.$$ 

- End of loop.
- $V$ is produced by accumulation of the Householder reflectors
    \begin{equation}
    V = V_1 \cdots V_{n-2}, \quad F = A_{n-2} = AV.
    \end{equation}

Remark 2.1. The elements of $A_{k-1}$ denoted by $\circ$ are used in the current step of the algorithm to compute the vector $z_k$, which determines the Householder reflector $V_k$. The Householder reflector $V_k$ is defined as

\begin{equation}
V_k = \begin{bmatrix} I_k & 0 \\ 0 & V_k \end{bmatrix},
\end{equation}

and the Householder reflector $V_k \in \mathbb{R}^{(n-k) \times (n-k)}$ is chosen so that $V_k z_k = \pm \|z_k\|_2 e_1$. The elements denoted by $\bullet$ are computed columns of $F$, and in the next steps they remain unchanged. The computed elements of $B$ are denoted by $\ast$. 

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.
Remark 2.2. In order to produce $U$ with $n$ columns, the matrix $A$ should have full column rank. If rank ($A$) < $n$, then $\psi_k = 0$ for some $k$. In [3], it is shown that this case can be easily handled by an $O(mn^2)$ postprocessing procedure that produces a decomposition

$$A = U_c B_c V_c^T,$$

where $U_c \in \mathbb{R}^{m \times p}$ and $V_c \in \mathbb{R}^{n \times p}$ are left orthogonal, $B_c \in \mathbb{R}^{p \times p}$ is upper bidiagonal and nonsingular, and $p = \text{rank}(A)$.

The following pseudocode provides the details of the described algorithm.$^1$

**Algorithm 2.1.** For $A \in \mathbb{R}^{m \times n}$, rank($A$) = $n > 2$, this algorithm computes a left orthogonal $U = [u_1, \ldots, u_n]$, a bidiagonal $B$ having the form (1.1), and an orthogonal $V = V^{(n-2)}$ such that $A = UBV^T$.

1. $A_0 = A$;
2. $f_1 = A(:, \cdot, 1); \quad \psi_1 = \|f_1\|_2$;
3. $u_1 = f_1/\psi_1$;

for $k = 1: n-2$

4. $z_k = A_{k-1}(:, k+1: n)^T u_k$;
5. $[\phi_{k+1}, v_k] = \text{householder}(z_k)$;
6. $A_k(:, : k) = A_{k-1}(:, : k)$;
7. $A_k(:, k+1: n) = A_{k-1}(:, k+1: n) - A_{k-1}(:, k+1: n) v_k v_k^T$;
8. $f_{k+1} = A_k(:, k+1)$;
9. $s_{k+1} = f_{k+1} - \phi_{k+1} u_k; \quad \psi_{k+1} = \|s_{k+1}\|_2$;
10. $u_{k+1} = s_{k+1}/\psi_{k+1}$;

end;

11. $f_n = A_{n-2}(:, n); \quad \phi_n = u_{n-1}^T f_n$;
12. $s_n = f_n - \phi_n u_{n-1}; \quad \psi_n = \|s_n\|_2$;
13. $u_n = s_n/\psi_n$;
14. $V^T = \text{householder_product}(v_1, \ldots, v_{n-2})$

end.

The auxiliary functions $\text{householder}(\cdot)$ and $\text{householder_product}(\cdot)$ are defined as follows.

**function** $[\phi, v] = \text{householder}(z)$$^2$

{ The function $\text{householder}(\cdot)$ computes $\phi$ and $v$ with $\|v\|_2 = \sqrt{2}$ as described in [25, Chapter 19]. The Householder reflector is then formed as $V = I - vv^T$ with property that $V z = \phi e_1$.}

**function** $V^T = \text{householder_product}(v_1, \ldots, v_n)$

{ The function $\text{householder_product}(\cdot)$ computes a matrix $V^T$ as a product of $n$ Householder reflectors, where $V^T = V_n \cdots V_1$, $V_k$ for $k = 1, \ldots, n$ are defined in relation (2.3), and $V_n = I - v_n v_n^T$. Accumulation of the Householder reflectors is done by a block algorithm implemented in the LAPACK routine $\text{sortbr}(\cdot)$ [1].}

In [3], the following theorem about the numerical stability of Algorithm 2.1 was proved. Here we denote the unit roundoff with $\varepsilon$.

**Theorem 2.1.** If $\hat{B}$ is the bidiagonal matrix computed by Algorithm 2.1, then there exist an orthogonal $(m+n) \times (m+n)$ matrix $\hat{P}$, an orthogonal $n \times n$ matrix $\hat{V}$, $\hat{B} = \hat{P} \hat{V}^T$.

---

$^1$FORTRAN routines for the SVD using the one-sided bidiagonalization methods described in this paper are available from the first author.
and backward perturbations $\Delta A, \delta A$ such that

\begin{equation}
\begin{bmatrix}
\tilde{B} \\
0
\end{bmatrix} = \mathcal{P}^T \begin{bmatrix}
\Delta A \\
A + \delta A
\end{bmatrix} \hat{V}, \quad \left\| \begin{bmatrix}
\Delta A \\
\delta A
\end{bmatrix} \right\|_F \leq \xi \| A \|_F,
\end{equation}

where $\xi = O(mn + n^3)\varepsilon$. The computed approximation $\hat{V}$ of the matrix $V$ satisfies $\left\| \hat{V} - V \right\|_F \leq O(n^2)\varepsilon$. Further, there exist a left orthogonal matrix $\hat{U}$ and a perturbation $\delta A$ such that

\begin{equation}
A + \delta A = \hat{U} \hat{B} \hat{V}^T, \quad \| \delta A \|_F \leq \sqrt{2} \xi \| A \|_F.
\end{equation}

Although this result implies that Algorithm 2.1 is numerically stable for computing $B$ and $V$, we cannot guarantee that the computed matrix $U$ is numerically orthogonal. In many circumstances, the possible loss of orthogonality is irrelevant \cite{8}. One can also get nearly orthogonal bases for the parts associated with the leading singular values of $A$ \cite[Theorem 3.19 and Corollary 3.20]{3}.

Another important characteristic of an algorithm is its efficiency, and the preferable way to evaluate efficiency is through execution time. The execution time of a numerical algorithm depends on two properties: the floating point operation count and the time spent on communication among hierarchically organized computer memory. We are concerned about the efficiency of full SVD algorithms that include bidiagonalization, when computing all of the SVD factors: $P, Q$, and $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_n)$ such that $A = P \Sigma Q^T$.

Extensive numerical tests were performed to test the efficiency of the SVD algorithms. Even though an SVD routine with Algorithm 2.1 requires fewer floating point operations (see \cite{3}), a straightforward implementation of Algorithm 2.1 is often slower than the current LAPACK \cite{1} routine in \texttt{sgesvd()}, since the LAPACK routine optimizes cache memory usage, whereas Algorithm 2.1 does not. In order to decrease cache communication time, we develop a block version of Algorithm 2.1 next.

3. Block version of the new stable one-sided bidiagonalization. The new block version of Algorithm 2.1 improves the usage of fast cache memory by performing as many operations as possible on the data that are currently stored in the cache. In order to do that, one has to transform the original algorithm. The first modification of the algorithm is that transformations by Householder reflectors are aggregated, where the WY representation is used for a product of Householder reflectors \cite{4}. In the WY representation the product of $j$ Householder reflectors is presented as $V_j \cdots V_1 = I - WY^T$, where $W$ and $Y$ have $j$ columns. This means that the matrix $A$ is updated after every $b$ steps, where $m \times b$ is the block dimension. Most of the operations in Algorithm 2.1 are matrix-vector operations, coded as BLAS 2 operations \cite{13}. Memory hierarchy is utilized more efficiently if such algorithms are written in terms of matrix-matrix operations, coded as BLAS 3 operations \cite{14}, \cite{15}, or grouped matrix-vector operations, called BLAS 2.5 operations \cite{11}, \cite{26}. Employing the WY representation of products of Householder transformations results in more BLAS 3 operations; using the BLAS 2.5 approach of Howell et al. \cite{26} leads to further improvement. Operations on the same data, but performed in different places in Algorithm 2.1, are now performed simultaneously. These operations are

\begin{align}
x & \leftarrow x + A^T y \\
w & \leftarrow Ax \quad \text{or} \quad A & \leftarrow A + uv^T \\
x & \leftarrow A^T y \\
w & \leftarrow Ax
\end{align}
Now we discuss our modifications of Algorithm 2.1. As an input to the algorithm we will take the matrix $A \in \mathbb{R}^{m \times n}$ and partition it into block columns. Let $n = b \cdot g + r$, $r \leq b + 1$, where $b$ is a given block column dimension and $g = \lfloor (n - 2)/b \rfloor$ is the number of blocks of dimension $m \times b$. We choose the last two columns to be outside of the block partition, because the last two steps of the one-sided bidiagonalization (corresponding to the last two columns) do not involve computation of a Householder reflector. The $g$ blocks will be updated by means of aggregated Householder transformations and BLAS 2.5 transformations related to the first group of transformations in (3.1). The remaining $r = n - b \cdot g$ columns will be updated with nonaggregated Householder transformations and the second group of BLAS 2.5 transformations in (3.1). As each block consists of $b$ columns, the steps of the algorithm will be organized in two loops: the outer loop going through $g$ blocks and the inner loop going through $b$ columns of the block. Thus we will denote by $A_{j,k}$ the matrix $A$ after the first $j - 1$ blocks, and the first $k$ columns in the $j$th block have been updated.

The main difference between the stable bidiagonalization and its block version is the way Householder reflectors are computed and applied to the matrix $A$. In the $k$th step of Algorithm 2.1, columns $k + 1$ through $n$ of the matrix $A$ are updated with the Householder reflector $V_k$. After this step, the $(k+1)$th column is not changed anymore and is consequently equal to the $(k+1)$th column of the matrix $F$ defined in (2.2).

In the block version of the stable bidiagonalization, updates with Householder reflectors are done blockwise. This means that, only when all of the columns in one block are updated and assigned to $F$ (they will not be modified in the next steps), the rest of the matrix will then be updated with $b$ Householder reflectors in aggregated form that correspond to $b$ steps of Algorithm 2.1. Until then, only the current column is updated. Let us assume that we have computed the first $(j-1)$ blocks of the matrix $F$ obtaining the matrix $A_{j,0}$ and that we are observing the operations in the $j$th step of the outer loop. Then for $k = 2, \ldots, b$ only the $((j-1)b+k)$th column is updated by Householder reflectors from the steps $1, \ldots, k-1$ of the same block, obtaining the matrix $A_{j,k}$, and a new Householder reflector $V_{(j-1)b+k}$ is computed. $V_{(j-1)b+k}$ will affect the columns $((j-1)b+k+1)$ through $n$, but no updates are done. The matrix $A_{j,1}$ is equal to $A_{j,0}$ because the $((j-1)b+1)$th column is already updated; only the Householder reflector $V_{(j-1)b+1}$ is computed. We use the WY form for a product of Householder reflectors described in [4] to write

$$
(3.2) \quad V_{(j-1)b+1} \ldots V_{(j-1)b+k-1} = I - Y_j(:, 1: k-1)W_j(:, 1: k-1)^T.
$$

After the $(jb)$th column has been updated, the columns $jb+1$ through $n$ are updated with the product $V_{(j-1)b+1}, \ldots, V_{jb}$ in WY form (3.2). This process is illustrated in Figure 3.1 with notation defined in (3.3). The $(g+1)$th block is updated with Householder reflectors in the usual way, as it is done in Algorithm 2.1. This is the same approach as in the LAPACK routine $\text{sgebd}()$ [16], where the routine $\text{slabrd}()$ is called first, followed by the routine $\text{sgebd2}()$. $\text{slabrd}()$ performs the two-sided aggregated Householder transformation over the first $g$ blocks, and $\text{sgebd2}()$ performs the unblocked transformations. The differences are that in the block version of Algorithm 2.1 only one-sided Householder transformations are performed and that the dimension of the block is computed differently.

Aggregated Householder transformations represent only one modification of Algorithm 2.1. The other modification is achieved by using the ideas described in [26].
Let us define the following correspondence:

$$\ell = (j - 1)b + k,$$

current $\ell$th column is the $k$th column in the $j$th block,

(3.3) \quad \ell \leftrightarrow (j, k) \quad \text{the indices with } \ell \text{ are replaced by } (j, k).

This correspondence is introduced only for notational convenience. Now we will investigate the lines (4), (5), and (7) in Algorithm 2.1, but with the index $k$ replaced by $\ell$. In all of these statements the vector $z_\ell \rightarrow z_{j,k}$ is directly or indirectly used. In line (4) $u_\ell$ is multiplied by $A_{j,k-1}(\ell + 1: n)^T \rightarrow A_{j,k-1}(\ell + 1: n)^T$ in order to obtain $z_{j,k}$. On the other hand, in line (7) the vector $v_{\ell} \rightarrow v_{j,k}$ is multiplied by $A_{j,k-1}(\ell + 1: n)$, and $v_{j,k}$ is realized from $z_{j,k}$ through line (5) and the function $\text{householder}(\cdot)$. From the definition of $\text{householder}(\cdot)$, we have

$$z_{j,k} = A_{j,k-1}(\ell + 1: n)^T u_\ell,$$

$$v_{j,k} = \frac{\sqrt{2}(z_{j,k} - \phi_{\ell+1}e_1)}{\|z_{j,k} - \phi_{\ell+1}e_1\|_2}, \quad \text{and thus}$$

$$A_{j,k-1}(\ell + 1: n)v_{j,k} = \frac{\sqrt{2}[A_{j,k-1}(\ell + 1: n)z_{j,k} - \phi_{\ell+1}A_{j,k-1}(\ell + 1: n)]}{\|z_{j,k} - \phi_{\ell+1}e_1\|_2}.$$

From the previous observations concerning the update of the matrix $A_{j,0}$ with Householder reflectors, in the $\ell$th step (which in the block version will correspond to the $j$th step of the outer loop and the $k$th step of the inner loop) columns $\ell + 1, \ldots, n$ are not yet updated. Since $A_{j,k-1} = A_{j,0}V_{j,1} \ldots V_{j,k-1}$, (3.2) and (3.4) imply that

$$z_{j,k} = A_{j,0}(\ell + 1, n)^T u_\ell$$

$$-W_{j}(\ell + 1: n, 1: k - 1)Y_{j}(\ell + 1: k - 1)A_{j,0}^T u_\ell,$$

(3.5)
\begin{align}
A_{j,k-1}(\ell, \ell + 1: n)v_{j,k} &= A_{j,0}(\ell, \ell + 1: n)v_{j,k} \\
&\quad - A_{j,0}Y_j(1: k - 1)W_j(\ell + 1: n, 1: k - 1)^T v_{j,k} \\
&= \sqrt{2}[A_{j,0}(\ell, \ell + 1: n)z_{j,k} - \phi_{\ell+1}A_{j,0}(\ell, \ell + 1)]
\end{align}

(3.6)

If we define

\[ z_{j,k}^{(1)} = -W_j(\ell + 1: n, 1: k - 1)Y_j(1: k - 1)^T A_{j,0}^T u_{\ell} \]

as the first phase in the computation of \( z_{j,k} \)

\[ x_{j,k}^{(1)} = A_{j,0}(\ell, \ell + 1: n)z_{j,k} \]

as the first phase in the computation of the vector \( x_{j,k}^{(1)} = A_{j,k-1}(\ell, \ell + 1: n)v_{j,k} \), then

| \( z_{j,k} \) | = | \( z_{j,k}^{(1)} + A_{j,0}(\ell, \ell + 1: n)^T u_{\ell} \) | from (3.5) \\
| \( x_{j,k}^{(1)} \) | = | \( A_{j,0}(\ell, \ell + 1: n)z_{j,k} \) | from (3.6)

will be computed simultaneously, and they comprise the first group of BLAS 2.5 transformations in (3.1). By simultaneous computation we mean that as soon as one component of \( z_{j,k} \) is computed, \( x_{j,k}^{(1)} \) is updated with this new data by the BLAS 1 \texttt{saxpy()} operation. The components of \( z_{j,k} \) can be partitioned in blocks of dimension \( c \), so that BLAS 2 \texttt{segmv()} is used in the simultaneous computation instead of BLAS 1 operations. This improves the cache memory usage even more.

In the \( k \)th step of the inner loop for the last \((g+1)\)th block update with \( V_{g+1,k-1} \), computation of \( z_{g+1,k} \) and \( x_{g+1,k}^{(1)} \) will be done simultaneously. Again let \( \ell = gb + k \).

First, we have

\begin{align}
A_{g+1,k-1}(\ell, \ell + 1: n) &= A_{g+1,k-2}(\ell, \ell + 1: n) \\
&\quad - A_{g+1,k-2}(\ell, n) v_{g+1,k-1} v_{g+1,k-1}(2: n - \ell + 1)^T \\
&= A_{g+1,k-2}(\ell, \ell + 1: n) - x_{g+1,k-1}(2: n - \ell + 1)^T,
\end{align}

where \( x_{g+1,k}^{(3)} = A_{g+1,k-1}(\ell, \ell + 1: n) v_{g+1,k} \), and from (3.4) it follows that

\begin{align}
A_{g+1,k-1}(\ell, \ell + 1: n) &= \sqrt{2}[A_{g+1,k-1}(\ell, \ell + 1: n) z_{g+1,k} - \phi_{\ell+1}A_{g+1,k-1}(\ell, \ell + 1)]
\end{align}

(3.7)

Again, if we define

\[ x_{g+1,k}^{(1)} = A_{g+1,k-1}(\ell, \ell + 1: n) z_{g+1,k} \]

as the first phase in the computation of the vector \( x_{g+1,k}^{(3)} \), then

\begin{align*}
A_{g+1,k-1}(\ell, \ell + 1: n) &= A_{g+1,k-2}(\ell, \ell + 1: n) \\
&\quad - x_{g+1,k-1}^{(3)} v_{g+1,k-1}(2: n - \ell + 1)^T \\
z_{g+1,k} &= A_{g+1,k-1}(\ell, \ell + 1: n)^T u_{\ell} \\
x_{g+1,k}^{(1)} &= A_{g+1,k-1}(\ell, \ell + 1: n) z_{g+1,k}
\end{align*}

comprises the second group of BLAS 2.5 transformations in (3.1).
The reason why these operations are performed simultaneously is that the same parts of the matrix $A$ are involved, as well as the same parts of the vector $z_{j,k}$. Thus, when a particular block of the matrix and the vector is stored in the fast cache memory, all of the operations can be done without transferring blocks from slower memory to the cache, saving some of the time spent on memory by Algorithm 2.1.

Before stating a detailed algorithm, we have to introduce one more definition. The update of the $\ell$th column of the matrix $A_{j,k}$, where $\ell = (j-1)b + k$, is done by the following relations:

$$A_{j,k}(::, \ell) = [A_{j,0}V_{j,1} \ldots V_{j,k-1}](::, \ell)$$

$$= A_{j,0}(::, \ell) - A_{j,0}Y_{j}(::, 1: k-1)W_{j}(\ell, 1: k-1)^T.$$ 

The term $A_{j,0}Y_{j}(::, 1: k-1)$ also occurs in relations (3.5) and (3.6); hence, we define $X_{j} = A_{j,0}Y_{j}$. From the definition of the matrices $Y_{j}$ and $W_{j}$ in [4], $W_{j}$ and $X_{j}$ satisfy the following recurrences:

$$W_{j}(::, 1) = v_{j,1},$$
$$W_{j}(::, 1: k) = [W_{j}(::, 1: k-1), v_{j,k}],$$

$$X_{j}(::, 1) = A_{j,0}Y_{j}(::, 1) = A_{j,0}v_{j,1},$$
$$X_{j}(::, 1: k) = A_{j,0}Y_{j}(::, 1: k)$$
$$= [X_{j}(::, 1: k-1), A_{j,0}v_{j,k} - X_{j}(::, 1: k-1)W_{j}(::, 1: k-1)^T v_{j,k}].$$

Now we can state the complete algorithm.

**Algorithm 3.1.** For $A \in \mathbb{R}^{m \times n}$, rank$(A) = n > 2$, this algorithm computes a left orthogonal $U$, a bidiagonal $B$, and an orthogonal $V$ such that $A = UBV^T$.

**Initialize:**
- the block dimension for aggregated Householder transformations $b$;
- the block dimension for BLAS 2.5 transformations $c$;
- $A_{1,0} = A$;
- $s_{1} = A_{1,0}(::, 1)$;
- $g = \left\lfloor (n-2)/b \right\rfloor$;

for $j = 1 : g$

\{ Update the $j$th block of the matrix $A$ with aggregated Householder transformations and the first group of BLAS 2.5 transformations from (3.1). \}

$X_{j} = 0_{m \times b}$; $W_{j} = 0_{n \times b}$;
for $k = 1 : b$

$\ell = (j-1)b + k$;

$A_{j,k}(::, 1: \ell - 1) = A_{j,k-1}(::, 1: \ell - 1)$;

if $k > 1$

$A_{j,k}(::, \ell) = A_{j,0}(::, \ell) - X_{j}(::, 1: k-1)W_{j}(\ell, 1: k-1)^T$;

$s_{k} = A_{j,k}(::, \ell) - \phi_{\ell} u_{\ell - 1}$;

else

$A_{j,k}(::, \ell) = A_{j,k-1}(::, \ell)$;

end;

$\psi_{\ell} = \|s_{\ell}\|_{2}$;

$u_{\ell} = s_{\ell}/\psi_{\ell}$;

if $k > 1$

$z_{j,k}^{(1)} = -W_{j}(\ell + 1: n, 1: k-1)X_{j}(::, 1: k-1)^Tu_{\ell}$;
\( \begin{align*}
\text{else} & \quad z_{j,k}^{(1)} = 0 \times 1 \\
\text{end}; & \quad x_{j,k}^{(1)} = 0_{m \times 1}; \\
\text{for } i = \ell + 1: c: n & \quad d = \min(c, n - i + 1); \\
 & \quad z_j,k(i - \ell: i - \ell + d - 1) = z_j,k^{(1)}(i - \ell: i - \ell + d - 1) + A_{j,0}(; , i: i + d - 1)T u_{\ell}; \\
 & \quad x_{j,k}^{(1)} = x_{j,k}^{(1)} + A_{j,0}(; , i: i + d - 1)z_j,k(i - \ell: i - \ell + d - 1); \\
\text{end}; & \quad [\phi_{\ell+1}, v_{j,k}, x_{j,k}^{(3)}] = \text{householder} \, 2(z_j,k, x_{j,k}^{(1)}, A_{j,0}(; , \ell + 1)); \\
 & \quad W_j(\ell + 1: n, k) = v_{j,k}; \\
 & \quad x_{j,k}^{(4)} = x_{j,k}^{(3)} - X_j(; : 1: k - 1)W_j(\ell + 1: n, k - 1)T v_{j,k}; \\
 & \quad X_j(; : k) = x_{j,k}^{(4)}; \\
\text{end};
\end{align*} \)

\[ \{ \text{Update the rest of the matrix } A \text{ with aggregated Householder transformations from the } jth \text{ block.} \} \]
\[ A_{j+1,0}(; : 1: jb) = A_{j,b}(; : 1: jb); \]
\[ A_{j+1,0}(; : jb + 1: n) = A_{j,b}(; : jb + 1: n) - X_jW_j(jb + 1: n, : )T; \]
\[ s_{jb+1} = A_{j+1,0}(; : jb + 1) - \phi_{jb+1}u_{jb}; \]
\[ \text{end}; \]
\[ r = n - gb; \]

\[ \{ \text{Update the last block of the matrix } A \text{ with the second group of BLAS 2.5 transformations from (3.1).} \} \]
\[ \text{for } k = 1: r - 1 \]
\[ \ell = gb + k; \]
\[ \text{if } k > 1 \]
\[ A_{g+1,k}(; : 1: \ell - 1) = A_{g+1,k-1}(; : 1: \ell - 1); \]
\[ A_{g+1,k}(; : \ell) = A_{g+1,k-1}(; : \ell) - v_{g+1,k-1}(1)x_{g+1,k-1}^{(3)}; \]
\[ s_k = A_{g+1,k}(; : \ell) - \phi_ku_{\ell-1}; \]
\[ \text{else} \]
\[ A_{g+1,k}(; : 1: \ell) = A_{g+1,k-1}(; : 1: \ell); \]
\[ \text{end}; \]
\[ \psi_\ell = \|s_\ell\|_2; \]
\[ u_\ell = s_k/\psi_\ell; \]
\[ x_{g+1,k}^{(1)} = 0_{m \times 1}; \]
\[ \text{for } i = \ell + 1: n \]
\[ \text{if } k > 1 \]
\[ A_{g+1,k-1}(; , i) = A_{g+1,k-2}(; , i) - v_{g+1,k-1}(i - \ell + 1)x_{g+1,k-1}^{(3)}; \]
\[ \text{end}; \]
\[ \text{if } \ell < n - 1 \]
\[ z_{g+1,k}(i - \ell) = A_{g+1,k-1}(; , i)T u_\ell; \]
\[ x_{g+1,k}^{(1)} = x_{g+1,k}^{(1)} + z_{g+1,k}(i - \ell)A_{g+1,k-1}(; , i); \]
\[ \text{end}; \]
end:

if \( \ell < n - 1 \)

\[
\begin{bmatrix} \phi_{\ell+1}, v_{g+1,k}, x_{g+1,k}^{(3)} \end{bmatrix} = \text{householder} 2(z_{g+1,k}, x_{g+1,k}^{(1)}, A_{g+1,k-1}(\cdot, \ell + 1));
\]

end;

end:

\( \phi_n = u_{n-1}^T A_{g+1,r-1}(\cdot, n); \)

\( s_n = A_{g+1,r-1}(\cdot, n) - \phi_n u_{n-1}; \)

\( \psi_n = \|s_n\|_2; \)

\( u_n = s_n/\psi_n; \)

\( V^T = \text{householder}_\text{product}(v_{1,1}, \ldots, v_{g+1,r-2}); \)

end.

The auxiliary function \text{householder} 2() is defined as follows.

\textbf{function} \( [\phi, v, y]=\text{householder} 2(z, x, b) \)

\{ \text{The function \text{householder} 2() computes } \phi, v, \text{ and } y, \text{ where } \phi \text{ and } v \text{ are computed by the function \text{householder}()} \text{ in the same way as in Algorithm 2.1. The Householder reflector is formed as } V = I - vv^T \text{ with the property that } Vz = \phi e_1 \text{ and } y = Bv, \text{ where } x = Bz \text{ and } b = Be_1. \}

\( [\phi, v]=\text{householder}(z); \)

if \( \phi > 0 \)

\( w = x - \phi b; \)

\( y = \sqrt{2}w/\|z - \phi e_1\|_2; \)

else

\( y = 0; \)

end

\textbf{Remark 3.1.} The choice of block dimensions \( b \) and \( c \) depends upon which computer executes Algorithm 3.1. Their sizes are chosen to obtain optimal efficiency. In LAPACK routines, the function \text{ilaenv()} is used to determine the optimal block size for block algorithms. Section 5.2 in [1] explains how \text{ilaenv()} works: “The version of \text{ilaenv()} supplied with the package contains default values that led to good behavior over a reasonable number of the test machines, but to achieve optimal performance, it may be beneficial to tune \text{ilaenv()} for the particular machine environment.” Our optimal block dimensions were obtained through tests.

Algorithm 2.1 is numerically backward stable for computing \( B \) and \( V \), but what about Algorithm 3.1? The answer to this question is given by Theorem 3.4. Before stating a proof of Theorem 3.4, we will need the results of three technical lemmas. The lemmas are based on the numerical analysis of basic numerical algorithms given by Higham [25] and the analysis of the modified Gram–Schmidt algorithm given by Björck and Paige [5]. The proofs of the lemmas can be found in [7]. In our numerical analysis we will use the following notation: Tildes (\( \tilde{\cdot} \)) will mark computed quantities, and hats (\( \hat{\cdot}, \grave{\cdot} \)) will denote vectors and matrices that correspond to certain exact relations and exist only as theoretical entities, not actually computed.

\textbf{Lemma 3.1.} When Algorithm 3.1 is executed in finite precision arithmetic with the unit roundoff error \( \varepsilon \), then the computed values are of the following form:

\[
\tilde{v}_{j,k} = \hat{v}_{j,k} + \delta\hat{v}_{j,k}, \quad \|\delta\hat{v}_{j,k}\|_2 \leq O(n-l)\varepsilon,
\]

\[
\tilde{W}_j(:, :1:k) = \hat{W}_j(:, :1:k) + \delta\hat{W}_j(:, :1:k), \quad \|\delta\hat{W}_j(:, :1:k)\|_F \leq O(\sqrt{k})\varepsilon,
\]

\[
\tilde{X}_j(:, k) = \hat{A}_{j,0}\hat{Y}_j(:, k) + \delta\hat{X}_j(:, k), \quad \|\delta\hat{X}_j(:, k)\|_2 \leq O(kn)\varepsilon\|\hat{A}_{j,0}\|_F,
\]

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.
where \( \tilde{v}_{j,k} \) define exact Householder reflectors \( \tilde{V}_{j,k} \), and \( \tilde{W}_j, \tilde{Y}_j, \) and \( \tilde{X}_j = \tilde{A}_{j,0}\tilde{Y}_j \) are the exact matrices that are related to Householder vectors \( \tilde{v}_{j,k} \) as described in (3.8). Further, the exact values can be bounded by

\[
\|\tilde{W}_j(:, 1: k)\|_F \leq \sqrt{2}\sqrt{k},
\]

\[
\|\tilde{X}_j(:, 1: k)\|_F = \|\tilde{A}_{j,0}\tilde{Y}_j(:, 1: k)\|_F \leq 2\sqrt{2}\sqrt{k}\|\tilde{A}_{j,0}\|_F.
\]

**Lemma 3.2.** The computed elements of the matrix \( \tilde{B} \) from Algorithm 3.1 satisfy the following relations:

\[
\begin{bmatrix}
\hat{\phi}_{k+1}e_k + \hat{\psi}_{k+1}e_{k+1} \\
0
\end{bmatrix} = \hat{P}_{k+1}\hat{P}_k \begin{bmatrix}
0 \\
\Delta f_{k+1}
\end{bmatrix} + \begin{bmatrix}
\Delta f_{k+1} \\
\delta f_{k+1}
\end{bmatrix},
\]

\[
\left\| \begin{bmatrix}
\Delta f_{k+1} \\
\delta f_{k+1}
\end{bmatrix} \right\|_2 \leq O(bm)\varepsilon\|F\|_F,
\]

where \( \hat{P}_k, k = 1, \ldots, n \), are exact \((m + n) \times (m + n)\) Householder reflectors defined in [5].

**Lemma 3.3.** The computed vector \( \tilde{u}_\ell \) and the computed submatrix \( \tilde{A}_{g+1,r-1}(\cdot, \ell + 2: n) \) from Algorithm 3.1 satisfy the following relation:

\[
\|\tilde{u}_\ell^T \tilde{A}_{g+1,r-1}(\cdot, \ell + 2: n)\|_2 \leq O(b^2n + bm + bn^2)\varepsilon\|A\|_F.
\]

**Theorem 3.4.** If \( \tilde{B} \) is the bidiagonal matrix computed by Algorithm 3.1, then there exist an orthogonal \((m + n) \times (m + n)\) matrix \( \tilde{P} \), an orthogonal \(n \times n\) matrix \( \tilde{V} \), and backward perturbations \( \Delta A, \delta A \) such that

(3.9) \[
\begin{bmatrix}
\tilde{B} \\
0
\end{bmatrix} = \tilde{P}^T \begin{bmatrix}
\Delta A \\
A + \delta A
\end{bmatrix} \tilde{V}, \quad \left\| \begin{bmatrix}
\Delta A \\
\delta A
\end{bmatrix} \right\|_F \leq \xi\|A\|_F,
\]

where \( 0 \leq \xi \leq O(b(mn + n^3))\varepsilon \). The computed approximation \( \tilde{V} \) of the matrix \( \hat{V} \) satisfies \( \|\tilde{V} - \hat{V}\|_F \leq O(n^2)\varepsilon \). Further, there exist a left orthogonal \( \tilde{U} \) and a perturbation \( \delta A \) such that

(3.10) \[
A + \delta A = \tilde{U} \tilde{B} \tilde{V}^T, \quad \|\delta A\|_F \leq \sqrt{2}\xi\|A\|_F.
\]

**Proof.** The proof is rather technical and we will divide it into three steps.

**Step 1.** We will set \( F = \text{fl}(AV) = \tilde{A}_{g+1,r-1} \), \( \ell = (j - 1)b + k \) and \( r = n - gb \), where \( \tilde{A}_{g+1,r-1} \) is the result of Algorithm 3.1 performed in finite precision arithmetic. Thus, in floating point computation we can use \( f_\ell = F(:, \ell) \) instead of

\[
f_\ell = \begin{cases}
\tilde{A}_{j,k}(\cdot, \ell) & \text{for } j = 1, \ldots, g, \ k = 1, \ldots, b, \ \ell = 1, \ldots, gb, \\
\tilde{A}_{g+1,k}(\cdot, \ell) & \text{for } k = 1, \ldots, r - 1, \ \ell = gb + 1, \ldots, n - 1, \\
\tilde{A}_{g+1,r-1}(\cdot, n) & \text{for } \ell = n,
\end{cases}
\]

because the denoted column will not be modified in successive steps of the algorithm (see Figure 3.1).

In this step of the proof we will analyze the application of Householder reflectors to the matrix \( A \), in floating point arithmetic. This application is divided into \( g \) steps, where \( b \) columns of \( F \) are computed in each step, and \( r \) remaining steps, where only one column of \( F \) is computed per step. First, we are investigating the computations performed in one block \( j \in \{1, 2, \ldots, g\} \).
Lemma 3.1 gives the error estimate

$$\tilde{X}_j(\cdot, 1: k) = \hat{A}_{j,0} \tilde{Y}_j(\cdot, 1: k) + \delta \hat{X}_j(\cdot, 1: k),$$

where

$$\|\delta \hat{X}_j(\cdot, 1: k)\|_F = \sqrt{\sum_{i=1}^{k} \|\delta \hat{X}_j(\cdot, i)\|_2^2} \leq O(k^2 n)\epsilon \|\hat{A}_{j,0}\|_F,$$

and

$$\tilde{W}_j(\cdot, 1: k) = \hat{W}_j(\cdot, 1: k) + \delta \hat{W}_j(\cdot, 1: k),$$

with

$$\|\delta \hat{W}_j(\cdot, 1: k)\|_F \leq O(\sqrt{kn})\epsilon.$$

The only thing that remains is to find a bound on the error introduced by the application of Householder reflectors to the matrix $A$. First, for $\ell = (j - 1)b + k$, $k = 1, \ldots, b$, we define the matrices $\tilde{V}_{j,k}, \hat{V}_{j,k} \in \mathbb{R}^{(n-l) \times (n-l)}$ as (see [7] for the proof of Lemma 3.1 and [1]) for the documentation of the LAPACK routine \texttt{slarfg()})

$$\hat{V}_{j,k} = I - \hat{\tau}_{j,k} \hat{v}_{j,k} \hat{v}_{j,k}^T, \quad \tilde{V}_{j,k} = I - \tilde{\tau}_{j,k} \tilde{v}_{j,k} \tilde{v}_{j,k}^T,$$

and $\hat{Q}_j, \tilde{Q}_j \in \mathbb{R}^{n \times n}$ as

$$\hat{Q}_j = \begin{bmatrix} I_{jb} & 0 & \cdots & I_{(j-1)b+2} & 0 \\ 0 & \hat{V}_{j,b} & \cdots & 0 & \hat{V}_{j,1} \end{bmatrix},$$

$$\tilde{Q}_j = \begin{bmatrix} I_{jb} & 0 & \cdots & I_{(j-1)b+2} & 0 \\ 0 & \tilde{V}_{j,b} & \cdots & 0 & \tilde{V}_{j,1} \end{bmatrix},$$

where $\hat{Q}_j = I - \hat{W}_j \hat{Y}_j^T$ and $\tilde{Q}_j = I - \tilde{W}_j \tilde{Y}_j^T$.

Then, for $\tilde{X}_j, \hat{X}_j \in \mathbb{R}^{m \times b}$ and $\tilde{W}_j, \hat{W}_j \in \mathbb{R}^{n \times b}$, from Lemma 3.1 it follows that

$$\hat{A}_{j+1,0} = \text{fl}(\hat{A}_{j,0} \tilde{Q}_j^T) = \text{fl}(\hat{A}_{j,0} - \text{fl}(\tilde{X}_j \hat{W}_j^T))$$

$$= \hat{A}_{j,0} - [(\tilde{X}_j + \delta \hat{X}_j)(\tilde{W}_j^T + \delta \hat{W}_j^T) + \delta_1 A_{j+1,0} + \delta_2 A_{j+1,0}]$$

$$= \hat{A}_{j,0} - \tilde{X}_j \hat{W}_j^T + \delta A_{j+1,0} = \hat{A}_{j,0} - \hat{A}_{j,0} \tilde{Y}_j \hat{W}_j^T + \delta A_{j+1,0}$$

$$= \hat{A}_{j,0} \hat{Q}_j^T + \delta A_{j+1,0},$$

where

$$\|\delta_1 A_{j+1,0}\|_F \leq O(b^2)\epsilon \|\hat{A}_{j,0}\|_F, \quad \|\delta_2 A_{j+1,0}\|_F \leq O(b)\epsilon \|\hat{A}_{j,0}\|_F,$$

which implies

$$\|\delta A_{j+1,0}\|_F \leq O(b^2 n)\epsilon \|\hat{A}_{j,0}\|_F.$$

Finally, we obtain the result for $F = \hat{A}_{g+1,r-1}$, where the first $g$ updates are performed as shown above, and the last $r - 1 = n - gb - 1$ updates can be considered in the
same framework but with $b = 1$. Let us denote $g_t = g + r - 1$ as the total number of update steps. We can note that

$$\|F\|_F \leq \|A_{g+1,r-2}\|_F + O(\varepsilon) \leq \cdots \leq \|A_{g+1,1}\|_F + O(\varepsilon) \leq \|A_{g+1,0}\|_F + O(\varepsilon) \leq \|A\|_F + O(\varepsilon).$$

Then by induction we have

$$F = \left(\cdots ((A\hat{Q}_1^T + \delta A_{2,0}\hat{Q}_2^T + \delta A_{3,0}) \cdots)\hat{Q}_{g_t-1}^T + \delta A_{g+1,r-2}\hat{Q}_{g_t}^T + \delta A_{g+1,r-1} \right) = A\hat{V} + \delta_1 F,$$

where

$$\|\delta_1 F\|_F \leq [O(gb^2n) + O((n - gb)n)\varepsilon]\|A\|_F \leq O(bn^2)\varepsilon\|A\|_F.$$

At the end of this step of the proof, for $\hat{V} = \hat{Q}_1^T \hat{Q}_2^T \cdots \hat{Q}_{g_t}^T$, we can state that

$$F = (A + \delta_1 A)\hat{V}, \quad \|\delta_1 A\|_F \leq \eta_F\|A\|_F, \quad \eta_F \leq O(bn^2)\varepsilon,$$

where $\delta_1 A = \delta_1 F \cdot \hat{V}^T$.

**Step 2.** Since the computation of $\hat{B}$ from $F = [f_1, \ldots, f_n]$ corresponds to the modified Gram-Schmidt algorithm, we can use the results from [5] and represent the computation in an equivalent form, as the Householder QR factorization of the augmented matrix

$$\begin{bmatrix} 0 \\ F \end{bmatrix} = \begin{bmatrix} 0 \\ A + \delta_1 A \end{bmatrix} \hat{V}.$$

By Lemma 3.2, the following relations hold:

$$\begin{bmatrix} \phi_{k+1}e_k + \tilde{\phi}_{k+1}e_{k+1} \\ 0 \end{bmatrix} = \hat{P}_{k+1}\hat{P}_k \begin{bmatrix} 0 \\ f_{k+1} \end{bmatrix} + \begin{bmatrix} \Delta f_{k+1} \\ \delta f_{k+1} \end{bmatrix},$$

$$\left\| \begin{bmatrix} \Delta f_{k+1} \\ \delta f_{k+1} \end{bmatrix} \right\|_2 \leq O(bm)\varepsilon\|F\|_F,$$

where

$$\hat{P}_k = I_{m+n} - \begin{bmatrix} -e_k \\ \hat{u}_k \end{bmatrix} \begin{bmatrix} -e_k^T \\ \hat{u}_k^T \end{bmatrix},$$

and $\hat{u}_k = \hat{s}_k/\|\hat{s}_k\|_2$ is the exact vector with $\|\hat{u}_k\|_2 = 1$. Putting all columns of $\hat{B}$ together, we get

$$\begin{bmatrix} \hat{B} \\ 0 \end{bmatrix} = \begin{bmatrix} \hat{\psi}_1e_1 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\psi}_2e_1 + \tilde{\psi}_2e_2 \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} \phi_ne_{n-1} + \tilde{\phi}_ne_n \\ 0 \end{bmatrix}$$

$$= \hat{P}_1 \begin{bmatrix} \Delta f_1 \\ f_1 + \delta f_1 \end{bmatrix}, \hat{P}_2\hat{P}_1 \begin{bmatrix} \Delta f_2 \\ f_2 + \delta f_2 \end{bmatrix}, \ldots, \hat{P}_n\hat{P}_{n-1} \begin{bmatrix} \Delta f_n \\ f_n + \delta f_n \end{bmatrix},$$
and using the fact that
\[ \tilde{P}_i \begin{bmatrix} \tilde{B}(.; j) \\ 0 \end{bmatrix} = \begin{bmatrix} \tilde{\phi}_j e_{j-1} + \tilde{\psi}_j e_j \\ 0 \end{bmatrix} = \begin{bmatrix} \tilde{B}(.; j) \\ 0 \end{bmatrix} \]
for all \( i \neq j, j - 1 \), we obtain
\[ \begin{bmatrix} \hat{B} \\ 0 \end{bmatrix} = \begin{bmatrix} \hat{P}_n \hat{P}_{n-1} \cdots \hat{P}_2 \hat{P}_1 \begin{bmatrix} \Delta f_1 \\ f_1 + \delta f_1 \end{bmatrix} + \hat{P}_n \hat{P}_{n-1} \cdots \hat{P}_2 \hat{P}_1 \begin{bmatrix} \Delta f_2 \\ f_2 + \delta f_2 \end{bmatrix} \\ \hat{P}_n \hat{P}_{n-1} \cdots \hat{P}_2 \hat{P}_1 \begin{bmatrix} \Delta f_3 \\ f_3 + \delta f_3 \end{bmatrix} + \hat{P}_n \hat{P}_{n-1} \cdots \hat{P}_2 \hat{P}_1 \begin{bmatrix} \Delta f_4 \\ f_4 + \delta f_4 \end{bmatrix} \\ \vdots \end{bmatrix} + \hat{P}_n \hat{P}_{n-1} \hat{P}_{n-2} \begin{bmatrix} \Delta f_{n-1} \\ f_{n-1} + \delta f_{n-1} \end{bmatrix} + \hat{P}_n \hat{P}_{n-1} \begin{bmatrix} \Delta f_n \\ f_n + \delta f_n \end{bmatrix} \]
where, after suitable reordering of the entries in the sums,
\[
\begin{bmatrix}
\Delta F \\
\delta F
\end{bmatrix} = \begin{bmatrix}
\Delta f_1 \\
\delta f_1
\end{bmatrix}, \begin{bmatrix}
\Delta f_2 \\
\delta f_2
\end{bmatrix}, \ldots, \begin{bmatrix}
\Delta f_k \\
\delta f_k
\end{bmatrix}, \ldots, \begin{bmatrix}
\Delta f_{n-2} \\
\delta f_{n-2}
\end{bmatrix}, \begin{bmatrix}
\Delta f_{n-1} \\
\delta f_{n-1}
\end{bmatrix}, \begin{bmatrix}
\Delta f_n \\
\delta f_n
\end{bmatrix} \\
\begin{bmatrix}
\Delta j_f_{j+2} \\
\delta j_f_{j+2}
\end{bmatrix}, \ldots, \begin{bmatrix}
\Delta j_f_m \\
\delta j_f_m
\end{bmatrix}
\end{bmatrix}.
\]

Taking norms, we obtain
\[
\left\| \begin{bmatrix}
\Delta F \\
\delta F
\end{bmatrix} \right\|_F \leq O(bm\sqrt{n})\varepsilon\|F\|_F + \sqrt{2}\sum_{j=1}^{n-2} \left\| \begin{bmatrix}
\Delta f_j \\
\delta f_j
\end{bmatrix} \begin{bmatrix}
f_{j+2} & f_{j+3} & \cdots & f_n
\end{bmatrix} \right\|_2
\leq O(bm\sqrt{n})\varepsilon\|F\|_F + \sqrt{2}\sum_{j=1}^{n-2} \left\| \begin{bmatrix}
\Delta f_j \\
\delta f_j
\end{bmatrix} \begin{bmatrix}
f_{j+2} \\
\cdots \\
f_n
\end{bmatrix} \right\|_2
+ \left\| \delta \tilde{u}_j \right\|_2 \|F\|_F.
\]

It remains to estimate the products \(\tilde{u}_j^T f\) for \(\ell = 3, \ldots, n\) and \(i = 1, \ldots, \ell - 2\), where \(\ell = (j - 1)b + k\), \(k = 1, \ldots, b\). For this estimate, the important role plays the choice of the vector \(z_{j,k}\). From Lemma 3.3 it follows that
\[
\left\| \begin{bmatrix}
\Delta f_j \\
\delta f_j
\end{bmatrix} \begin{bmatrix}
\tilde{u}_j^T \\
A_{g+1,r-1}(\cdot, \ell + 2; n)
\end{bmatrix} \right\|_2 \leq O(b^2n + bm + bn^2)\varepsilon\|A\|_F.
\]

Then
\[
\left\| \begin{bmatrix}
\Delta F \\
\delta F
\end{bmatrix} \right\|_F \leq O(bmn + n^3)\varepsilon\|F\|_F \leq O(bmn + n^3)(1 + \eta_F)\|A\|_F.
\]

To get the relation (3.9), we collect the perturbations from both implicit tridiagonalization and the Gram-Schmidt computation
\[
\begin{bmatrix}
\tilde{B} \\
0
\end{bmatrix} = \hat{P}^T \left\{ \begin{bmatrix}
0 \\
F
\end{bmatrix} + \begin{bmatrix}
\Delta F \\
\delta F
\end{bmatrix} \right\} = \hat{P}^T \left\{ \begin{bmatrix}
0 \\
A + \delta_1 A
\end{bmatrix} \hat{V} + \begin{bmatrix}
\Delta F \\
\delta F
\end{bmatrix} \right\}
= \hat{P}^T \left\{ \begin{bmatrix}
0 \\
A + \delta_1 A
\end{bmatrix} + \begin{bmatrix}
\Delta F \\
\delta F
\end{bmatrix} \hat{V} + \begin{bmatrix}
\Delta F \\
\delta F
\end{bmatrix} \hat{V}^T \hat{V} - \begin{bmatrix}
\Delta F \\
\delta F
\end{bmatrix} \hat{V} + \begin{bmatrix}
\Delta F \\
\delta F
\end{bmatrix} \right\} \hat{V}.
\]

Step 3. Finally, using \(P_{11} = P(1: n, 1: n)\), \(P_{21} = P(n + 1: n + m, 1: n)\), we have
\[
\begin{bmatrix}
\Delta A \\
A + \delta_1 A
\end{bmatrix} \hat{V} = \begin{bmatrix}
P_{11} \\
P_{21}
\end{bmatrix} \hat{B}, \begin{bmatrix}
P_{11}^T P_{11} + P_{21}^T P_{21}
\end{bmatrix} = I,
\]
and (3.10) follows by an application of [5, Lemma 3.1]. The proof that (3.10) holds for the nonblock version of the algorithm is given in Theorem 3.18 [3]. The same arguments can be applied to the block version.

Note that in (3.10) we can write \(A + \delta A = \hat{U} \hat{B} \hat{V}^T (I + \Gamma)\), \(\|\Gamma\|_F \leq O(n^2)\varepsilon\). \(\square\)

In our numerical experiments, the optimal choice for the block dimension \(b\) was usually 16, so the accuracy predicted by the bound in Theorem 3.4 is close to the accuracy predicted by Theorem 2.1.
Example 3.1. Let $A = [a_{ij}]$ be the $n \times n$ Kahan matrix as in [3], with

$$a_{ij} = \begin{cases} \alpha^{i-1} & i = j, \\ -\alpha^{j-1} & i > j, \end{cases}$$

where $\alpha^2 + \beta^2 = 1$ and $\alpha, \beta > 0$. For our tests we chose $\alpha = \sin(1.2)$ and $n = 50, 60, \ldots, 200$. In this case the matrices are ill-conditioned. The first $n - 1$ singular values gradually decay and are bounded away from zero, but, on the other hand, the smallest singular value decays rapidly with $n$.

We compare the accuracy of Algorithm 3.1 with Algorithm 2.1 and Ralha’s one-sided bidiagonalization, by using the Wielandt–Hoffman measure

$$\sqrt{\frac{1}{n} \sum_{k=1}^{n} (\sigma_k(A) - \sigma_k(B))^2} \|A\|_F. \tag{3.11}$$

This example is performed in MATLAB and in double precision. The one-sided bidiagonalization routines are implemented in the same way as the FORTRAN routines in the next section. The singular values $\sigma_k(A)$ of the matrix $A$ are computed by the MATLAB command $\text{svd}(\cdot)$. The results are shown in Figure 3.2.

We can note that Algorithm 3.1 sometimes produces the bidiagonal matrix $B$ with slightly less accurate singular values than Algorithm 2.1. Theorem 3.4 asserts that the bound on (3.11) for Algorithm 3.1 is $b$ times larger than the corresponding bound for Algorithm 2.1, where $b$ is the block dimension. In our case, we took $b = 16$.

If we compare the computed errors measured by (3.11), we can see that the largest difference is obtained for $n = 180$, where the error of Algorithm 3.1 is 1.67 times larger than the error of Algorithm 2.1. In this case, the estimation of the error bounds on
(3.11) from Theorems 2.1 and 3.4 are

<table>
<thead>
<tr>
<th>Algorithm 2.1</th>
<th>Algorithm 3.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>((n^2 + n^3)\varepsilon = 6.51 \cdot 10^{-10})</td>
<td>(b(n^2 + n^3)\varepsilon = 1.04 \cdot 10^{-8})</td>
</tr>
</tbody>
</table>

Hence, our computed results are in accordance with the approximate bounds in Theorems 2.1 and 3.4 and reveal that these bounds can significantly overestimate the actual error.

Similar to the results of Björck and Paige in [5], Corollary 3.18 in [3] states that for the nonblock version of one-sided bidiagonalization algorithm we have

\[
\|\tilde{U}^T \tilde{U} - I\|_F \leq p(m, n)\kappa_2(\tilde{B})\varepsilon + O(\varepsilon^2),
\]

where \(\tilde{U}\) is the computed matrix of left singular vectors, \(p(m, n)\) is a polynomial with modest degree, and \(\kappa_2(\tilde{B}) = \|\tilde{B}\|_2\|\tilde{B}^{-1}\|_2\). In fact, arguments given in the proof of this corollary hold for any bidiagonalization algorithm for which Theorem 3.4 can be proved. That means that if \(B\) is ill-conditioned, \(\tilde{U}\) might be far from left orthogonal matrix, but sometimes the bound in (3.12) is too pessimistic.

On the other hand, suppose that a numerically backward stable SVD is performed on the matrix \(\tilde{B}\) from Theorem 3.4, obtaining matrices \(\tilde{Y}, \tilde{\Sigma}, \text{ and } \tilde{W}\). From [3, Theorem 3.7] it follows that there exists a perturbation \(\delta A\) such that \(A + \delta A = \tilde{P}\tilde{\Sigma}\tilde{Q}^T\), where \(\tilde{P} = U\tilde{Y}, \tilde{Q} = V\tilde{W}\), and \(\|\delta A\|_F \leq O(\varepsilon)\|A\|_F\). \(\tilde{P}\) can be considered as a matrix of computed left singular vectors of \(A\), \(\tilde{Q}\) as a matrix of computed right singular vectors of \(A\), and \(\tilde{\Sigma} = \text{diag}(\tilde{\sigma}_1, \ldots, \tilde{\sigma}_n)\), where \(\tilde{\sigma}_1 \geq \cdots \geq \tilde{\sigma}_n\), is a matrix of computed singular values. By the same argument as in Corollary 3.20 in [3], we can state that

\[
\|\tilde{P}_1^T \tilde{P}_1 - I\|_F \leq q(m, n)\frac{\tilde{\sigma}_1}{\sigma_k}\varepsilon + O(\varepsilon^2),
\]

where \(\tilde{P}_1\) is a matrix which contains \(k\) computed left singular vectors that correspond to the \(k\) largest singular values, and \(q(m, n)\) is a polynomial with modest degree. This shows that the basis for the leading principal subspace of left singular vectors can be computed with near orthogonality.

4. Efficiency of the stable block one-sided bidiagonalization. For the block version of the new stable one-sided bidiagonalization, extensive testing was carried out. Computations were performed in the Advanced Computing Laboratory of the Department of Mathematics, University of Zagreb. The laboratory consists of 20 computers, connected in a local 1Gb network. The specifications of the computers are shown in Table 4.1.

The computers are working under a Debian GNU/Linux operating system. The tests were written in FORTRAN 77 programming language, and a GNU (v0.5.24)
Table 4.2

Average execution times for full SVD algorithms.

<table>
<thead>
<tr>
<th>$m \times n$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_L$</th>
<th>$p_{2,1}$</th>
<th>$p_{2,L}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 $\times$ 100*</td>
<td>0.008</td>
<td>0.008</td>
<td>0.011</td>
<td>0.00%</td>
<td>27.27%</td>
</tr>
<tr>
<td>200 $\times$ 200*</td>
<td>0.058</td>
<td><strong>0.055</strong></td>
<td>0.065</td>
<td>5.17%</td>
<td>15.38%</td>
</tr>
<tr>
<td>500 $\times$ 50*</td>
<td>0.005</td>
<td>0.005</td>
<td>0.008</td>
<td>0.00%</td>
<td>37.50%</td>
</tr>
<tr>
<td>500 $\times$ 100*</td>
<td>0.022</td>
<td><strong>0.021</strong></td>
<td>0.035</td>
<td>4.55%</td>
<td>40.00%</td>
</tr>
<tr>
<td>500 $\times$ 500</td>
<td>4.910</td>
<td>4.360</td>
<td>4.590</td>
<td>11.20%</td>
<td>5.01%</td>
</tr>
<tr>
<td>1000 $\times$ 100*</td>
<td>0.072</td>
<td><strong>0.054</strong></td>
<td>0.096</td>
<td>25.00%</td>
<td>43.75%</td>
</tr>
<tr>
<td>1000 $\times$ 500</td>
<td>6.980</td>
<td>5.810</td>
<td>5.290</td>
<td>16.76%</td>
<td>-9.83%</td>
</tr>
<tr>
<td>1000 $\times$ 500**</td>
<td>5.550</td>
<td><strong>5.030</strong></td>
<td>5.290</td>
<td>9.37%</td>
<td>4.91%</td>
</tr>
<tr>
<td>1000 $\times$ 1000</td>
<td>39.810</td>
<td>35.200</td>
<td>36.730</td>
<td>11.58%</td>
<td>-11.58%</td>
</tr>
<tr>
<td>2000 $\times$ 200</td>
<td>1.860</td>
<td>1.490</td>
<td><strong>0.670</strong></td>
<td>19.89%</td>
<td>-122.39%</td>
</tr>
<tr>
<td>2000 $\times$ 1000</td>
<td>47.590</td>
<td>43.150</td>
<td>44.480</td>
<td>9.33%</td>
<td>2.99%</td>
</tr>
<tr>
<td>2000 $\times$ 2000</td>
<td>320.640</td>
<td>285.960</td>
<td>294.380</td>
<td>10.82%</td>
<td>4.17%</td>
</tr>
<tr>
<td>3000 $\times$ 3000</td>
<td>1284.300</td>
<td><strong>1152.940</strong></td>
<td>1193.340</td>
<td>10.23%</td>
<td>3.39%</td>
</tr>
</tbody>
</table>

compiler with optimization (option "-O") was used to obtain executable files. LAPACK and BLAS routines were called in these programs, and ATLAS (Automatically Tuned Linear Algebra Software) tuned for Athlon processors was used to implement BLAS routines. Tests were done in single precision. Matrices in the tests were generated as products $A = UV^T$, where $\Sigma$ is a diagonal matrix with fixed singular values $\{1, 2, \ldots, n\}$, and $U$ and $V$ are random orthogonal matrices.

The block dimensions $b$ and $c$ in the tests were chosen to obtain the best execution time. In our case it turned out to be $b = 16$, and we took $c = 8$. Table 4.2 gives average execution times for full SVD algorithms, expressed in seconds.

The meaning of the headers in Table 4.2 are as follows:

$t_1$ — the SVD with Algorithm 2.1 for bidiagonalization. The LAPACK routine $\text{sbdsqr()}$ is used for the SVD of a bidiagonal matrix, which implements the bidiagonal QR algorithm.

$t_2$ — the SVD with Algorithm 3.1 for bidiagonalization. The LAPACK routine $\text{sbdsqr()}$ is used for the SVD of a bidiagonal matrix, which implements the bidiagonal QR algorithm.

$t_L$ — the LAPACK $\text{sgesvd()}$ routine.

$p_{2,1} = 100(t_1 - t_2)/t_1$ — the percentage of time decrease, when the SVD with Algorithm 3.1 is compared to the SVD with Algorithm 2.1.

$p_{2,L} = 100(t_L - t_2)/t_L$ — the percentage of time decrease, when the SVD with Algorithm 3.1 is compared to the LAPACK routine.

* — the time was measured for 10 consecutive executions of the routines and then divided by 10.

** — QR factorization is performed before the SVD with Algorithms 2.1 and 3.1 for bidiagonalization.

We can conclude that the block version of the one-sided bidiagonalization algorithm did decrease the execution time of Algorithm 2.1, as expected. Compared to the SVD with Algorithm 2.1 the most significant time decrease is 25.00% for matrix dimensions $1000 \times 100$. The SVD routine with Algorithm 3.1 produces a code that...
is not slower than the LAPACK \texttt{sgesvd()} routine in most cases when all of the SVD factors are required, although this varies with the dimensions of the matrix. In many cases we observed some gains in speed. Here it should be emphasized that the QR factorization was not applied before the one-sided bidiagonalization algorithms except in cases denoted by **, and we can note that the one-sided bidiagonalization algorithms achieved the largest speedup compared to the LAPACK routine when \( m \) is much larger than \( n \). This can be explained by the fact that the SVD with the one-sided bidiagonalization algorithm has a smaller operation count than the LAPACK \texttt{sgesvd()} routine (see [3, Table 1]). On the other hand, for dimensions 1000 \( \times \) 500, 2000 \( \times \) 200, and 2000 \( \times \) 1000 the QR factorization was necessary to make Algorithm 3.1 faster than the \texttt{sgesvd()} routine. It seems that the memory hierarchy was utilized better that way. If the matrix \( U \) is not needed, then the advantage of the one-sided bidiagonalization over the LAPACK routine might be lost. That happens because \( U \) is always computed, whether it is needed or not (see [3, Table 1]). When solving the problems described in [8], our algorithm would be preferable since possible loss of orthogonality of the matrix \( \tilde{U} \) is irrelevant in these cases, and the SVD with the one-sided bidiagonalization algorithm is often faster than the LAPACK routine.

5. Parallel version of the new stable one-sided bidiagonalization. The parallel bidiagonalization algorithm is performed on several processors simultaneously. Each matrix is distributed over the memories of all processors, and this distribution is balanced. This means that the dimensions of the submatrices assigned to each processor are almost the same. It is important to minimize communication between processors, as the time spent for communication can be expected to be a significant part of the overall execution time.

In our case we used the following setting:
- the processors were organized in linear order:
  
  ![Processor Layout](image)

  
  - we used ScaLAPACK [6] for the computation;
  - we used the message passing interface [22] for the interprocessor communication.

The matrix distribution over the processors is performed rowwise, because the algorithm is one-sided and column-oriented.

The most important features of the parallel version of the stable one-sided bidiagonalization algorithm are the following:

1. The matrix layout is one-dimensional block-cyclic row distribution. Each \( m \times n \) matrix is divided in \( m_b \times n \) blocks of contiguous rows, where \( m_b \) is the block row dimension. Then the blocks are distributed across the processors in cyclic order, which guarantees good load balancing (see Figure 5.1 and [6]).

2. The algorithm is performed in the same way as Algorithm 2.1, with extra interprocessor communication. Interprocessor communication is required for:
   - computation of \( z_k \) as matrix–vector multiplication,
   - broadcasting the vector \( z_k \) to all processors,
   - computing scalar products.

The rest of the computations consist of BLAS 1 operations (operations with vectors), as well as computation and application of Householder reflectors, which need no additional communication.
The complete parallel algorithm with explanations is listed in Algorithm 5.1.

**Algorithm 5.1.** For \( A \in \mathbb{R}^{m \times n}, \ \text{rank}(A) = n > 2, \) this algorithm computes in parallel a left orthogonal \( U = [u_1, \ldots, u_n], \) a bidiagonal \( B, \) and an orthogonal \( V \) such that \( A = UBV^T. \)

1. Distribute \( \Psi = [\psi_1 \ldots \psi_n]^T \) over the processors;
2. distribute \( \Phi = [\phi_1 \ldots \phi_{n-1}]^T \) over the processors;
3. \( A_0 = A; \)
4. \( f_1 = A(:,1); \ \psi_1 = \|f_1\|_2; \) parallel dot product;
5. \( u_1 = f_1/\psi_1; \) BLAS 1 operation without communication;

for \( k = 1 : n - 2 \)

6. \( z_k = A_{k-1}(:,k+1:n)^T u_k; \) parallel matrix-vector product; the resulting vector is stored in one processor and then broadcast to all other processors; computation performed on each processor; each processor owns a copy of \( v_k \) and \( \phi_{k+1}; \)
7. \( [\phi_{k+1}, v_k] = \text{householder}(z_k); \)
8. \( A_k(:,1:k) = A_{k-1}(:,1:k); \)
9. \( A_k(:,k+1:n) = A_{k-1}(:,k+1:n) - A_{k-1}(:,k+1:n)v_kv_k^T; \) parallel update without communication;
10. \( f_{k+1} = A_k(:,k+1); \)
11. \( s_{k+1} = f_{k+1} - \phi_{k+1}u_k; \) BLAS 1 operation without communication;
12. \( \psi_{k+1} = \|s_{k+1}\|_2; \) parallel dot product;
13. \( u_{k+1} = s_{k+1}/\psi_{k+1}; \) BLAS 1 operation without communication;

end;
14. \( f_n = A_{n-2}(:,n); \ \phi_n = u_{n-1}^T f_n; \) parallel dot product;
15. \( s_n = f_n - \phi_n u_{n-1}; \) BLAS 1 operation without communication;
16. \( \psi_n = \|s_n\|_2; \) parallel dot product;
17. \( u_n = s_n/\psi_n; \) BLAS 1 operation without communication;
18. \( V^T = \text{householder_product}(v_1, \ldots, v_{n-2}) \) parallel computation;

end.
The parallel version of the stable bidiagonalization algorithm performs the same operations as the serial nonblock version. Preliminary numerical experiments showed that a parallel block version, applying Algorithm 3.1 in parallel, has a large overhead on our computers; thus, it was almost always slower than the ScaLAPACK routine. The results of Theorem 2.1 hold for Algorithm 5.1 as well.


6.1. Tests performed at the Advanced Computing Laboratory. The tests for the parallel version of the new stable bidiagonalization algorithm were done over a large variety of matrix dimensions. The computations were performed in the computational environment described in section 4 and matrices were generated in the same way. QR factorization was not performed before bidiagonalization, because Algorithm 5.1 is suitable for the parallel computing in its original form. The QR factorization would just increase the interprocessor communication. This statement even for the ScaLAPACK \texttt{psgesvd()} routine, where in most cases the QR factorization introduced before bidiagonalization caused a slowdown in producing the full SVD. The block cyclic distribution was also used for testing the ScaLAPACK routine, but the linear layout of the processors may not always be optimal for this routine. So in this case we performed our test with all possible layouts for the fixed processor number, and we chose the best execution time. The parallel tests were performed over a variety of block dimensions, where block column and row dimensions were the same. The block dimensions were chosen from the set \{8, 16, 32, 64, 128, 256\}, and the best execution time among block dimensions was displayed in Table 6.1. For the SVD with Algorithm 5.1 the block dimension was equal to 8 in most cases, and for the ScaLAPACK SVD routine it was equal to 32 in most cases. Table 6.1 gives the average execution times expressed in seconds for full SVD algorithms when computed on \(p\) processors. The meaning of the headers in Table 6.1 are as follows:

\[
\begin{align*}
& t_3 \quad \text{— the parallel SVD with Algorithm 5.1.} \\
& p_m \times p_n \quad \text{— processor layout with the best execution time} \\
& t_S \quad \text{— the ScaLAPACK psgesvd()} \text{ routine.} \\
& p_{3,S} = 100(t_S - t_3)/t_S \quad \text{— the percentage of time decrease, when the} \\
& \quad \text{parallel SVD with Algorithm 5.1 is compared to the ScaLAPACK routine.} \\
& \eta_{3,p} = (t_2/t_3)/p \quad \text{— the efficiency of the parallel SVD with} \\
& \quad \text{Algorithm 5.1 on} \ p \ \text{processors.} \\
& \eta_{S,p} = (t_L/t_S)/p \quad \text{— the efficiency of the ScaLAPACK routine on} \ p \\
& \quad \text{processors.}
\end{align*}
\]

As we can see from Table 6.1, we accomplished a considerable decrease in execution time for \(m \times n\) matrices when \(m > n\). In that case the SVD with the described parallel version of the one-sided bidiagonalization algorithm is much faster than the ScaLAPACK routine \texttt{psgesvd()} . Compared to the ScaLAPACK routine, the most significant time decrease is 68.28% for matrix dimensions 5000 \(\times\) 100 and for 8 processors. On the other hand, for squared matrices the ScaLAPACK routine is up to 55.08% faster, and in this case a block version of the parallel one-sided algorithm is required. Efficiency of such block algorithms is described in the next subsection.

Another important feature of parallel algorithms is the efficiency. In an ideal situation an algorithm executed on \(p\) processors should be \(p\) times faster than the same algorithm executed on only one processor. The efficiency measures departure from the ideal execution time. Table 6.1 shows the efficiency for both SVD algorithms applied
to matrices with small dimensions. In the case of larger dimensions we were not able to run the codes on a single processor due to memory limitations, and therefore the efficiency is not computed. We can see that the parallel SVD with Algorithm 5.1 has better efficiency than the ScaLAPACK routine `psgesvd()` in all cases. The new algorithm has also better scalability than the ScaLAPACK routine when the number of processors is increased from 4 to 8 processors, which is illustrated in Figure 6.1. The y axis in Figure 6.1 represents the reduction factor in execution time when the number of processors is doubled and the matrix dimensions are fixed. The labels on the x axis denote matrix dimensions and ratios $p_1/p_2$, which indicate that the number of processors is increased from $p_1$ to $p_2$. Theoretically, the reduction factor in the execution time, when the number of processors is doubled, is bounded by 2. However, in two cases we obtained a reduction factor greater than 2, once for the one-sided bidiagonalization and once for the ScaLAPACK routine. This “superlinear speedup” is usually explained by the fact that, as we reduce the amount of data per processor, a larger percentage of the local data is stored in the caches, thus reducing the memory traffic overheads. So it appears that, in this case, such savings in time are more than enough to pay for the extra time required by the communication involving twice as many processors. We can conclude that, for squared matrices in case the number of processors is increased from 8 to 16 processors, the ScaLAPACK routine becomes much more efficient since it is a blocked algorithm. This illustrates the importance of the block algorithms and gives us a motivation for developing an efficient block parallel version of the one-sided bidiagonalization algorithm.

### 6.2. Tests performed at the High Performance Computing Center North
The same tests as in subsection 6.1 were also performed on the “Sarek” cluster at the High Performance Computing Center North Sweden. The cluster con-

---

Table 6.1

<table>
<thead>
<tr>
<th>$m \times n$</th>
<th>$p$</th>
<th>$t_1$</th>
<th>$p_1 \times p_2$</th>
<th>$t_2$</th>
<th>$p_2 \times S_3$</th>
<th>$p_2 \times S_4$</th>
<th>$p_2 \times S_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1000 \times 100$</td>
<td>4</td>
<td>0.1305</td>
<td>2x2</td>
<td>0.3061</td>
<td>56.56</td>
<td>0.1034</td>
<td>0.0799</td>
</tr>
<tr>
<td>$1000 \times 500$</td>
<td>4</td>
<td>2.2733</td>
<td>2x2</td>
<td>3.1061</td>
<td>26.81</td>
<td>0.5532</td>
<td>0.4258</td>
</tr>
<tr>
<td>$1000 \times 1000$</td>
<td>4</td>
<td>10.1454</td>
<td>2x2</td>
<td>12.3271</td>
<td>17.70</td>
<td>0.8674</td>
<td>0.7449</td>
</tr>
<tr>
<td>$1000 \times 1000$</td>
<td>8</td>
<td>7.4465</td>
<td>8x1</td>
<td>16.0945</td>
<td>53.73</td>
<td>0.5909</td>
<td>0.2853</td>
</tr>
<tr>
<td>$1000 \times 1000$</td>
<td>16</td>
<td>5.2669</td>
<td>4x4</td>
<td>8.5679</td>
<td>38.53</td>
<td>0.4177</td>
<td>0.2679</td>
</tr>
<tr>
<td>$2000 \times 200$</td>
<td>4</td>
<td>0.5734</td>
<td>2x2</td>
<td>0.9863</td>
<td>41.86</td>
<td>0.2572</td>
<td>0.1698</td>
</tr>
<tr>
<td>$2000 \times 1000$</td>
<td>4</td>
<td>14.1258</td>
<td>2x2</td>
<td>15.8176</td>
<td>10.70</td>
<td>0.7637</td>
<td>0.7030</td>
</tr>
<tr>
<td>$2000 \times 1000$</td>
<td>8</td>
<td>9.6341</td>
<td>8x1</td>
<td>18.5550</td>
<td>48.08</td>
<td>0.5599</td>
<td>0.2996</td>
</tr>
<tr>
<td>$2000 \times 1000$</td>
<td>16</td>
<td>6.6574</td>
<td>4x4</td>
<td>11.7242</td>
<td>43.22</td>
<td>0.4051</td>
<td>0.2371</td>
</tr>
<tr>
<td>$2000 \times 2000$</td>
<td>4</td>
<td>92.2496</td>
<td>2x2</td>
<td>96.6973</td>
<td>4.60</td>
<td>0.7750</td>
<td>0.7611</td>
</tr>
<tr>
<td>$2000 \times 2000$</td>
<td>8</td>
<td>41.3098</td>
<td>8x1</td>
<td>63.7957</td>
<td>35.25</td>
<td>0.8653</td>
<td>0.5768</td>
</tr>
<tr>
<td>$2000 \times 2000$</td>
<td>16</td>
<td>35.3350</td>
<td>4x4</td>
<td>37.7961</td>
<td>6.51</td>
<td>0.5058</td>
<td>0.4868</td>
</tr>
<tr>
<td>$4000 \times 200$</td>
<td>8</td>
<td>0.9272</td>
<td>8x1</td>
<td>2.4123</td>
<td>61.56</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$4000 \times 1000$</td>
<td>16</td>
<td>14.0244</td>
<td>8x1</td>
<td>21.7543</td>
<td>35.53</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$4000 \times 1000$</td>
<td>16</td>
<td>9.0173</td>
<td>16x1</td>
<td>16.2978</td>
<td>44.67</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$4000 \times 4000$</td>
<td>8</td>
<td>378.1784</td>
<td>8x1</td>
<td>433.7231</td>
<td>12.81</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$4000 \times 4000$</td>
<td>16</td>
<td>246.8128</td>
<td>2x8</td>
<td>181.0293</td>
<td>-36.34</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$5000 \times 1000$</td>
<td>8</td>
<td>0.4337</td>
<td>8x1</td>
<td>1.3674</td>
<td>68.28</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$5000 \times 1000$</td>
<td>16</td>
<td>10.0963</td>
<td>16x1</td>
<td>17.1380</td>
<td>41.09</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$5000 \times 5000$</td>
<td>16</td>
<td>538.0723</td>
<td>4x4</td>
<td>346.9633</td>
<td>-35.88</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$8000 \times 1000$</td>
<td>16</td>
<td>13.3496</td>
<td>16x1</td>
<td>19.1207</td>
<td>30.18</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$8000 \times 8000$</td>
<td>16</td>
<td>2449.5129</td>
<td>4x4</td>
<td>2077.7632</td>
<td>-17.89</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$10000 \times 1000$</td>
<td>16</td>
<td>15.4075</td>
<td>16x1</td>
<td>19.4387</td>
<td>20.74</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$10000 \times 10000$</td>
<td>16</td>
<td>3422.0708</td>
<td>4x4</td>
<td>2963.5295</td>
<td>-15.47</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
Fig. 6.1. Reduction in execution time in the case when the number of processors is doubled.

Table 6.2
The specifications of the nodes in the “Sarek” cluster.

<table>
<thead>
<tr>
<th>2 processors</th>
<th>AMD Opteron 248</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>2.2 GHz</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>64 kB + 64 kB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1024 kB</td>
</tr>
<tr>
<td>RAM</td>
<td>8 GB</td>
</tr>
</tbody>
</table>

sists of 190 nodes, connected to a Gigabit ethernet communication network and to a Myrinet high-performance network. The specifications of the nodes are shown in Table 6.2. The nodes are working under a Debian GNU/Linux operating system. The test codes were compiled with MPIF77 1.2.5.12 and PGF77 with optimization (option “-fast”). “Goto BLAS” (r0.94) was used to implement BLAS routines. QR factorization was not performed before bidiagonalization. Table 6.3 gives the execution times expressed in seconds for full SVD algorithms when computed on \( p \) processors. Four rounds of tests were performed with four different parallel variants of the stable one-sided bidiagonalization algorithm. In each round the one-sided bidiagonalization routine was executed together with the ScALAPACK \texttt{psgesvd()} routine for comparison. Due to the lack of space in Table 6.3, instead of the execution time of the ScALAPACK routine \( t_S \) we present the ratio \( t_i/t_S \), where \( t_i \) is the execution time of the \( i \)th version of the parallel bidiagonalization algorithm.

The meaning of the headers in Table 6.3 are as follows:
- \( t_S \) — the ScALAPACK \texttt{psgesvd()} routine.
- \( t_3 \) — the parallel SVD with Algorithm 5.1.
- \( t_4 \) — the parallel SVD with a modified one-sided bidiagonalization algorithm. Campos et al. [9] have proposed a modification of the stable one-sided bidiagonalization. This version reduces communications events...
Table 6.3
Average execution times for full parallel SVD algorithms.

<table>
<thead>
<tr>
<th>$m \times n$</th>
<th>$p$</th>
<th>$t_5$</th>
<th>$t_5/t_S$</th>
<th>$t_4$</th>
<th>$t_5/t_S$</th>
<th>$t_6$</th>
<th>$t_6/t_S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1000 \times 100$</td>
<td>4</td>
<td>0.023</td>
<td>0.88</td>
<td>0.022</td>
<td>0.83</td>
<td>0.023</td>
<td>0.88</td>
</tr>
<tr>
<td>$1000 \times 500$</td>
<td>4</td>
<td>0.530</td>
<td>0.87</td>
<td>0.524</td>
<td>0.86</td>
<td>0.495</td>
<td>0.81</td>
</tr>
<tr>
<td>$1000 \times 1000$</td>
<td>8</td>
<td>1.732</td>
<td>0.83</td>
<td>1.697</td>
<td>0.81</td>
<td>1.669</td>
<td>0.80</td>
</tr>
<tr>
<td>$1000 \times 1000$</td>
<td>16</td>
<td>1.284</td>
<td>0.92</td>
<td>1.185</td>
<td>0.85</td>
<td>1.262</td>
<td>0.91</td>
</tr>
<tr>
<td>$2000 \times 200$</td>
<td>4</td>
<td>0.116</td>
<td>0.89</td>
<td>0.113</td>
<td>0.89</td>
<td>0.108</td>
<td>0.85</td>
</tr>
<tr>
<td>$2000 \times 1000$</td>
<td>4</td>
<td>4.626</td>
<td>0.99</td>
<td>4.588</td>
<td>0.99</td>
<td>4.139</td>
<td>0.90</td>
</tr>
<tr>
<td>$2000 \times 1000$</td>
<td>8</td>
<td>2.395</td>
<td>0.98</td>
<td>2.285</td>
<td>0.93</td>
<td>2.281</td>
<td>0.93</td>
</tr>
<tr>
<td>$2000 \times 1000$</td>
<td>16</td>
<td>1.519</td>
<td>0.94</td>
<td>1.440</td>
<td>0.89</td>
<td>1.388</td>
<td>0.87</td>
</tr>
<tr>
<td>$2000 \times 2000$</td>
<td>4</td>
<td>26.95</td>
<td>0.96</td>
<td>26.31</td>
<td>0.94</td>
<td>25.06</td>
<td>0.89</td>
</tr>
<tr>
<td>$2000 \times 1000$</td>
<td>8</td>
<td>13.78</td>
<td>0.92</td>
<td>13.66</td>
<td>0.91</td>
<td>14.22</td>
<td>0.91</td>
</tr>
<tr>
<td>$2000 \times 1000$</td>
<td>16</td>
<td>7.666</td>
<td>0.95</td>
<td>7.417</td>
<td>0.93</td>
<td>7.539</td>
<td>0.93</td>
</tr>
<tr>
<td>$4000 \times 200$</td>
<td>8</td>
<td>3.058</td>
<td>0.98</td>
<td>3.029</td>
<td>0.93</td>
<td>2.970</td>
<td>0.91</td>
</tr>
<tr>
<td>$4000 \times 1000$</td>
<td>8</td>
<td>114.5</td>
<td>0.98</td>
<td>114.2</td>
<td>0.97</td>
<td>120.3</td>
<td>0.94</td>
</tr>
<tr>
<td>$4000 \times 1000$</td>
<td>16</td>
<td>58.53</td>
<td>0.98</td>
<td>57.38</td>
<td>0.97</td>
<td>54.69</td>
<td>0.97</td>
</tr>
<tr>
<td>$5000 \times 100$</td>
<td>8</td>
<td>0.126</td>
<td>0.93</td>
<td>0.123</td>
<td>0.87</td>
<td>0.116</td>
<td>0.82</td>
</tr>
<tr>
<td>$5000 \times 1000$</td>
<td>16</td>
<td>2.012</td>
<td>1.03</td>
<td>2.029</td>
<td>1.03</td>
<td>1.918</td>
<td>0.98</td>
</tr>
<tr>
<td>$5000 \times 5000$</td>
<td>16</td>
<td>112.8</td>
<td>1.06</td>
<td>112.5</td>
<td>1.06</td>
<td>105.7</td>
<td>0.99</td>
</tr>
<tr>
<td>$8000 \times 1000$</td>
<td>16</td>
<td>3.767</td>
<td>1.26</td>
<td>3.660</td>
<td>1.24</td>
<td>3.762</td>
<td>1.15</td>
</tr>
<tr>
<td>$8000 \times 8000$</td>
<td>16</td>
<td>638.2</td>
<td>1.05</td>
<td>635.9</td>
<td>1.00</td>
<td>620.9</td>
<td>0.99</td>
</tr>
<tr>
<td>$10000 \times 1000$</td>
<td>16</td>
<td>5.613</td>
<td>1.04</td>
<td>5.613</td>
<td>1.04</td>
<td>5.613</td>
<td>1.04</td>
</tr>
<tr>
<td>$10000 \times 10000$</td>
<td>16</td>
<td>1231</td>
<td>1.07</td>
<td>1226</td>
<td>0.97</td>
<td>1129</td>
<td>0.87</td>
</tr>
</tbody>
</table>

$t_5$ — the parallel SVD with a block one-sided bidiagonalization algorithm. It is a simplified block version of the one-sided bidiagonalization. Only the application of the Householder reflectors is aggregated, but the BLAS 2.5 approach is not used since it introduces one matrix-vector product more per iteration.

$t_6$ — the parallel SVD with a modified block one-sided bidiagonalization algorithm. It is the blocked version with the modification of Campos et al.

As we can see from Table 6.3, the situation on the Sarek cluster is a little bit different from the situation described in subsection 6.1. The Sarek cluster has very fast interprocessor communication and Algorithm 5.1 is not so favorable as it was on the cluster of computers in the Advanced Computing Laboratory in the case when $m > n$. The worst result was obtained for matrix dimensions $10000 \times 1000$ and 16 processors when the execution time of the parallel SVD with Algorithm 5.1 was 1.46 times longer than the execution time of $\text{psgesvd}()$. This was the reason for developing three additional implementations of the parallel stable one-sided bidiagonalization routines, which are explained in the description of Table 6.3. The modification of Campos and his coauthors proposes reduction in communication events. This is achieved by simultaneous execution of the operations in steps (6) and (12) of Algorithm 5.1, since each of these operations presents a communication event. Thus instead of two communication events, only one is performed per iteration. The work of Campos et al. was developed independently of the work presented in this paper. The block parallel implementation is similar to Algorithm 3.1, except that BLAS 2.5 operations are not implemented. We can conclude that the presented variants of Algorithm 3.1 did gain some speedup and that block and modified block algorithms are faster than the
ScaLAPACK routine in most cases. For some matrix dimensions these two variants are up to 21.5% faster. On the other hand, the matrix dimension $10000 \times 1000$ for 16 processors remains critical. We achieved a reduction in slowdown from 45.53% to 18.25%, but the ScaLAPACK routine is still faster in this case. There is still an open question of whether we can make our parallel implementation of the one-sided bidiagonalization more efficient on the clusters with a fast network. This will be the subject of our future work.

7. Conclusion. Ralha’s one-sided bidiagonalization [29] performs fewer operations than the standard Golub–Kahan bidiagonalization [20], but it is numerically unstable. Barlow’s modification [3] of Ralha’s algorithm avoided this problem, so the new algorithm became numerically backward stable for computing the matrices $\Sigma$ and $V$, with the same operation count. On modern computers, a smaller operation count does not necessarily lead to reduced execution times, due to the time spent on communication between different levels of memory. This is the reason why we have developed block and parallel versions of the new stable one-sided bidiagonalization algorithm. The block version optimizes the usage of faster memory, without sacrificing numerical backward stability. As numerical tests demonstrate, the SVD algorithm with the block one-sided bidiagonalization is faster than the corresponding LAPACK routine. The stable one-sided bidiagonalization is more suitable for parallelization than the corresponding ScaLAPACK routine. The tests established improvement of the parallel one-sided bidiagonalization when compared with the ScaLAPACK routine in most cases. In the best case our algorithm is 68.28% faster than the ScaLAPACK routine on a cluster with slow interprocessor communication and 21.5% faster on a cluster with fast interprocessor communication. From the numerical point of view it is equivalent to the original stable one-sided algorithm proposed in [3] or its block version and thus numerically stable in the same way. In the future we will work on the more efficient parallel version of the stable one-sided bidiagonalization.

Acknowledgments. We thank Zlatko Drmač and Daniel Kressner for carefully reading the paper and making many helpful suggestions. Rui Ralha and Carlos Campos are developing one-sided bidiagonalization software, and we thank them for their support. We also thank both referees for helping us to make this paper better. This research was conducted using the resources of the High Performance Computing Center North (HPC2N).

REFERENCES


