Householder Bidiagonalization
Computational Efficiency
Extensions to the Parallel and Sparse Cases

Gary Howell, HPC/ITD
NC State University
gary_howell@ncsu.edu
Charles Fulton, Madhan Premkumar, Sumit Malhotra
Jim Parker, Karen Marmol, Florida Tech
James Demmel, UC Berkeley
Sven Hammarling, NAG
If you have your laptop and want to see these slides

• google hpc +, ncsu
  or http://www.ncsu.edu/itd/hpc/main.php

• click Docs & Pubs in left toolbar,

• click Gary Howell

• click BLAS2.5 San Francisco
Acknowledgements

• NSF EIA-0103642 Next Generation Software – which supported workers at Florida Tech and paid for visits by myself and Sven Hammarling to Melbourne.

• ERDC MSRC, Pittsburgh Supercomputing, NC State and Florida Tech for computer time

• NAG for Sven’s time. HP and NC State ITD/HPC for my time.

• This work was funded in part by the National Institutes of Health through the NIH Roadmap for Medical Research, Grant 1 P20 HG003900-01. Information on the Molecular Libraries Roadmap Initiative can be obtained from http://nihroadmap.nih.gov/molecularlibraries/
Bidiagonalization is a Bottleneck in Computing Singular Values and Vectors

If only singular values are desired, the reduction to bidiagonal form is the only \( O(n^3) \) flop part of the algorithm.

In LAPACK DGESVD, bidiagonalization required only a small fraction of the total singular value and vector computation. The more recent LAPACK DGESSD (using divide and conquer methods to produce singular values and vectors) spends 40-50% of the computation in bidiagonalization. Work of Dhillon and Parlett may further decrease the proportion of the computation spent in computing singular vectors.
Speeding Bidiagonalization

- Here we show how to accomplish Householder bidiagonalization by combining several BLAS 2 calls into one BLAS 2.5 call (combining calls $y^tA$ and $Ax$ into one call, so need read $A$ only once) The combining call we refer to as _GEMVT (in place of two _GEMV calls. It significantly speeds the computation.

- The parallel _GEMVT improves performance on machines for which passing short messages is less expensive than refilling the cache.
Extension of Dense Algorithms to Sparse Case

• All block update algorithms extend naturally to the sparse case. Because until the block update is done, you still are performing matrix vector multiplications by the original sparse matrix.

• So how do cache considerations extend to performing $y^T A$ and $Ax$ in the sparse case? Required for either the BLAS 2.5 - BLAS 3 algorithm or Lanczos methods.

• Grosser, Lang reduce to small band form by totally BLAS 3 operations, then carry on to bidiagonal via BLAS 1 operations. Sparse $AX$, $A$ sparse $X$ dense can be much faster than $Ax$. 
Factor of 100 serial speedup

Mflops in Multiplying by a Random Sparse Matrix

Number of Column Blocks

Number of Vectors
Status

- A Fortran 77 Householder bidiagonalization routine fits into LAPACK, giving good speedups and passing the LAPACK tests for the case $m \geq n$. Double precision and double precision complex codes exist.

- Good performance depends on adjusting block size to the machine. Each call to _GEMVT makes a call to a revised ILAENV routine. Should I provide automated routines to get the two required parameters? What format do you want?

- Results are less consistent when $n \geq m$ when performed by row blocking as opposed to column blocking. For consistent
speedups, use "in-place" transposition of matrices.

- For machines with large cache memories and fast interconnects, parallel _GEMVT is effective (MS theses of Sumit Malhotra and Madhan Premkumar). We do not have parallel bidiagonalization code.

- _GEMVT and/or blocking speed computation in the sparse case. There are preliminary sparse Fortran codes.
Half the flops in reduction to small-band are BLAS-2.

Column by column reductions to Hessenberg, tridiagonal, or bidiagonal form involve two matrix vector multiplies per column elimination.

For orthogonal reduction to Hessenberg, the trailing matrix must be transferred from main memory to cache twice for each column elimination. Example, Pentium 833 (MHz CPU) with a 64 MHz 64 bit bus (64 bits per bus clock cycle) transfers at most 64 million double precision numbers per second.

For a matrix vector multiply for which the matrix $A$ resides in RAM, we can do 2 flops per element of $A$ transported, i.e., our speed is limited to 128 MFlops. Which is about what is obtained.
Naive Householder
Bidiagonalization

\[ A = UBV \]

is a \( URV \) decomposition that is also the \( O(n^3) \) flop portion of determining singular values.

A column-row elimination with \( \|u\| = \sqrt{2} = \|v\| \)

is accomplished by

\[ A \leftarrow (I - uu^T)A(I - vv^T) \]

naturally corresponding to the four BLAS-2 operations

\[ z^T \leftarrow u^T A, \quad \text{_GEMV} \]
\[ w \leftarrow Av, \quad \text{_GEMV} \]
\[ A \leftarrow A - uz^T, \quad \text{_GER} \]
\[ A \leftarrow A - wv^T, \quad \text{_GER} \]

where _GER operations are rank one updates.

But this requires four reads and two writes of \( A \) for eliminating one column-row pair.
Data Transfer of Bidiagonalization Algorithms

The current LAPACK algorithm defers updates (performed as a BLAS-3 matrix matrix multiply) but requires the two _GEMV calls.

Table Compares Floating Point Reads and Writes of the Trailing Matrix in One Row Column Elimination

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>BLAS level</th>
<th>Naive</th>
<th>LAPACK</th>
<th>I</th>
<th>III</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2 &amp; 3</td>
<td>2.5</td>
<td>2.5 &amp; 3</td>
</tr>
<tr>
<td>READS</td>
<td></td>
<td>4</td>
<td>2 + 1/b</td>
<td>1</td>
<td>1 + 1/b</td>
</tr>
<tr>
<td>WRITES</td>
<td></td>
<td>2</td>
<td>1/b</td>
<td>1</td>
<td>1/b</td>
</tr>
</tbody>
</table>

BLAS 2.5 operations reduce data transfer by combining several BLAS-2 calls into one operation.
Reductions to similar Hessenberg, banded Hessenberg, tridiagonal, or bidiagonal forms are the predominant execution time in determining principal or eigenvalues. The two new BLAS 2.5 _GEMVER and _GEMVT operators make the combined operation $Ax$ and $y^TA$ more efficient by eliminating a read from each column row elimination.

_GEMVER performs

$$\begin{align*}
\hat{A} & \leftarrow A + \alpha u_1 v_1^T + \beta u_2 v_2^T \\
\hat{y}^T & \leftarrow y^T \hat{A} \\
x & \leftarrow \hat{A}\hat{y} + z
\end{align*}$$

Reduction to small band or bidiagonal form by _GEMVER does not require a BLAS-3 update.
BLAS 2 \_GEMV\ performs

\[ y \leftarrow Ax + z \]

BLAS 2.5 \_GEMVT\ performs

\[ \hat{y}^T \leftarrow y^T A \]
\[ \hat{x} \leftarrow A\hat{y} + z \]

\_GEMVT has no update.

MFlops for Matrix Vector Multiplies

<table>
<thead>
<tr>
<th></th>
<th>BLAS 2</th>
<th>BLAS 2.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_GEMV Ax</td>
<td>26.2</td>
<td>49.3</td>
</tr>
<tr>
<td>_GEMVT (y^T A &amp; Ax)</td>
<td>120</td>
<td>190</td>
</tr>
<tr>
<td>SPARC 200Mhz</td>
<td>221</td>
<td>335</td>
</tr>
<tr>
<td>Pentium 833Mhz</td>
<td>371</td>
<td>691</td>
</tr>
<tr>
<td>Alpha 1000Mhz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xeon 2800Mhz</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tricks for Efficient Bidiagonalization

A. Compute $u^T A$ and $A(A^T u)$ in one pass of data through cache.

B. Perform $(I - uu^T)A(I - vv^T)$ in one pass of data through cache.

C. Multiply by a matrix not yet updated.

D. Multiply by a Householder vector which eliminates a row (where the row is being computed in the same pass of the matrix through cache memory).
Trick A. Perform $u^T A$ and $A( A^T u)$ in one access of data

Let

$$A = \begin{pmatrix} A_1 | A_2 | \cdots | A_n \end{pmatrix}$$

If we perform

For $i = 1 : n,$

$$v_i = u^T A_i,$$

$$w = w + A_iv_i,$$

End

then only one block of $A$ is accessed at a time.
Choosing the Column Block Size for 
_GEMVT, call to ILAENV on each 
call to GEMVT

ELSE IF( C3.EQ.'MVT' ) THEN
  IF ( SNAME ) THEN
    NBMIN = 12
  *
  * Tuning consists of choosing NBMIN and "120000"
  *
    NX = 120000/N1
    NB = MIN(NBMIN*(NX/NBMIN),160)
    IF ( NX.LT.NBMIN .AND. NX.GE.NBMIN/2) THEN
      NB = NBMIN/2
    ELSEIF (NX.LT.NBMIN/2) THEN
      NB = N2
    ENDIF
  ELSE
    NBMIN depends on loop-unrolling parameters
    in underlying BLAS. "120000" depends on the
    size of the largest cache.
Mflop Rates for Bidiagonalization on a 2.8 GHz Xeon with 512KByte Cache

- ○ LAPACK dgebrd
- + new dgebrd

Square Matrix Size

Mflop rate (millions of flops per second)
Bidirectional on a 2.0 GHz Opeteron, 2MByte cache

Megaflop Rate — Column Blocks of 120K Entries

Size of a Square Matrix
Related work available by anonymous
ftp at
cs.fit.edu
cd pub/howell
or
Parallel GEMVT – First Steps

Initially we performed GEMVT by two successive calls to the parallel GEMV PDGEMV (PBLAS parallel matrix vector multiply).

Sumit Malhotra modeled the communication overhead and verified that as problem size grows large a block cyclic layout entails less communication overhead than a column cyclic layout. The column cyclic layout has more total messages, but the total length of passed messages is less.

For a column cyclic data layout, each processor passes messages of length $O(n)$. For a block cyclic layout, each of $p$ processor passes messages of length $n/\sqrt{p}$. Madhan Premkumar verified the results shown on following graphs.
Why Use Parallel GEMVT?

There are two reasons parallel GEMVT can be helpful. As in the serial case, reuse of in-cache data can improve efficiency. But also we can reduce the volume of data transmitted.

Analysis and numeric experiments indicate that cache reuse is useful on high performance machines with 8 MByte caches and low latency communication (ERDC, U of Pittsburgh supercomputing center), but less useful on machines such as Beowulfs style computers with ethernet interconnects, (Florida Tech and NC State University).

If the time to reload the cache is longer than the time to pass log(p) messages than parallel GEMVT makes sense.

For the SC 45 with a quadrics interconnect, we get 60% parallel efficiency for GEMVT (up to 200 processors) where the same operations by two calls to GEMV require twice the time.
Computations on 512 CPU Alpha SC at ERDC MSRC
DGEMVT — each CPU has 40K columns and 400 rows

Number of Processors — only communication on 2nd gemv

Time in Seconds — Fixed number of flops/processor

+ 1 block
O 10 blocks
X 100 blocks
Computations on 512 CPU Alpha SC at ERDC MSRC
Computations on 512 CPU Alpha SC at ERDC MSRC
**Trick A. Parallel** $u^T A$ and $A(A^T u)$  
**reusing data in cache**

Let

\[ A = \left( \begin{array}{c|c|c} A_1 & | & A_2 \\ \vdots & | & \vdots \\ | & | & | \\ A_l & | & A_l \end{array} \right) \]

and also

\[ A = \begin{pmatrix} \{A^1\} \\ \{A^2\} \\ \vdots \\ \{A^k\} \end{pmatrix} \]

so that processor $i, j$ of $k l$ processors contains the intersection of $A^i$ and $A_j$. 
The parallel operation is

\[ v_{ij} = u_i^T A_{ij} \quad \text{each\_proc} \]

\[ v_j = \text{allreduce}(+v_{ij}) \quad \text{col\_comm} \]

\[ w_{ij} = A_{ij} v_j \quad \text{each\_proc} \]

\[ w_i = \text{allreduce}(+w_{ij}) \quad \text{row\_comm} \]

Here we’re assuming that \( A_{ij} \) fits in cache. In practice, fitting \( A_{ij} \) in cache is likely to mean that a given processor has more than one block \( A_{ij} \).
Trick C. Multiplying by a Matrix as Yet Not Updated

If

\[ \tilde{A} = A - \sum_{i}^{k} w_i v_i^T \]

then

\[ u^T \tilde{A} = u^T A - \sum_{i}^{k} (u^T w_i) v_i^T \]
\[ = u^T A - (u^T W) V^T \]

becomes three matrix vector products.

Inexpensive so long as \( k \) is small.

Trick C is essential in block algorithms (or in using a dense algorithm in the sparse case).
The Sparse Case

Golub and Van Loan, P. 498, 2nd Ed. “Unfortunately, if \( A \) is large and sparse, then we can expect large, dense submatrices to arise during the Householder bidiagonalization.” In context, this is a justification for using the unstable Lanczos procedure for bidiagonalizing sparse matrices.

Actually, Algorithm II allows sparse bidiagonalization without fill. That is to say, as long as we don’t actually do the updates, we are still operating on the the original matrix, touching it only to do matrix vector multiplications.
Householder Bidiagonalization
Comparable in Expense to Lanczos Bidiagonalization

Claim: if the Householder vectors require comparable storage to the original sparse $A$, Algorithm II is competitive to Lanczos in storage and overall computational costs.

For storage: If more than a few steps of Lanczos are used than the Lanczos vectors must be saved for use in re-orthogonalization. But this is as much storage as saving the Householder vectors.

For computational costs: the extra Householder flops are in dense computations, which are relatively fast compared to the predominant cost $y^TA$ and $Ax$.

All things being equal we should use the more stable algorithm?
Sparse Matrix Vector Multiplication and BLAS 2.5

For either sparse Householder bidiagonalization or for Lanczos bidiagonalization the main computational expense is in the multiplications $y^T A$ and $Ax$. Let’s take the case of dense $x, y$, sparse $A$, $A$ too large to fit in cache.

Multiplication by sparse $A$ is arranged so that either $y ← Ax$ or $w ← A^T y$ stream $A$ from RAM. When $A$ is banded or consists as in finite differences of several bands, then none of $w, y, x$ suffer many cache misses in the multiplication.

Combining the multiplications by $A$ and $A^T$ gives some speedup. For example, on a Xeon I saw (intel compiler -xW -tpp7 flags) 244 Mflops vs. 304 Mflops for the combined operations. (about 1/20 of the advertised peak speed).

This trick should be orthogonal to other techniques such as getting dense subblocks.
$x$, $y$ with many cache misses

For definiteness, take the artificially evil case that entries of $A$ are randomly distributed, $A$ stored in sparse row storage. That is, we read the entries of $A$ as English speakers read text.

Then $Ax$ is sparse ddots, relatively fast, which entail read misses in $x$.

For $y^TA$ we have no read misses, as the same entry $y(i)$ multiplies every entry in the $i$th row of $A$, but we update random entries in the product, so we have write misses.

For $A$ in sparse row storage, $Ax$ is typically faster than $y^tA$. Not so much in coordinate storage, matrix ordered the same way (complete vector of row indices for $A$).
One solution sometimes used is to keep two copies of $A$. If we want to save storage, we can instead use block storage.

What I’ve been trying is to use row storage for column blocks. (like a reading a newspaper with multiple columns).

The conversion routine first goes to coordinate storage, then sorts. In place quick sorts, so time goes like $O(nz \log(nz))$.

This storage scheme appears to equalize the speeds of $Ax$ and $y^t A$. 
As any dense method that defers updates is really a sparse method (until the update), note that the Lang - Bischof - Sun type algorithms also extend to the sparse case.
Mflops in Multiplying by a Random Sparse Matrix

Number of Column Blocks

Number of Vectors
Mflops — increasing numbers of vectors and column blocks, 2.8 GHz, g77 -O3

Mflops along the diagonal of number of vectors = number of column blocks

A\times x
\times tr(A)\times y
\circ combined operation
Mflops for no column blocking

Number of vectors multiplied by sparse matrix

- $A\times x$
- $\text{tr}(A)\times y$
- Combined operation

Mflops - 200k by 200K matrix, 0.05% dense
Mflops for no column blocking

- $A^*x$
- $\text{tr}(A)^*y$
- combined operation

Mflops — 200k by 200K matrix, 0.05% dense

Number of vectors multiplied by sparse matrix

0 2 4 6 8 10 12

0 100 200 300 400 500 600 700 800 900 1000

36
Mflops for Sparse Matrix vector multiply with column blocking

Increasing number of column blocks, single vector

A*x
tr(A)*y
combined operation

Mflops — 200k by 200K matrix, 0.5% dense
Column Blocking (or more generally blocking) helps when the product and/or multiplicand vectors get too large for storage. It’s cost is $O\log(nz)nz$ and additional storage for storing a complete and column vectors. (About 33 compared to compressed row storage).

Multiplying by more vectors at a time ”always” improves performance. It’s easy to do – but alas may involve a rewrite of whatever algorithm you’re interested in. For example, the bidiagonalization rewrite is a bit tough. (I’ve done the sparse BLAS 2.5 - 3 version in Fortran, but don’t yet have Matlab code for the multiple vector version).
Statistician complaints about a partial SVD for a sparse matrix

- We want a sparse decomposition
- We want a positive decomposition
- We want a local decomposition
- Sometimes we don’t have a matrix (missing data for some entries).

But trying all these things at once gives neural net like algorithms which are orders of magnitude slower?
Conclusions

• BLAS 2.5 can significantly reduce the time to find singular values (bidiagonalization), eigenvalues of symmetric matrices (Stanley tridiagonalization), eigenvalues of unsymmetric matrices (Howell reduction to small-band form). In each case the reduced form allows determination of eigenvalues or singular values in $O(n^2)$ additional flops.

• Applying these ideas to the sparse and parallel cases can lead to efficient algorithms in those cases also.

• The new bidiagonalization algorithms give a useful $URV$ decomposition for sparse least squares problems and low rank least squares problems.
e-mails

gary_howell@ncsu.edu
fulton@zach.fit.edu

For related work, see the web page
//http://hpc.ncsu.edu/Documents/Publications/gary_howell/contents.html
Blocking for Bidiagonalization

\[
A = \begin{bmatrix}
B_k & 0 \\
\vdots & \ddots & \ddots \\
0 & \ddots & \ddots & 0 \\
\end{bmatrix}
\begin{bmatrix}
\hat{v}^T \\
\hat{u} & A_{n-k-1} \\
\end{bmatrix}
\]

\[
= \begin{bmatrix}
B_k & 0 \\
\vdots & \ddots & \ddots \\
0 & \ddots & \ddots & 0 \\
\end{bmatrix}
\begin{bmatrix}
V^T \\
U & A_{n-k-b} \\
\end{bmatrix}
\]

Block bordered algorithms produce \( U \) and \( V \), deferring BLAS-3 updates to be performed on \( A_{n-k-b} \).
3 BLAS 2.5 Bidiagonalization Algorithms

In associative arithmetic these would each result in the same bidiagonal matrix.

**Alg I.** Put almost all flops in the BLAS 2.5 operator \_GEMVER. May be efficient when parallel communications are relatively noisy so that the cost of a write to RAM is amortized.

**Alg II.** A never update algorithm. All mat-vec-mults with the original sparse matrix. Desirable for low rank and/or sparse least squares.

**Alg III.** BLAS 2.5 \_GEMVT with BLAS-3 matrix updates. Efficient when writes to main memory are more expensive than reads.
Assumption: algorithm execution time is proportional to amount of data transferred from cache to main memory.
Trick B. Performing

\((I - uu^T)A(I - vv^T)\)

Break up the computation into

\[ \hat{A} \leftarrow A - u_{old}z_{old}^T - w_{old}v_{old}^T \]

From knowing the updated matrix we can compute

\[ u_{new}^T\hat{A} \]

and

\[ \hat{A}v_{new}. \]

As in the last slide this can be performed in one loop through the matrix. In this case each successive column block of \(A\) is involved in a rank two update and two \_GEMVs.

Trick B allows implementation of Algorithm I by \_GEMVER.
Trick D. Multiplying by a Householder vector before we have it

The row to be eliminated is given as \( \hat{y} \rightarrow y + u^T A \). We have seen we can perform \( u^T A \) and \( A\hat{y} \) in one call to \_GEMVT. But actually we want to perform \( Av \) where

\[
v = \frac{\hat{y} + \alpha e_1}{\kappa}.
\]

Thus

\[
Av = \frac{1}{\kappa}(A\hat{y} + \alpha A e_1)
\]

with \( A e_1 \) is simply a column of \( A \).

So \( A\hat{y} \) is merely a BLAS-1 \_AXPY away from the desired product.