Mixed single-double precision solver

Julien Langou, University of Colorado Denver.

Mixed single-double precision solver
Wednesday April 1st, 2008.

1. Mixed single-double precision solver
2. Rectangular Full Packed Format
3. Communication optimal algorithm
<table>
<thead>
<tr>
<th>Architecture (BLAS/LAPACK)</th>
<th>n</th>
<th>DGEMM/SGEMM</th>
<th>DGETRF/SGETRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Pentium III Coppermine (Goto)</td>
<td>3500</td>
<td>2.10</td>
<td>2.24</td>
</tr>
<tr>
<td>Intel Pentium III Katmai (Goto)</td>
<td>3000</td>
<td>2.12</td>
<td>2.11</td>
</tr>
<tr>
<td>SunUltraSPARC IIe (Sunperf)</td>
<td>3000</td>
<td>1.45</td>
<td>1.79</td>
</tr>
<tr>
<td>Intel Pentium IV Prescott (Goto)</td>
<td>4000</td>
<td>2.00</td>
<td>1.86</td>
</tr>
<tr>
<td>Intel Pentium IV-M Northwood (Goto)</td>
<td>4000</td>
<td>2.02</td>
<td>1.98</td>
</tr>
<tr>
<td>AMD Opteron (Goto)</td>
<td>4000</td>
<td>1.98</td>
<td>1.93</td>
</tr>
<tr>
<td>Cray X1 (libsci)</td>
<td>4000</td>
<td>1.68</td>
<td>1.54</td>
</tr>
<tr>
<td>IBM Power PC G5 (2.7 GHz) (VecLib)</td>
<td>4000</td>
<td>2.29</td>
<td>2.03</td>
</tr>
<tr>
<td>Compaq Alpha EV6 (CXML)</td>
<td>3000</td>
<td>0.99</td>
<td>1.08</td>
</tr>
<tr>
<td>IBM SP Power3 (ESSL)</td>
<td>2000</td>
<td>1.03</td>
<td>1.13</td>
</tr>
<tr>
<td>SGI Octane (ATLAS)</td>
<td>1500</td>
<td>1.08</td>
<td>1.13</td>
</tr>
<tr>
<td>Intel Itanium 2 (Goto)</td>
<td>1500</td>
<td>0.71</td>
<td>–</td>
</tr>
</tbody>
</table>
Mixed single-double precision solver

<table>
<thead>
<tr>
<th></th>
<th>Single</th>
<th>Double</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x87</td>
<td>SSE</td>
<td>3DNow!</td>
</tr>
<tr>
<td>Pentium</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pentium II</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Pentium IV</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Athlon</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Enhanced Athlon</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Athlon 4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Athlon MP</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**MMX** Set of “MultiMedia eXtensions” to the x86 ISA. Mainly new instructions for integer performance, and maybe some prefetch. For Intel, all chips starting with the Pentium MMX processor possess these extensions. For AMD, all chips starting with the K6 possess these extensions.

**SSE** Streaming SIMD (Single Instruction Multiple Data) Extensions. SSE is a superset of MMX. These instructions are used to speed up single precision (32 bit) floating point arithmetic. For Intel, all chips listed starting with the Pentium III possess SSE extensions. For AMD, all chips starting from Athlon4 possess SSE.

**3DNow!** AMD’s extension to MMX that does almost the exact same thing SSE does, except the single precision arithmetic is not IEEE compliant. It is also a superset of MMX (but not of SSE; 3DNow! was released before SSE). It is supported only on AMD, starting with the K6-2 chip.

**SSE2** Additional instructions that perform double precision floating arithmetic. Allows for 2 double precision FLOPs every cycle. For Intel, supported on the Pentium 4 and for AMD, supported on the Opteron.
Mixed single-double precision solver

- Factorize the matrix in single precision ($O(n^3)$ FLOPs)
  - Then use this factorization as a preconditioner in a double precision iterative method ($O(n^2)$)
- The single LU factorization is a **poor** double LU factorization but an **excellent** preconditioner.
- Typically the iterative methods will converge in a few steps

- single precision computation dominate the # of FLOP: $O(n^3)$ in single vs $O(n^2)$ in double:

  \begin{align*}
  \text{Speed of single precision}
  \end{align*}

- Double precision iterative solver:

  Accuracy in double precision
Mixed precision, Iterative Refinement for Direct Solvers

1. $LU \leftarrow PA$ \hspace{1cm} ($\varepsilon_s$)
2. solve $Ly = Pb$ \hspace{1cm} ($\varepsilon_s$)
3. solve $Ux_0 = y$ \hspace{1cm} ($\varepsilon_s$)
   do $k = 1, 2, ...$
4. $r_k \leftarrow b - Ax_{k-1}$ ($\varepsilon_d$)
5. solve $Ly = Pr_k$ ($\varepsilon_s$)
6. solve $Uz_k = y$ ($\varepsilon_s$)
7. $x_k \leftarrow x_{k-1} + z_k$ ($\varepsilon_d$)
   check convergence
done

\[ TIME = \rho_{32} \left( \frac{2}{3} n^3 \right) + \#ITER \cdot \rho_{64} \cdot O(n^2) \]
Mixed single-double precision solver

Hardware and software details of the systems used for performance experiments.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Clock [GHz]</th>
<th>Peak SP / Peak DP</th>
<th>Memory [MB]</th>
<th>BLAS</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Opteron 246</td>
<td>2.0</td>
<td>2</td>
<td>2048</td>
<td>Goto-1.13</td>
<td>Intel-9.1</td>
</tr>
<tr>
<td>IBM PowerPC 970</td>
<td>2.5</td>
<td>2</td>
<td>2048</td>
<td>Goto-1.13</td>
<td>IBM-8.1</td>
</tr>
<tr>
<td>Intel Xeon 5100</td>
<td>3.0</td>
<td>2</td>
<td>4096</td>
<td>Goto-1.13</td>
<td>Intel-9.1</td>
</tr>
<tr>
<td>STI Cell BE</td>
<td>3.2</td>
<td>14</td>
<td>512</td>
<td>–</td>
<td>Cell SDK-1.1</td>
</tr>
</tbody>
</table>
Mixed single-double precision solver

Performance improvements for direct dense methods when going from a full double precision solve (reference time) to a mixed precision solve.

<table>
<thead>
<tr>
<th></th>
<th>Nonsymmetric</th>
<th>Symmetric</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Opteron 246</td>
<td>1.82</td>
<td>1.54</td>
</tr>
<tr>
<td>IBM PowerPC 970</td>
<td>1.56</td>
<td>1.35</td>
</tr>
<tr>
<td>Intel Xeon 5100</td>
<td>1.56</td>
<td>1.43</td>
</tr>
<tr>
<td>STI Cell BE</td>
<td>8.62</td>
<td>10.64</td>
</tr>
</tbody>
</table>

Mixed precision, iterative refinement method for the solution of dense linear systems on the STI Cell BE processor.
Mixed single-double precision solver

Intel Pentium III CopperMine (0.9GHz), Goto BLAS

![Graph showing performance comparison between different BLAS functions on Intel Pentium III Coppermine (0.9GHz) processor using Goto BLAS library. The graph plots percent time versus DGETRF for various operations including SGEMM/DGEMM, DGESV/DGETRF, DSXGESV/DGETRF, SGETRF/DGETRF, ITER*SGETRS/DGETRF, ITER*DGEMV/DGETRF, and ITER*EXTRA/DGETRF. The x-axis represents the size of the matrix (n) ranging from 500 to 4000, while the y-axis shows the percent time compared to DGETRF, with values ranging from 0 to 110 percent.]
Mixed single-double precision solver

Cluster AMD Opteron(tm) 64 processors, Open MPI (MX) (nb=121, p=4, q=16)
Mixed single-double precision solver

Test matrices for sparse mixed precision, iterative refinement solution methods.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SiO</td>
<td>33401</td>
<td>1317655</td>
<td>yes</td>
<td>no</td>
<td>$O(10^3)$</td>
</tr>
<tr>
<td>2</td>
<td>Lin</td>
<td>25600</td>
<td>1766400</td>
<td>yes</td>
<td>no</td>
<td>$O(10^5)$</td>
</tr>
<tr>
<td>3</td>
<td>c-71</td>
<td>76638</td>
<td>859554</td>
<td>yes</td>
<td>no</td>
<td>$O(10)$</td>
</tr>
<tr>
<td>4</td>
<td>cage-11</td>
<td>39082</td>
<td>559722</td>
<td>no</td>
<td>no</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>5</td>
<td>raefsky3</td>
<td>21200</td>
<td>1488768</td>
<td>no</td>
<td>no</td>
<td>$O(10)$</td>
</tr>
<tr>
<td>6</td>
<td>poisson3Db</td>
<td>85623</td>
<td>2374949</td>
<td>no</td>
<td>no</td>
<td>$O(10^3)$</td>
</tr>
</tbody>
</table>

Performance improvements for direct sparse methods when going from a full double precision solve (reference time) to a mixed precision solve.

<table>
<thead>
<tr>
<th>Matrix number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Opteron 246</td>
<td>1.827</td>
<td>1.783</td>
<td>1.580</td>
<td>1.858</td>
<td>1.846</td>
<td>1.611</td>
</tr>
<tr>
<td>IBM PowerPC 970</td>
<td>1.393</td>
<td>1.321</td>
<td>1.217</td>
<td>1.859</td>
<td>1.801</td>
<td>1.463</td>
</tr>
<tr>
<td>Intel Xeon 5100</td>
<td>1.799</td>
<td>1.630</td>
<td>1.554</td>
<td>1.768</td>
<td>1.728</td>
<td>1.524</td>
</tr>
</tbody>
</table>
Mixed single-double precision solver

Mixed precision, inner-outer FGMRES\( (m_{\text{out}}) \)-GMRES\( SP(m_{\text{in}}) \)

1: for \( i = 0, 1, \ldots \) do  
2: \( r = b - Ax_i \) \((\varepsilon_d)\)  
3: \( \beta = h_{1,0} = ||r||_2 \) \((\varepsilon_d)\)  
4: check convergence and exit if done  
5: for \( k = 1, \ldots, m_{\text{out}} \) do  
6: \( v_k = r / h_{k,k-1} \) \((\varepsilon_d)\)  
7: Perform one cycle of GMRES\( SP(m_{\text{in}}) \) in order to (approximately) solve \( A\hat{z}_k = v_k \), (initial guess \( \hat{z}_k = 0 \)) \((\varepsilon_s)\)  
8: \( r = A \hat{z}_k \) \((\varepsilon_d)\)  
9: for \( j=1, \ldots, k \) do  
10: \( h_{j,k} = r^T v_j \) \((\varepsilon_d)\)  
11: \( r = r - h_{j,k} v_j \) \((\varepsilon_d)\)  
12: end for  
13: \( h_{k+1,k} = ||r||_2 \) \((\varepsilon_d)\)  
14: end for  
15: Define \( Z_k = [z_1, \ldots, z_k] \), \( H_k = \{ h_{i,j} \}_{1 \leq i \leq k+1, 1 \leq j \leq k} \) \((\varepsilon_d)\)  
16: Find \( y_k \), the vector of size \( k \), that minimizes \( ||\beta e_1 - H_k y_k||_2 \) \((\varepsilon_d)\)  
17: \( x_{i+1} = x_i + Z_k y_k \) \((\varepsilon_d)\)  
18: end for
Mixed single-double precision solver

Mixed precision iterative refinement with FGMRES-GMRES\textsubscript{SP} vs FGMRES-GMRES\textsubscript{DP} (left) and vs full double precision GMRES (right).

<table>
<thead>
<tr>
<th>matrix n.</th>
<th>$m_{in}$</th>
<th>$m_{out}$</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>20</td>
<td>150</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>9</td>
<td>300</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>20</td>
<td>300</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>10</td>
<td>50</td>
</tr>
</tbody>
</table>
Number of iterations needed for our mixed precision method to converge to an accuracy better than the one of the associated double precision solve as a function of the condition number of the coefficient matrix in the context of direct dense nonsymmetric solves.
Improving the robustness of the solver
Use FGMRES instead of Richardson.

Iterative Refinement in Quadruple Precision on a Intel Xeon 3.2GHz

<table>
<thead>
<tr>
<th>n</th>
<th>time (s)</th>
<th>QGESV</th>
<th>time (s)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.29</td>
<td>0.29</td>
<td>0.03</td>
<td>9.5</td>
</tr>
<tr>
<td>200</td>
<td>2.27</td>
<td>2.27</td>
<td>0.10</td>
<td>20.9</td>
</tr>
<tr>
<td>300</td>
<td>7.61</td>
<td>7.61</td>
<td>0.24</td>
<td>30.5</td>
</tr>
<tr>
<td>400</td>
<td>17.81</td>
<td>17.81</td>
<td>0.44</td>
<td>40.4</td>
</tr>
<tr>
<td>500</td>
<td>34.71</td>
<td>34.71</td>
<td>0.69</td>
<td>49.7</td>
</tr>
<tr>
<td>600</td>
<td>60.11</td>
<td>60.11</td>
<td>1.01</td>
<td>59.0</td>
</tr>
<tr>
<td>700</td>
<td>94.95</td>
<td>94.95</td>
<td>1.38</td>
<td>68.7</td>
</tr>
<tr>
<td>800</td>
<td>141.75</td>
<td>141.75</td>
<td>1.83</td>
<td>77.3</td>
</tr>
<tr>
<td>900</td>
<td>201.81</td>
<td>201.81</td>
<td>2.33</td>
<td>86.3</td>
</tr>
<tr>
<td>1000</td>
<td>276.94</td>
<td>276.94</td>
<td>2.92</td>
<td>94.8</td>
</tr>
</tbody>
</table>

Time for the various Kernels in the Quadruple Accuracy Versions for n=900

<table>
<thead>
<tr>
<th>driver name</th>
<th>time (s)</th>
<th>kernel name</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QGESV</td>
<td>201.81</td>
<td>QGETRF</td>
<td>201.1293</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QGETRS</td>
<td>0.6845</td>
</tr>
<tr>
<td>QDGESV</td>
<td>2.33</td>
<td>DGETRF</td>
<td>0.3200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DGETRS</td>
<td>0.0127</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DLANGE</td>
<td>0.0042</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DGECON</td>
<td>0.0363</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QGEMV</td>
<td>1.5526</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ITERREF</td>
<td>1.9258</td>
</tr>
</tbody>
</table>
Extension to Other Algorithms

• As a rule of thumb, any algorithm with $O(n^3)$ computation and $O(n)$ output can benefit from mixed precision iterative refinement. (Idea: the residual computation needs to be at most $O(n^2)$.)

• This is not the case for eigendecomposition, SVD, matrix-matrix multiplication. This is the case for linear system of equations solve.

Mixed single-double precision solver

Julien Langou, University of Colorado Denver.

Mixed single-double precision solver
Wednesday April 1st, 2008.

1. Mixed single-double precision solver
2. Rectangular Full Packed Format
3. Communication optimal algorithm

![Diagram with colored nodes labeled DGEQRT, DTSQRT, DLARFB, DSSRFB for k=1, k=2, k=3.]
F. G. Gustavson, J. Wasniewski, J. J. Dongarra, and Julien Langou.
Rectangular Full Packed Format for Cholesky’s Algorithm: Factorization, Solution and Inversion.
Case n is even (n= 6), TRANSR = 'N', UPLO = 'L'

\[
\begin{array}{cccccc}
1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 & 4 & 4 \\
5 & 5 & 5 & 5 & 5 & 5 \\
6 & 6 & 6 & 6 & 6 & 6 \\
\end{array}
\]

Case n is odd (n= 7), TRANSR = 'C', UPLO = 'U'

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 & 4 & 4 \\
5 & 5 & 5 & 5 & 5 & 5 \\
6 & 6 & 6 & 6 & 6 & 6 \\
\end{array}
\]

Case n is odd (n= 7), TRANSR = 'L', UPLO = 'U'

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 & 4 & 4 \\
5 & 5 & 5 & 5 & 5 & 5 \\
6 & 6 & 6 & 6 & 6 & 6 \\
\end{array}
\]

Case n is odd (n= 7), TRANSR = 'N', UPLO = 'L'

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 & 4 & 4 \\
5 & 5 & 5 & 5 & 5 & 5 \\
6 & 6 & 6 & 6 & 6 & 6 \\
\end{array}
\]

Case n is odd (n= 7), TRANSR = 'L', UPLO = 'U'

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 & 4 & 4 \\
5 & 5 & 5 & 5 & 5 & 5 \\
6 & 6 & 6 & 6 & 6 & 6 \\
\end{array}
\]

This web page is automatically generated by the following C-code:

```
On the RIGHT: example matrix in Rectangular Full Packed Format (so ZHF, ZSF, ZTF)
On the LEFT: example matrix in Full Format (so ZHE, ZSY, ZTR),
N is ODD or EVEN
EIGHT cases:   N is ODD or EVEN   TRANSR = 'L' or TRANSR = 'C'   UPLO = 'L' or...```
Mixed single-double precision solver

Case n is even (n= 6), TRANSR = 'N', UPLO = 'L'

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( +0,+71)</td>
<td>( +1,+70)</td>
<td>( +2,+69)</td>
<td>( +3,+68)</td>
<td>( +4,+67)</td>
<td>( +5,+66)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +6,+65)</td>
<td>( +7,+64)</td>
<td>( +8,+63)</td>
<td>( +9,+62)</td>
<td>( +10,+61)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +11,+60)</td>
<td>( +12,+59)</td>
<td>( +13,+58)</td>
<td>( +14,+57)</td>
<td>( +15,+56)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +16,+55)</td>
<td>( +17,+54)</td>
<td>( +18,+53)</td>
<td>( +19,+52)</td>
<td>( +20,+51)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +21,+50)</td>
<td>( +22,+49)</td>
<td>( +23,+48)</td>
<td>( +24,+47)</td>
<td>( +25,+46)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +26,+45)</td>
<td>( +27,+44)</td>
<td>( +28,+43)</td>
<td>( +29,+42)</td>
<td>( +30,+41)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +31,+40)</td>
<td>( +32,+39)</td>
<td>( +33,+38)</td>
<td>( +34,+37)</td>
<td>( +35,+36)</td>
<td></td>
</tr>
</tbody>
</table>

Case n is even (n= 6), TRANSR = 'N', UPLO = 'U'

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( +0,+71)</td>
<td>( +1,+70)</td>
<td>( +2,+69)</td>
<td>( +3,+68)</td>
<td>( +4,+67)</td>
<td>( +5,+66)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +6,+65)</td>
<td>( +7,+64)</td>
<td>( +8,+63)</td>
<td>( +9,+62)</td>
<td>( +10,+61)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +11,+60)</td>
<td>( +12,+59)</td>
<td>( +13,+58)</td>
<td>( +14,+57)</td>
<td>( +15,+56)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +16,+55)</td>
<td>( +17,+54)</td>
<td>( +18,+53)</td>
<td>( +19,+52)</td>
<td>( +20,+51)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +21,+50)</td>
<td>( +22,+49)</td>
<td>( +23,+48)</td>
<td>( +24,+47)</td>
<td>( +25,+46)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +26,+45)</td>
<td>( +27,+44)</td>
<td>( +28,+43)</td>
<td>( +29,+42)</td>
<td>( +30,+41)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +31,+40)</td>
<td>( +32,+39)</td>
<td>( +33,+38)</td>
<td>( +34,+37)</td>
<td>( +35,+36)</td>
<td></td>
</tr>
</tbody>
</table>

Case n is even (n= 6), TRANSR = 'C', UPLO = 'L'

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( +0,+71)</td>
<td>( +1,+70)</td>
<td>( +2,+69)</td>
<td>( +3,+68)</td>
<td>( +4,+67)</td>
<td>( +5,+66)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +6,+65)</td>
<td>( +7,+64)</td>
<td>( +8,+63)</td>
<td>( +9,+62)</td>
<td>( +10,+61)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +11,+60)</td>
<td>( +12,+59)</td>
<td>( +13,+58)</td>
<td>( +14,+57)</td>
<td>( +15,+56)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +16,+55)</td>
<td>( +17,+54)</td>
<td>( +18,+53)</td>
<td>( +19,+52)</td>
<td>( +20,+51)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +21,+50)</td>
<td>( +22,+49)</td>
<td>( +23,+48)</td>
<td>( +24,+47)</td>
<td>( +25,+46)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +26,+45)</td>
<td>( +27,+44)</td>
<td>( +28,+43)</td>
<td>( +29,+42)</td>
<td>( +30,+41)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +31,+40)</td>
<td>( +32,+39)</td>
<td>( +33,+38)</td>
<td>( +34,+37)</td>
<td>( +35,+36)</td>
<td></td>
</tr>
</tbody>
</table>

Case n is even (n= 6), TRANSR = 'C', UPLO = 'U'

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( +0,+71)</td>
<td>( +1,+70)</td>
<td>( +2,+69)</td>
<td>( +3,+68)</td>
<td>( +4,+67)</td>
<td>( +5,+66)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +6,+65)</td>
<td>( +7,+64)</td>
<td>( +8,+63)</td>
<td>( +9,+62)</td>
<td>( +10,+61)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +11,+60)</td>
<td>( +12,+59)</td>
<td>( +13,+58)</td>
<td>( +14,+57)</td>
<td>( +15,+56)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +16,+55)</td>
<td>( +17,+54)</td>
<td>( +18,+53)</td>
<td>( +19,+52)</td>
<td>( +20,+51)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +21,+50)</td>
<td>( +22,+49)</td>
<td>( +23,+48)</td>
<td>( +24,+47)</td>
<td>( +25,+46)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +26,+45)</td>
<td>( +27,+44)</td>
<td>( +28,+43)</td>
<td>( +29,+42)</td>
<td>( +30,+41)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( +31,+40)</td>
<td>( +32,+39)</td>
<td>( +33,+38)</td>
<td>( +34,+37)</td>
<td>( +35,+36)</td>
<td></td>
</tr>
</tbody>
</table>
SUBROUTINE DPFRF( TRANSR, UPLO, N, A, INFO )
*
*  -- LAPACK routine (version 3.2)                                    --
*
IF( NISODD ) THEN
  IF( NORMALTRANSR ) THEN
    IF( LOWER ) THEN
      CALL DPOTRF('L', N1, A( 0 ), N, INFO )
      IF( INFO.GT.0 )
        RETURN
      CALL DTRSM('R', 'L', 'T', 'N', N2, N1, ONE, A( 0 ), N,
                  A( N1 ), N )
      CALL DSYRK('U', 'N', N2, N1, -ONE, A( N1 ), N, ONE,
                  A( N ), N )
      CALL DPOTRF('U', N2, A( N ), N, INFO )
      IF( INFO.GT.0 )
        INFO = INFO + N1
      ELSE
      ELSE
      END IF
    ELSE
      CALL DPOTRF('U', N2, A( N ), N, INFO )
      IF( INFO.GT.0 )
        INFO = INFO + N1
      ELSE
      END IF
SUBROUTINE DPFTRI( TRANSR, UPLO, N, A, INFO )
* -- LAPACK routine (version 3.2) --
*
IF( NISODD ) THEN
   IF( NORMALTRANSR ) THEN
      IF( LOWER ) THEN
*        CALL DLAUUM( 'L', N1, A( 0 ), N, INFO )
        CALL DSYRK( 'L', 'T', N1, N2, ONE, A( N1 ), N, ONE,
                +      A( 0 ), N )
        CALL DTRMM( 'L', 'U', 'N', 'N', N2, N1, ONE, A( N ), N,
                +      A( N1 ), N )
        CALL DLAUUM( 'U', N2, A( N ), N, INFO )
*   END IF
END IF
END IF
END IF
*
SUBROUTINE DTFSM( TRANSR, SIDE, UPLO, TRANS, DIAG, M, N, ALPHA, A, B, LDB )

* -- LAPACK routine (version 3.2) --
*

IF( MISODD ) THEN
  IF( NORMALTRANSR ) THEN
    IF( LOWER ) THEN
      IF( NOTRANS ) THEN
        CALL DTRSM( 'L', 'L', 'N', DIAG, M1, N, ALPHA, A( 0 ), M, B, LDB )
        CALL DGEMM( 'N', 'N', M2, N, M1, -ONE, A( M1 ), M, B, LDB, ALPHA, B( M1, 0 ), LDB )
        CALL DTRSM( 'L', 'U', 'T', DIAG, M2, N, ONE, A( M ), M, B( M1, 0 ), LDB )
      *
    END IF
  END IF
END IF
END IF
*

Mixed single-double precision solver

Performance in Mflop/s of Cholesky Factorization/Inversion/Solution on SUN UltraSPARC IV+ computer.

long real arithmetic.

Performance of Cholesky Factorization on SUN UltraSPARC IV+ computer, long real arithmetic

Performance of Cholesky Inversion on SUN UltraSPARC IV+ computer, long real arithmetic

Performance of Cholesky Solution on SUN UltraSPARC IV+ computer, long real arithmetic

long complex arithmetic.

Performance of Cholesky Factorization on SUN UltraSPARC IV+ computer, long complex arithmetic

Performance of Cholesky Inversion on SUN UltraSPARC IV+ computer, long complex arithmetic

Performance of Cholesky Solution on SUN UltraSPARC IV+ computer, long complex arithmetic
Mixed single-double precision solver

Performance in Mflop/s of Cholesky Factorization/Inversion/Solution on SGI Altix 3700, Intel Itanium 2 computer. long real arithmetic.

Performance in Mflop/s of Cholesky Factorization/Inversion/Solution on SGI Altix 3700, Intel Itanium 2 computer. long complex arithmetic.
Mixed single-double precision solver

Performance in Mflop/s of Cholesky Factorization/Inversion/Solution on ia64 Itanium computer.
Performance in Mflop/s of Cholesky Factorization/Inversion/Solution on SX-6 NEC computer.
Performance of Cholesky Factorization/Inversion/Solution on quad-socket quad-core Intel Tigerton computer. We use reference LAPACK-3.2.0 (from netlib) and MKL-10.0.1.014 multithreaded BLAS. For the solution phase, \( nrhs \) is fixed to 100 for any \( n \). Due to time limitation, the experiment was stopped for the packed storage format inversion at \( n = 4000 \).
Mixed single-double precision solver

Performance of Cholesky Factorization/Inversion/Solution on quad-socket quad-core Intel Tigerton computer. We use MKL-10.0.1.014 multithreaded LAPACK and BLAS. For the solution phase, \( nrhs \) is fixed to 100 for any \( n \). Due to time limitation, the experiment was stopped for the packed storage format inversion at \( n = 4000 \).
Performance in Gflop/s of Cholesky Factorization on IBM Power 4 (left) and SUN UltraSPARC-IV (right) computer with a different number of Processors, testing the SMP Parallelism. The implementation of PPTRF of sunperf does not show any SMP parallelism. UPLO = ’L’. $N = 5,000$ (strong scaling experiment).
Julien Langou, University of Colorado Denver.

Mixed single-double precision solver
Wednesday April 1st, 2008.

1. Mixed single-double precision solver
2. Rectangular Full Packed Format
3. Communication optimal algorithm
For more information:


Mixed single-double precision solver

1. TSQR: Tall Skinny QR
2. CAQR: Communication Avoiding QR
1. **TSQR: Tall Skinny QR**

2. **CAQR: Communication Avoiding QR**
AllReduce Algorithms: Application to Householder QR Factorization

Jim Demmel, University of California, Berkeley;
Laura Grigori, INRIA, France;
Mark Hoemmen, University of California, Berkeley;
Julien Langou, University of Colorado, Denver
Reduce Algorithms: Introduction

The QR factorization of a long and skinny matrix with its data partitioned vertically across several processors arises in a wide range of applications.

**Input:**
A is block distributed by rows

**Output:**
Q is block distributed by rows
R is global
Reduce Algorithms: Introduction

Example of applications:

a) in linear least squares problems which the number of equations is extremely larger than the number of unknowns
Reduce Algorithms: Introduction

Example of applications:

a) in linear least squares problems which the number of equations is extremely larger than the number of unknowns

b) in block iterative methods (iterative methods with multiple right-hand sides or iterative eigenvalue solvers)
Example of applications: in block iterative methods.

a) in iterative methods with multiple right-hand sides (block iterative methods:)
   1) Trilinos (Sandia National Lab.) through Belos (R. Lehoucq, H. Thornquist, U. Hetmaniuk).
   2) BlockGMRES, BlockGCR, BlockCG, BlockQMR, ...

b) in iterative methods with a single right-hand side
   1) s-step methods for linear systems of equations (e.g. A. Chronopoulos),
   2) LGMRES (Jessup, Baker, Dennis, U. Colorado at Boulder) implemented in PETSc,
   3) Recent work from M. Hoemmen and J. Demmel (U. California at Berkeley).

e) in iterative eigenvalue solvers,
   1) PETSc (Argonne National Lab.) through BLOPEX (A. Knyazev, UCDHSC),
   2) HYPRE (Lawrence Livermore National Lab.) through BLOPEX,
   3) Trilinos (Sandia National Lab.) through Anasazi (R. Lehoucq, H. Thornquist, U. Hetmaniuk),
   4) PRIMME (A. Stathopoulos, Coll. William & Mary ),
   5) And also TRLAN, BLZPACK, IRBLEIGS.
Reduce Algorithms: Introduction

Example of applications:

a) in linear least squares problems which the number of equations is extremely larger than the number of unknowns

b) in block iterative methods (iterative methods with multiple right-hand sides or iterative eigenvalue solvers)

c) in dense large and more square QR factorization where they are used as the panel factorization step
Blocked LU and QR algorithms (LAPACK)

LAPACK block LU (right-looking): dgetrf

- Panel factorization
  - dgetf2
  - dtrsm (+ dswap)
  - dgemm

LAPACK block QR (right-looking): dgeqrf

- Panel factorization
  - dgeqf2 + dlarft
  - dlarfb
### Blocked LU and QR algorithms (LAPACK)

<table>
<thead>
<tr>
<th>LAPACK block LU (right-looking): dgetrf</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram of LAPACK block LU" /></td>
</tr>
</tbody>
</table>

**Panel factorization**
- `dgetf2`
- `lu( )`

Latency bounded: more than nb AllReduce for n*nb^2 ops

**Update of the remaining submatrix**
- `dtrsm (+ dswap)`
- `dgemm`

CPU - bandwidth bounded: the bulk of the computation: n*n*nb ops:
- Highly parallelizable, efficient and scalable.
Mixed single-double precision solver

Parallelization of LU and QR.

Parallelize the update:

- Easy and done in any reasonable software.
- This is the 2/3n^3 term in the FLOPs count.
- Can be done efficiently with LAPACK+multithreaded BLAS

\[
\begin{align*}
&U \\
&L \\
&A^{(1)} \\
&\text{dgetf2} \\
&\text{lu}() \\
&\text{dtrsm (+ dswap)} \\
&\text{dgemm} \\
&\text{dgemm} \\
&\text{L} \\
&\text{U} \\
&A^{(2)} \\
\end{align*}
\]
## Parallelization of LU and QR.

### Parallelize the update:
- Easy and done in any reasonable software.
- This is the $2/3n^3$ term in the FLOPs count.
- Can be done efficiently with LAPACK+multithreaded BLAS.

### Parallelize the panel factorization:
- Not an option in multicore context ($p < 16$)
- See e.g. ScALAPACK or HPL but still by far the slowest and the bottleneck of the computation.

### Hide the panel factorization:
- Lookahead (see e.g. High Performance LINPACK)
- Dynamic Scheduling
Mixed single-double precision solver

Hiding the panel factorization with dynamic scheduling.

Courtesy from Alfredo Buttari, UTennessee
What about strong scalability?
What about strong scalability?

N = 1536
NB = 64
procs = 16

Courtesy from Jakub Kurzak, UTennessee
Mixed single-double precision solver

What about strong scalability?

N = 1536
NB = 64
procs = 16

We can not hide the panel factorization in the MM, actually it is the MMs that are hidden by the panel factorizations!

Courtesy from Jakub Kurzak, UTennessee
What about strong scalability?

\[ N = 1536 \]
\[ NB = 64 \]
\[ \text{procs} = 16 \]

We can not hide the panel factorization \( (n^2) \) with the \( \text{MM}(n^3) \), actually it is the MMs that are hidden by the panel factorizations!

**NEED FOR NEW MATHEMATICAL ALGORITHMS**

Courtesy from Jakub Kurzak, UTennessee
A new generation of algorithms?

<table>
<thead>
<tr>
<th>Algorithms follow hardware evolution along time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINPACK (80’s)  (Vector operations)</td>
</tr>
<tr>
<td>LAPACK (90’s)  (Blocking, cache friendly)</td>
</tr>
<tr>
<td>Rely on - Level-1 BLAS operations</td>
</tr>
<tr>
<td>Rely on - Level-3 BLAS operations</td>
</tr>
</tbody>
</table>
A new generation of algorithms?

Algorithms follow hardware evolution along time.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Rely on</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINPACK (80’s)</td>
<td>- Level-1 BLAS operations</td>
</tr>
<tr>
<td>(Vector operations)</td>
<td></td>
</tr>
<tr>
<td>LAPACK (90’s)</td>
<td>- Level-3 BLAS operations</td>
</tr>
<tr>
<td>(Blocking, cache friendly)</td>
<td></td>
</tr>
<tr>
<td>New Algorithms (00’s)</td>
<td>- a DAG/scheduler</td>
</tr>
<tr>
<td>(multicore friendly)</td>
<td>- block data layout</td>
</tr>
<tr>
<td></td>
<td>- some extra kernels</td>
</tr>
</tbody>
</table>

Those new algorithms
- have a very low granularity, they scale very well (multicore, petascale computing, ...)
- removes a lots of dependencies among the tasks, (multicore, distributed computing)
- avoid latency (distributed computing, out-of-core)
- rely on fast kernels

Those new algorithms need new kernels and rely on efficient scheduling algorithms.
2005-2007: New algorithms based on 2D partitionning:

- UTexas (van de Geijn): SYRK, CHOL (multicore), LU, QR (out-of-core)
- UTennessee (Dongarra): CHOL (multicore)
- HPC2N (Kågström)/IBM (Gustavson): Chol (Distributed)
- UCBerkeley (Demmel)/INRIA(Grigori): LU/QR (distributed)
- UCDenver (Langou): LU/QR (distributed)

A 3\textsuperscript{rd} revolution for dense linear algebra?
Reduce Algorithms: Introduction

Example of applications:

a) in block iterative methods (iterative methods with multiple right-hand sides or iterative eigenvalue solvers),
b) in dense large and more square QR factorization where they are used as the panel factorization step, or more simply
c) in linear least squares problems which the number of equations is extremely larger than the number of unknowns.

The main characteristics of those three examples are that

a) there is only one column of processors involved but several processor rows,
b) all the data is known from the beginning,
c) and the matrix is dense.
Reduce Algorithms: Introduction

Example of applications:

a) in block iterative methods (iterative methods with multiple right-hand sides or
iterative eigenvalue solvers),
b) in dense large and more square QR factorization where they are used as the panel
factorization step, or more simply
c) in linear least squares problems which the number of equations is extremely larger
than the number of unknowns.

The main characteristics of those three examples are that

a) there is only one column of processors involved but several processor rows,
b) all the data is known from the beginning,
c) and the matrix is dense.

Various methods already exist to perform the QR factorization of such matrices:

a) Gram-Schmidt (mgs(row),cgs),
b) Householder (qr2, qrf),
c) or CholeskyQR.

We present a new method:

Allreduce Householder (rhh_qr3, rhh_qrf).
The CholeskyQR Algorithm

SYRK: \[ C := A^T A \] (\( mn^2 \))

CHOL: \[ R := \text{chol}( C ) \] (\( n^3/3 \))

TRSM: \[ Q := A / R \] (\( mn^2 \))
Bibliography


• Popularized by iterative eigensolver libraries:
  1) PETSc (Argonne National Lab.) through BLOPEX (A. Knyazev, UCDHSC),
  2) HYPRE (Lawrence Livermore National Lab.) through BLOPEX,
  3) Trilinos (Sandia National Lab.) through Anasazi (R. Lehoucq, H. Thornquist, U. Hetmaniuk),
  4) PRIMME (A. Stathopoulos, Coll. William & Mary).
Parallel distributed CholeskyQR

The CholeskyQR method in the parallel distributed context can be described as follows:

1: SYRK: \( C := A^T A \) (\( mn^2 \))
2: MPI_Reduce: \( C := \sum_{\text{procs}} C \) (on proc 0)
3: CHOL: \( R := \text{chol}(C) \) (\( n^3/3 \))
4: MPI_Bdcast: Broadcast the R factor on proc 0 to all the other processors
5: TRSM: \( Q := A/R \) (\( mn^2 \))
In this experiment, we fix the problem: \( m=100,000 \) and \( n=50 \).

<table>
<thead>
<tr>
<th># of procs</th>
<th>cholqr</th>
<th>cgls</th>
<th>mgs(row)</th>
<th>qrfs</th>
<th>mgs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>489.2</td>
<td>(1.02)</td>
<td>134.1</td>
<td>(3.73)</td>
<td>73.5</td>
</tr>
<tr>
<td>2</td>
<td>467.3</td>
<td>(0.54)</td>
<td>78.9</td>
<td>(3.17)</td>
<td>39.0</td>
</tr>
<tr>
<td>4</td>
<td>466.4</td>
<td>(0.27)</td>
<td>71.3</td>
<td>(1.75)</td>
<td>38.7</td>
</tr>
<tr>
<td>8</td>
<td>434.0</td>
<td>(0.14)</td>
<td>67.4</td>
<td>(0.93)</td>
<td>36.7</td>
</tr>
<tr>
<td>16</td>
<td>359.2</td>
<td>(0.09)</td>
<td>54.2</td>
<td>(0.58)</td>
<td>31.6</td>
</tr>
<tr>
<td>32</td>
<td>197.8</td>
<td>(0.08)</td>
<td>41.9</td>
<td>(0.37)</td>
<td>29.0</td>
</tr>
</tbody>
</table>
Simple enough?

```c
int choleskyqr_A_v0(int mloc, int n, double *A, int lda, double *R, int ldr, MPI_Comm mpi_comm){
    int info;
    cblas_dsyrk( CblasColMajor, CblasUpper, CblasTrans, n, mloc,
                1.0e+00, A, lda, 0e+00, R, ldr );
    MPI_Allreduce( MPI_INPLACE, R, n*n, MPI_DOUBLE, MPI_SUM, mpi_comm );
    lapack_dpotrf( lapack_upper, n, R, ldr, &info );
    cblas_dtrsm( CblasColMajor, CblasRight, CblasUpper, CblasNoTrans, CblasNonUnit,
                mloc, n, 1.0e+00, R, ldr, A, lda );
    return 0;
}
```

(... and, OK, you might want to add an MPI user defined datatype to send only the upper part of R)
Mixed single-double precision solver

Stable enough?

\[ \frac{\| A - QR \|_2}{\| A \|_2} \]

\[ \frac{\| I - Q^TQ \|_2}{\| A \|_2} \]

\( \kappa_2(A) \)

\( m=100, \ n=50 \)
The CholeskyQR method in the parallel distributed context can be described as follows:

1: SYRK: \( C := A^T A \) (\( mn^2 \))
2: MPI_Reduce: \( C := \sum_{\text{procs}} C \) (on proc 0)
3: CHOL: \( R := \text{chol}(C) \) (\( n^3/3 \))
4: MPI_Bdcast: Broadcast the \( R \) factor on proc 0 to all the other processors
5: TRSM: \( Q := A/R \) (\( mn^2 \))

This method is extremely fast. For two reasons:
1. first, there is only one or two communications phase,
2. second, the local computations are performed with fast operations.

Another advantage of this method is that the resulting code is exactly four lines,
3. so the method is simple and relies heavily on other libraries.
Despite all those advantages,
4. this method is highly unstable.
Reduce Algorithms

The gather-scatter variant of our algorithm can be summarized as follows:

1. perform local QR factorization of the matrix $A$
2. gather the $p$ R factors on processor 0
3. perform a QR factorization of all the R put the ones on top of the others, the R factor obtained is the R factor
4. scatter the the Q factors from processor 0 to all the processors
5. multiply locally the two Q factors together, done.
Reduce Algorithms

• This is the scatter-gather version of our algorithm.

• This variant is not very efficient for two reasons:
  – first the communication phases 2 and 4 are highly involving processor 0;
  – second the cost of step 3 is $p/3*n^3$, so can get prohibitive for large $p$.

• Note that the CholeskyQR algorithm can also be implemented in a scatter-gather way but reduce-broadcast. This leads naturally to the algorithm presented below where a reduce-broadcast version of the previous algorithm is described. This will be our final algorithm.
On two processes

Mixed single-double precision solver
Mixed single-double precision solver

On two processes

\[ A_0 \rightarrow V_0^{(0)} \rightarrow R_0^{(0)} \]

\[ A_1 \rightarrow V_1^{(0)} \rightarrow R_1^{(0)} \]
On two processes
Mixed single-double precision solver

On two processes

QR ( ) → ( )

A₀ V₀(0)

QR ( ) → ( )

R₀(0) R₀(1)
V₀(1) V₁(1)

QR ( ) → ( )

A₁ V₁(0)

QR ( ) → ( )

R₁(0)
On two processes
On two processes
On two processes

Mixed single-double precision solver
The big picture ....
Mixed single-double precision solver

The big picture ....

A process communication
The big picture ....

Mixed single-double precision solver
Mixed single-double precision solver

The big picture ....
Mixed single-double precision solver

The big picture ....
Mixed single-double precision solver

The big picture ....

A0
A1
A2
A3
A4
A5
A6

4 me processes

communication

computation
Mixed single-double precision solver

The big picture ....
Mixed single-double precision solver

The big picture ....

[Diagram showing processes and communication]
Mixed single-double precision solver

The big picture ....
The big picture ....
Latency but also possibility of fast panel factorization.

- **DGEQR3** is the recursive algorithm (see Elmroth and Gustavson, 2000), **DGEQRF** and **DGEQR2** are the LAPACK routines.
- Times include QR and DLARFT.
- Run on Pentium III.

<table>
<thead>
<tr>
<th>n</th>
<th>DGEQR3</th>
<th>DGEQRF</th>
<th>DGEQR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>173.6</td>
<td>65.0</td>
<td>64.6</td>
</tr>
<tr>
<td>100</td>
<td>240.5</td>
<td>62.6</td>
<td>65.3</td>
</tr>
<tr>
<td>150</td>
<td>277.9</td>
<td>81.6</td>
<td>64.2</td>
</tr>
<tr>
<td>200</td>
<td>312.5</td>
<td>111.3</td>
<td>65.9</td>
</tr>
</tbody>
</table>

**QR factorization and construction of T**

- **m = 10,000**
- Perf in MFLOP/sec (Times in sec)

For **m=1000000**, the x-axis is n.
When only $R$ is wanted
When only $R$ is wanted: The MPI_Allreduce

In the case where only $R$ is wanted, instead of constructing our own tree, one can simply use MPI_Allreduce with a user defined operation. The operation we give to MPI is basically the Algorithm 2. It performs the operation:

$$QR \begin{pmatrix} R_1 \\ R_2 \end{pmatrix} \rightarrow R$$

This binary operation is associative and this is all MPI needs to use a user-defined operation on a user-defined datatype. Moreover, if we change the signs of the elements of $R$ so that the diagonal of $R$ holds positive elements then the binary operation $Rfactor$ becomes commutative.

The code becomes two lines:

```c
lapack_dgeqrf( mloc, n, A, lda, tau, &dlwork, lwork, &info );
MPI_Allreduce( MPI_IN_PLACE, A, 1, MPI_UPPER, LILA_MPIOP_QR_UPPER, mpi_comm);
```
Mixed single-double precision solver

Does it work?

• The experiments are performed on the beowulf cluster at the University of Colorado at Denver. The cluster is made of 35 bi-pro Pentium III (900MHz) connected with Dolphin interconnect.
• Number of operations is taken as $2mn^2$ for all the methods
• The block size used in ScaLAPACK is 32.
• The code is written in C, use MPI (mpich-2.1), LAPACK (3.1.1), BLAS (goto-1.10), the LAPACK Cwrappers (http://icl.cs.utk.edu/~delmas/lapwrapmw.htm ) and the BLAS C wrappers (http://www.netlib.org/blas/blast-forum/cblas.tgz)
• The codes has been tested in various configuration and have never failed to produce a correct answer, releasing those codes is in the agenda

### Number of operations is taken as $2mn^2$ for all the methods

<table>
<thead>
<tr>
<th>Method</th>
<th>FLOPs (total) for R only</th>
<th>FLOPs (total) for Q and R</th>
</tr>
</thead>
<tbody>
<tr>
<td>CholeskyQR</td>
<td>$mn^2 + \frac{n^3}{3}$</td>
<td>$2mn^2 + \frac{n^3}{3}$</td>
</tr>
<tr>
<td>Gram-Schmidt</td>
<td>$2mn^2$</td>
<td>$2mn^2$</td>
</tr>
<tr>
<td>Householder</td>
<td>$2mn^2 - \frac{2}{3}n^3$</td>
<td>$4mn^2 - 4\frac{3n^3}{3}$</td>
</tr>
<tr>
<td>Allreduce HH</td>
<td>$(2mn^2 - \frac{2}{3}n^3) + \frac{2}{3}n^3 p$</td>
<td>$(4mn^2 - 4\frac{3n^3}{3}) + 4\frac{3n^3}{3} p$</td>
</tr>
</tbody>
</table>
**Q and R: Strong scalability**

- In this experiment, we fix the problem: $m=100,000$ and $n=50$. Then we increase the number of processors.

- Once more the algorithm $\text{rhh}_\text{qr3}$ is the second behind $\text{CholeskyQR}$. Note that $\text{rhh}_\text{qr3}$ is incondionnally stable while the stability of $\text{CholeskyQR}$ depends on the square of the condition number of the initial matrix.

<table>
<thead>
<tr>
<th># of procs</th>
<th>cholqr</th>
<th>rhh_qr3</th>
<th>cgs</th>
<th>mgs(row)</th>
<th>rhh_qr</th>
<th>qr</th>
<th>qr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>489.2</td>
<td>120.0</td>
<td>134.1</td>
<td>73.5</td>
<td>51.9</td>
<td>39.1</td>
<td>34.3</td>
</tr>
<tr>
<td>2</td>
<td>467.3</td>
<td>100.8</td>
<td>78.9</td>
<td>39.0</td>
<td>31.2</td>
<td>22.3</td>
<td>20.2</td>
</tr>
<tr>
<td>4</td>
<td>466.4</td>
<td>97.9</td>
<td>71.3</td>
<td>38.7</td>
<td>31.0</td>
<td>22.2</td>
<td>18.8</td>
</tr>
<tr>
<td>8</td>
<td>434.0</td>
<td>95.9</td>
<td>67.4</td>
<td>36.7</td>
<td>34.0</td>
<td>20.8</td>
<td>17.7</td>
</tr>
<tr>
<td>16</td>
<td>359.2</td>
<td>103.8</td>
<td>54.2</td>
<td>31.6</td>
<td>27.8</td>
<td>18.3</td>
<td>16.3</td>
</tr>
<tr>
<td>32</td>
<td>197.8</td>
<td>84.9</td>
<td>41.9</td>
<td>29.0</td>
<td>33.3</td>
<td>15.8</td>
<td>14.5</td>
</tr>
</tbody>
</table>

- The table above shows the MFLOP/sec/proc and Time in sec for different algorithms and the number of processors.
Q and R: Weak scalability with respect to m

- We fix the local size to be $m_{loc}=100,000$ and $n=50$. When we increase the number of processors, the global $m$ grows proportionally.

- $rhh\_qr3$ is the Allreduce algorithm with recursive panel factorization, $rhh\_qrf$ is the same with LAPACK Householder QR. We see the obvious benefit of using recursion. See as well (6). qr2 and qrf correspond to the ScALAPACK Householder QR factorization routines.

<table>
<thead>
<tr>
<th># of procs</th>
<th>cholqr</th>
<th>rhh_qr3</th>
<th>Cgs</th>
<th>mgs(row)</th>
<th>rhh_qrf</th>
<th>qrf</th>
<th>qr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>489.2</td>
<td>(1.02)</td>
<td>121.2</td>
<td>(4.13)</td>
<td>135.7</td>
<td>(3.69)</td>
<td>70.2</td>
</tr>
<tr>
<td>2</td>
<td>466.9</td>
<td>(1.07)</td>
<td>102.3</td>
<td>(4.89)</td>
<td>84.4</td>
<td>(5.93)</td>
<td>35.6</td>
</tr>
<tr>
<td>4</td>
<td>454.1</td>
<td>(1.10)</td>
<td>96.7</td>
<td>(5.17)</td>
<td>67.2</td>
<td>(7.44)</td>
<td>41.4</td>
</tr>
<tr>
<td>8</td>
<td>458.7</td>
<td>(1.09)</td>
<td>96.2</td>
<td>(5.20)</td>
<td>67.1</td>
<td>(7.46)</td>
<td>33.2</td>
</tr>
<tr>
<td>16</td>
<td>451.3</td>
<td>(1.11)</td>
<td>94.8</td>
<td>(5.27)</td>
<td>67.2</td>
<td>(7.45)</td>
<td>33.3</td>
</tr>
<tr>
<td>32</td>
<td>442.1</td>
<td>(1.13)</td>
<td>94.6</td>
<td>(5.29)</td>
<td>62.8</td>
<td>(7.97)</td>
<td>32.5</td>
</tr>
<tr>
<td>64</td>
<td>414.9</td>
<td>(1.21)</td>
<td>93.0</td>
<td>(5.38)</td>
<td>62.8</td>
<td>(7.96)</td>
<td>32.3</td>
</tr>
</tbody>
</table>
Mixed single-double precision solver

Q and R: Weak scalability with respect to n

• We fix the global size $m=100,000$ and then we increase $n$ as $\sqrt{p}$ so that the workload $mn^2$ per processor remains constant.

• Due to better performance in the local factorization or SYRK, CholeskyQR, rhh_qr3 and rhh_qrf exhibit increasing performance at the beginning until the $n^3$ comes into play.

<table>
<thead>
<tr>
<th># of procs</th>
<th>cholqr</th>
<th>rhh_qr3</th>
<th>cgs</th>
<th>mgs(row)</th>
<th>rhh_qrf</th>
<th>qr</th>
<th>qr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>490.7</td>
<td>120.8</td>
<td>134.0</td>
<td>69.7</td>
<td>51.7</td>
<td>39.6</td>
<td>39.9</td>
</tr>
<tr>
<td>2</td>
<td>510.2</td>
<td>126.0</td>
<td>78.6</td>
<td>40.1</td>
<td>32.1</td>
<td>25.4</td>
<td>19.0</td>
</tr>
<tr>
<td>4</td>
<td>541.1</td>
<td>149.4</td>
<td>75.6</td>
<td>39.1</td>
<td>31.1</td>
<td>25.5</td>
<td>18.9</td>
</tr>
<tr>
<td>8</td>
<td>540.2</td>
<td>173.8</td>
<td>72.3</td>
<td>38.5</td>
<td>43.6</td>
<td>27.8</td>
<td>20.2</td>
</tr>
<tr>
<td>16</td>
<td>501.5</td>
<td>195.2</td>
<td>66.8</td>
<td>38.4</td>
<td>51.3</td>
<td>28.9</td>
<td>19.3</td>
</tr>
<tr>
<td>32</td>
<td>379.2</td>
<td>177.4</td>
<td>59.8</td>
<td>36.2</td>
<td>61.4</td>
<td>29.5</td>
<td>19.3</td>
</tr>
<tr>
<td>64</td>
<td>266.4</td>
<td>83.9</td>
<td>32.3</td>
<td>36.1</td>
<td>52.9</td>
<td>28.2</td>
<td>18.4</td>
</tr>
</tbody>
</table>

MFLOP/sec/proc

Time in sec
In this experiment, we fix the problem: $m=100,000$ and $n=50$. Then we increase the number of processors.
R only: Weak scalability with respect to m

- We fix the local size to be $m_{loc}=100,000$ and $n=50$. When we increase the number of processors, the global $m$ grows proportionally.

```
<table>
<thead>
<tr>
<th># of procs</th>
<th>cholqr</th>
<th>rhh_qr3</th>
<th>cgs</th>
<th>mgs(row)</th>
<th>rhh_qrf</th>
<th>qrf</th>
<th>qr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1098.7</td>
<td>(0.45)</td>
<td>145.4</td>
<td>(3.43)</td>
<td>138.2</td>
<td>(3.61)</td>
<td>70.2</td>
</tr>
<tr>
<td>2</td>
<td>1048.3</td>
<td>(0.47)</td>
<td>124.3</td>
<td>(4.02)</td>
<td>70.3</td>
<td>(7.11)</td>
<td>35.6</td>
</tr>
<tr>
<td>4</td>
<td>1044.0</td>
<td>(0.47)</td>
<td>116.5</td>
<td>(4.29)</td>
<td>82.0</td>
<td>(6.09)</td>
<td>41.4</td>
</tr>
<tr>
<td>8</td>
<td>993.9</td>
<td>(0.50)</td>
<td>116.2</td>
<td>(4.30)</td>
<td>66.3</td>
<td>(7.53)</td>
<td>33.2</td>
</tr>
<tr>
<td>16</td>
<td>918.7</td>
<td>(0.54)</td>
<td>115.2</td>
<td>(4.33)</td>
<td>64.1</td>
<td>(7.79)</td>
<td>33.3</td>
</tr>
<tr>
<td>32</td>
<td>950.7</td>
<td>(0.52)</td>
<td>112.9</td>
<td>(4.42)</td>
<td>63.6</td>
<td>(7.85)</td>
<td>32.5</td>
</tr>
<tr>
<td>64</td>
<td>764.6</td>
<td>(0.65)</td>
<td>112.3</td>
<td>(4.45)</td>
<td>62.7</td>
<td>(7.96)</td>
<td>32.3</td>
</tr>
</tbody>
</table>
```
Mixed single-double precision solver

Q and R: Strong scalability

In this experiment, we fix the problem: \(m=1,000,000\) and \(n=50\).
Then we increase the number of processors.

Blue Gene L
frost.ncar.edu

![Graph showing MFLOPs/sec/proc vs. number of processors](image)

- ReduceHH (QR3)
- ReduceHH (QRF)
- ScaLAPACK QRF
- ScaLAPACK QR2
Conclusions

We have described a new method for the Householder QR factorization of skinny matrices. The method is named Allreduce Householder and has four advantages:

1. there is only one synchronization point in the algorithm,
2. the method harvests most of efficiency of the computing unit by large local operations,
3. the method is stable,
4. and finally the method is elegant in particular in the case where only R is needed.

Allreduce algorithms have been depicted here with Householder QR factorization. However it can be applied to anything for example Gram-Schmidt or LU.

Current development is in writing a 2D block cyclic QR factorization and LU factorization based on those ideas.
1. TSQR: Tall Skinny QR
2. CAQR: Communication Avoiding QR
Mixed single-double precision solver

Mixed single-double precision solver

for (k = 0; k < TILES; k++) {
    dgeqrt(A[k][k], T[k][k]);
    for (n = k+1; n < TILES; n++) {
        dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
            dtsqrt(A[k][k], A[m][k], T[m][k]);
            for (n = k+1; n < TILES; n++)
                dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
    }
}
1. `dgeqrt(A[0][0], T[0][0]);`

```c
for (k = 0; k < TILES; k++) {
    dgeqrt(A[k][k], T[k][k]);
    for (n = k+1; n < TILES; n++) {
        dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
            dtsqrt(A[k][k], A[m][k], T[m][k]);
            for (n = k+1; n < TILES; n++)
                dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
    }
}
```
Mixed single-double precision solver

for (k = 0; k < TILES; k++) {
    dgeqrt(A[k][k], T[k][k]);
    for (n = k+1; n < TILES; n++) {
        dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
            dtsqrt(A[k][k], A[m][k], T[m][k]);
            for (n = k+1; n < TILES; n++)
                dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
    }
}
Mixed single-double precision solver

for (k = 0; k < TILES; k++) {
    dgeqrt(A[k][k], T[k][k]);
    for (n = k+1; n < TILES; n++) {
        dlarfb(A[k][k], T[k][k], A[k][n]);
    }
    for (m = k+1; m < TILES; m++) {
        dtsqrt(A[k][k], A[m][k], T[m][k]);
        for (n = k+1; n < TILES; n++)
            dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
    }
}
Mixed single-double precision solver

1. \texttt{dgeqrt}(A[0][0], T[0][0]);
2. \texttt{dlarfb}(A[0][0], T[0][0], A[0][1]);

\begin{verbatim}
for (k = 0; k < TILES; k++) {
    \texttt{dgeqrt}(A[k][k], T[k][k]);
    for (n = k+1; n < TILES; n++) {
        \texttt{dlarfb}(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
            \texttt{dtsqrt}(A[m][k], A[m][k], T[m][k]);
            for (n = k+1; n < TILES; n++)
                \texttt{dssrfb}(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
    }
}
\end{verbatim}
Mixed single-double precision solver

for (k = 0; k < TILES; k++) {
    dgeqrt(A[k][k], T[k][k]);
    for (n = k+1; n < TILES; n++) {
        dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
            dtsqrt(A[k][k], A[m][k], T[m][k]);
            for (n = k+1; n < TILES; n++)
                dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
    }
}
Mixed single-double precision solver

for (k = 0; k < TILES; k++) {
    dgeqrt(A[k][k], T[k][k]);
    for (n = k+1; n < TILES; n++) {
        dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
            dtsqrt(A[k][k], A[m][k], T[m][k]);
            for (n = k+1; n < TILES; n++)
            dssrff(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
    }
}
for (k = 0; k < TILES; k++) {
    dgeqrt(A[k][k], T[k][k]);
    for (n = k+1; n < TILES; n++) {
        dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
            dtsqrt(A[k][k], A[m][k], T[m][k]);
            for (n = k+1; n < TILES; n++)
                dssrft(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
    }
}

1. dgeqrt(A[0][0], T[0][0]);
2. dlarfb(A[0][0], T[0][0], A[0][1]);
3. dlarfb(A[0][0], T[0][0], A[0][2]);
4. dlarfb(A[0][0], T[0][0], A[0][3]);
5. dtsqrt(A[0][0], A[1][0], T[1][0]);
for (k = 0; k < TILES; k++) {
    dgeqrt(A[k][k], T[k][k]);
    for (n = k+1; n < TILES; n++) {
        dlarf(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
            dtsqrt(A[k][k], A[m][k], T[m][k]);
            for (n = k+1; n < TILES; n++)
                dssrff(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
    }
}
Mixed single-double precision solver

for (k = 0; k < TILES; k++) {
    dgeqrt(A[k][k], T[k][k]);
    for (n = k+1; n < TILES; n++) {
        dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
            dtsqrt(A[k][k], A[m][k], T[m][k]);
            for (n = k+1; n < TILES; n++)
                dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
    }
}

4. dlarfb(A[0][0], T[0][0], A[0][3]);
5. dtsqrt(A[0][0], A[1][0], T[1][0]);
6. dssrfb(A[1][0], T[1][0], A[0][1], A[1][1]);
for (k = 0; k < TILES; k++) {
    dgeqrt(A[k][k], T[k][k]);
    for (n = k+1; n < TILES; n++) {
        dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
            dtsqrt(A[k][k], A[m][k], T[m][k]);
            for (n = k+1; n < TILES; n++)
                dssrffb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
    }
}

4. dlarfb(A[0][0], T[0][0], A[0][3]);
5. dtsqrt(A[0][0], A[1][0], T[1][0]);
6. dssrffb(A[1][0], T[1][0], A[0][1], A[1][1]);
for (k = 0; k < TILES; k++) {
    dgeqrt(A[k][k], T[k][k]);
    for (n = k+1; n < TILES; n++) {
        dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
            dtsqrt(A[k][k], A[m][k], T[m][k]);
            for (n = k+1; n < TILES; n++)
                dssrfrb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
    }
}

...
4. dlarfb(A[0][0], T[0][0], A[0][3]);
5. dtsqrt(A[0][0], A[1][0], T[1][0]);
6. dssrfrb(A[1][0], T[1][0], A[0][1], A[1][1]);
7. dssrfrb(A[1][0], T[1][0], A[0][1], A[1][1]);
8. dssrfrb(A[1][0], T[1][0], A[0][1], A[1][1]);
for (k = 0; k < TILES; k++) {
    dgeqrt(A[k][k], T[k][k]);
    for (n = k+1; n < TILES; n++) {
        dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++) {
            dtsqrt(A[k][k], A[m][k], T[m][k]);
            for (n = k+1; n < TILES; n++)
                dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
    }
}
Mixed single-double precision solver

for (k = 0; k < TILES; k++) {
    dgeqrt(A[k][k], T[k][k]);
    for (n = k+1; n < TILES; n++) {
        dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
            dtsqrt(A[k][k], A[m][k], T[m][k]);
            for (n = k+1; n < TILES; n++)
                dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
    }
}

...
Mixed single-double precision solver

\[ k=1 \quad k=2 \quad k=3 \]

- DGEQRT
- DTSQRT
- DLARFB
- DSSRFB
Mixed single-double precision solver

Execution of the parallel TSQR factorization on a binary tree of four processors. The gray boxes indicate where local QR factorizations take place. The $Q$ and $R$ factors each have two subscripts: the first is the sequence number within that stage, and the second is the stage number.
Mixed single-double precision solver

Execution of the sequential TSQR factorization on a flat tree with four submatrices. The gray boxes indicate where local QR factorizations take place. The $Q$ and $R$ factors each have two subscripts: the first is the sequence number for that stage, and the second is the stage number.
Execution of a hybrid parallel / out-of-core TSQR factorization. The matrix has 16 blocks, and four processors can execute local QR factorizations simultaneously. The gray boxes indicate where local QR factorizations take place. We number the blocks of the input matrix $A$ in hexadecimal to save space (which means that the subscript letter $A$ is the number 10₁₀, but the non-subscript letter $A$ is a matrix block). The $Q$ and $R$ factors each have two subscripts: the first is the sequence number for that stage, and the second is the stage number.
$A$ is $m$–by–$n$ with $m = pb$ and $n = qb$.
We are only interested in the first step of the Householder QR factorization.
1 Classic unblocked Householder factorization: DGEQF2

1. DGEQR2: Panel factorization.

\[ 2mb^2 - \frac{2}{3}b^3 = 2(p - \frac{1}{3})b^3 \]

2. DLARF: Apply the V vector one-by-one to the remaining matrix. This makes \( b \) small step

\[ \sum_{\ell=(p-1)b+1}^{pb} 4(q-1)b\ell \]
\[ = 4(q-1)b\sum_{i=1}^{b} (p-1)b+i \]
\[ = 4(q-1)b((p-1)b^2 + \frac{1}{2}b^2) \]
\[ = (4p-2)(q-1)b^3 \]

total \( (4pq - 2p - 2q + \frac{7}{3})b^3 \)
2 Classic block Householder factorization: **DGEQRF**

1. **DGEQR2**: Panel factorization.

\[
2mb^2 - \frac{2}{3}b^3 = 2(p - \frac{1}{3})b^3
\]

2. **DLARFT**: Construction of the T matrix to apply the Householder by block

\[
mb^2 - \frac{1}{3}b^3 = (p - \frac{1}{3})b^3
\]

3. **DLARFB**: Apply the V vectors by block to the remaining matrix.

\[
4bm(n - b) - b^2(n - b) = (4p - 1)(q - 1)b^3 \\
= (4p - 1)(q - 1)b^3 \\
= (2p - 1)(q - 1)b^3 \\
+ (q - 1)b^3 \\
+ (2p - 1)(q - 1)b^3 \\
= (4p - 1)(q - 1)b^3
\]

\[
(4pq - p - q)b^3
\]

So if we want to compare quickly the unblocked Householder code with the block Householder code, there is an overhead of \((p + q)b^3\). There \(pb^3\) overhead due to **DLARFT** and \(qb^3\) due to **DLARFB**.
### 3 SUQRA

#### Step 1.

1.a. First QR factorization.  
\[ \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \quad \begin{array}{c} \begin{array}{c} \text{\color{blue}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \]  
\[ \frac{4}{3} b^3 \]

1.b. DLARFT: Construction of the T matrix to apply the Householder by block  
\[ \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \quad \begin{array}{c} \begin{array}{c} \text{\color{blue}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \]  
\[ \frac{2}{7} b^3 \]

1.c. DLARFB: Apply the V vectors by block to the remaining matrix.  
\[ \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \quad \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \quad \begin{array}{c} \begin{array}{c} \text{\color{blue}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \]  
\[ (I - \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} ) \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \]  
\[ 3(q - 1)b^3 \]

#### Step 2. (repeat this step \( p - 1 \) times.)

2.a. TSQR factorization (TS=triangle-square)  
\[ \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \quad \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \]  
\[ 2b^3 \]

2.b. TS-LARFT: Construction of the T matrix to apply the Householder by block  
\[ \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \quad \begin{array}{c} \begin{array}{c} \text{\color{blue}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \]  
\[ \frac{4}{3} b^3 \]

2.c. TS-LARFB: Apply the V vectors by block to the remaining matrix.  
\[ \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \quad \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \quad \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \]  
\[ (I - \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} ) \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \begin{array}{c} \begin{array}{c} \text{\color{red}↓} \\
\text{\color{blue}↑} \end{array} \end{array} \]  
\[ 5(q - 1)b^3 \]

**Total**  
\[ (5pq - \frac{5}{3}p - 2q + \frac{2}{3})b^3 \]

This algorithm is going to perform 20% more FLOPS. ⇒ Need for inner blocking.
4 SUQRA without blocking

Step 1.

1.a. First QR factorization.

\[ \begin{array}{c}
\text{Apply } \downarrow \text{ to } \\
\end{array} \]

\[ 4^3 b^3 \]

1.c. DLARFB: Apply the V vectors by block to the remaining matrix.

\[ \begin{array}{c}
\text{Apply } \downarrow \text{ to } \\
\end{array} \]

\[ 2(q - 1)b^3 \]

Step 2. (repeat this step \((p - 1)\) times.

2.a. TSQR factorization (TS=triangle-square)

\[ \begin{array}{c}
\text{Apply } \downarrow \text{ to } \\
\end{array} \]

\[ 2b^3 \]

2.c. TS-LARFB: Apply the V vectors by block to the remaining matrix.

\[ \begin{array}{c}
\text{Apply } \downarrow \text{ to } \\
\end{array} \]

\[ 4(q - 1)b^3 \]

**total**

\[ (4pq - 2p - 2q + \frac{4}{3})b^3 \]

Exactly the same number of FLOPs as for the unblocked QR factorization
5 comparison

<table>
<thead>
<tr>
<th></th>
<th>unblocked QR</th>
<th>unblocked SUQRA</th>
<th>blocked QR</th>
<th>blockedSUQRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>panel</td>
<td>$(2p - \frac{4}{3})b^3$</td>
<td>$(2p - \frac{4}{3})b^3$</td>
<td>$(2p - \frac{4}{3})b^3$</td>
<td>$(2p - \frac{4}{3})b^3$</td>
</tr>
<tr>
<td>$T$</td>
<td>$(p - \frac{1}{3})b^3$</td>
<td>$(p - \frac{1}{3})b^3$</td>
<td>$(p - \frac{1}{3})b^3$</td>
<td>$(4p - \frac{4}{3})b^3$</td>
</tr>
<tr>
<td>update</td>
<td>$(4pq - 2q - 4p + 2)b^3$</td>
<td>$(4pq - 2q - 4p + 2)b^3$</td>
<td>$(4pq - q - 4p + 1)b^3$</td>
<td>$(5pq - 2q - 5p + 2)b^3$</td>
</tr>
<tr>
<td>total</td>
<td>$(4pq - 2q - 2p + \frac{2}{3})b^3$</td>
<td>$(4pq - 2q - 2p + \frac{2}{3})b^3$</td>
<td>$(4pq - q - p)b^3$</td>
<td>$(5pq - 2q - \frac{4}{3}p + \frac{5}{3})b^3$</td>
</tr>
</tbody>
</table>

We can integrate those values to know the FLOPS count of the algorithm, we replace $q$ by $i$ when $i$ varies from $q$ to 1 and we replace $p$ with $p - q + i$. We assume that $p > q$, i.e. the matrix is taller than longer.

This gives for the unblocked QR algorithm:

$$\sum_{i=1}^{q} (4(p - q + i)i - 2i - 2(p - q + i) + \frac{4}{3})b^3$$

$$= \sum_{i=1}^{q} (4p - 4q - 4)i + 4i^2 - 2p + 2q + \frac{4}{3}b^3$$

$$= (4p - 4q - 4)\frac{q^2}{2} + 4\frac{q^3}{3} - 2pq + 2q^2 + \frac{4}{3}q)b^3$$

$$= (2pq^2 - 2q^3 - 2q^2 + \frac{4}{3}q^3 - 2pq + 2q^2 + \frac{4}{3}q)b^3$$

$$= (2pq^2 - \frac{2}{3}q^3 - 2pq + \frac{4}{3}q)b^3$$

$$= 2mn^2 - \frac{2}{3}n^3 - 2mb + \frac{4}{3}nb^2$$

The order is OK we find: $2mn^2 - \frac{2}{3}n^3$ as expected.

This gives for the block SUQRA:

$$\sum_{i=1}^{q} (5(p - q + i)i - 2i - \frac{5}{3}(p - q + i) + \frac{2}{3})b^3$$

$$\sim \sum_{i=1}^{q} (5pi - 5qi + 5i^2)b^3$$

$$\sim (\frac{5}{2}pq^2 - \frac{5}{2}q^3 + \frac{5}{3}q^3)b^3$$

$$\sim \frac{5}{2}mn^2 - \frac{5}{6}n^3$$

which means 20% overhead.
## 6 Remark on the panel factorization

<table>
<thead>
<tr>
<th>Classical method</th>
<th>2(2b)b^2 - \frac{2}{3}b^3</th>
<th>\frac{10}{3}b^3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUQRA-step1</td>
<td>\frac{4}{3}b^3 + \frac{4}{3}b^3</td>
<td></td>
</tr>
<tr>
<td>PUQRA-step2</td>
<td>\frac{2}{3}b^3</td>
<td>\frac{10}{3}b^3</td>
</tr>
<tr>
<td>SUQRA-step1</td>
<td>\frac{4}{3}b^3</td>
<td></td>
</tr>
<tr>
<td>SUQRA-step2</td>
<td>2b^3</td>
<td>\frac{10}{3}b^3</td>
</tr>
</tbody>
</table>
Mixed single-double precision solver

k=1

DGEQRT
DTSQRT
DLARFB
DSSRFB

k=2

k=3
Mixed single-double precision solver

An interesting middleware: SMPSs

```c
#pragma css task
inout(RV1[NB][NB]) output(T[NB][NB])
void dgeqrt(double *RV1, double *T);

#pragma css task
inout(R[NB][NB], V2[NB][NB]) output(T[NB][NB])
void dtsqrt(double *R, double *V2, double *T);

#pragma css task
input(V1[NB][NB], T[NB][NB]) inout(C1[NB][NB])
void dlarfb(double *V1, double *T, double *C1);

#pragma css task
input(V2[NB][NB], T[NB][NB]) inout(C1[NB][NB], C2[NB][NB])
void dssrfb(double *V2, double *T, double *C1, double *C2);
```

```latex
#pragma css start
for (k = 0; k < TILES; k++) {
    dgeqrt(A[k][k], T[k][k]);
    for (m = k+1; m < TILES; m++)
        dtsqrt(A[k][k], A[m][k], T[m][k]);
    for (n = k+1; n < TILES; n++)
        dlarfb(A[k][k], T[k][k], A[k][n]);
    for (m = k+1; m < TILES; m++)
        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
} #pragma css finish
```

From:


See also:


Performance of the tile QR factorization in single precision on a 3.2 GHz CELL processor with eight SPEs. Square matrices were used. Solid horizontal line marks performance of the SSSRFB kernel times the number of SPEs ($22.16 \times 8 = 177$ [Gflop/s]).

“The presented implementation of tile QR factorization on the CELL processor allows for factorization of a 4000–by–4000 dense matrix in single precision in exactly half of a second. To the author’s knowledge, at present, it is the fastest reported time of solving such problem by any semiconductor device implemented on a single semiconductor die.”

Jakub Kurzak and Jack Dongarra, LAWN 201 – QR Factorization for the CELL Processor, May 2008.
Q and R: Strong scalability

In this experiment, we fix the problem: $m=1,000,000$ and $n=50$.
Then we increase the number of processors.

Blue Gene L
frost.ncar.edu
Mixed single-double precision solver
Mixed single-double precision solver

Speedup of CAQR over ScalAPACK

- Speedup on the y-axis
- Log10(N) on the x-axis
- Log2(P) on the z-axis

The graph shows the speedup of CAQR over ScalAPACK for different values of log10(N) and log2(P).
Strategy:

1. obtain some lower bounds for the cost (latency, bandwidth, # of operations) of LU, QR and Cholesky in sequential and parallel distributed

2. compute the costs of our algorithms and compare with the lower bound.

Lower bounds:

1. For LU, observe that:

\[
\begin{pmatrix}
I & 0 & -B \\
A & I & 0 \\
0 & 0 & I
\end{pmatrix} = \begin{pmatrix}
I & A \\
0 & 0 \\
0 & 0
\end{pmatrix} \begin{pmatrix}
I & 0 & -B \\
A & I & A \cdot B \\
0 & 0 & I
\end{pmatrix}
\]

therefore lower bound for matrix-matrix multiply (latency, bandwidth and operations) also holds for LU.

2. For Cholesky, observe that:

\[
\begin{pmatrix}
I & A^T & -B \\
A & I + AA^T & 0 \\
-B^T & 0 & D
\end{pmatrix} = \begin{pmatrix}
I & A \\
0 & 0 \\
0 & 0
\end{pmatrix} \begin{pmatrix}
I & A^T & -B \\
A & I & A \cdot B \\
-B^T & (A \cdot B)^T & X
\end{pmatrix}
\]


Performance models of parallel CAQR and ScaLAPACK’s parallel QR factorization PDGEQRF on a square $n \times n$ matrix with $P$ processors, along with lower bounds on the number of flops, words, and messages. The matrix is stored in a 2-D $P_r \times P_c$ block cyclic layout with square $b \times b$ blocks. We choose $b$, $P_r$, and $P_c$ optimally and independently for each algorithm. Everything (messages, words, and flops) is counted along the critical path.
## Performance models of sequential CAQR and blocked sequential Householder QR on a square $n \times n$ matrix with fast memory size $W$, along with lower bounds on the number of flops, words, and messages.

<table>
<thead>
<tr>
<th></th>
<th>Seq. CAQR</th>
<th>Householder QR</th>
<th>Lower bound</th>
</tr>
</thead>
<tbody>
<tr>
<td># flops</td>
<td>$\frac{4}{3} n^3$</td>
<td>$\frac{4}{3} n^3$</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td># words</td>
<td>$3 \frac{n^3}{\sqrt{W}}$</td>
<td>$\frac{1}{3} n^4$</td>
<td>$O(\frac{n^3}{\sqrt{W}})$</td>
</tr>
<tr>
<td># messages</td>
<td>$12 \frac{n^3}{W^{3/2}}$</td>
<td>$\frac{1}{2} n^3$</td>
<td>$O(\frac{n^3}{W^{3/2}})$</td>
</tr>
</tbody>
</table>
Opportunities for further research:

how to tile the matrix? square blocks, nonsquare blocks?

**kernel tuning:** introduction of a lots of new kernels (e.g. QR fact. of a triangle on top of a square). For each kernel:

1. how to optimize the blocking parameter \((nb)\)?
2. which algorithmic variants to choose (left looking, recursive, ...) ?
3. the inner blocking parameter \((ib)\).

Question 2 and 3 are standard autotuning problems. Choosing \(ib\) and the algorithmic vairant is done in term of \(nb\). Question 1 is more subtle. Choosing \(nb\) is done at the matrix level \((n)\) since it influences the granularity of the algorithm.

**reduction algorithm:** Which reduction tree to use? Binary tree? Flat tree? Hybrid tree? Each of these choices represent an algorithm change. No framework to accomodate this yet.

**scheduling:** How to schedule all these tasks?

1. static scheduling or dynamic scheduling?
2. and in parallel distributed ...