Fast computation of QR factorization and eigenvalue decomposition via one-sided plane rotations

Ivan Slapničar

University of Split, Croatia

Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture

Joint work with Krešimir Veselić[a], Fernuniversität Hagen, and

Zlatko Drmač, University of Zagreb

[a]I. Slapničar and K. Veselić acknowledge the grant from the Croatian Science Foundation

Motivation

Drmač and Veselić (2006, see LAWN #169, 170) derived an SVD routine which is:

- as fast or faster than the QR method from (D,S)GESVD and
- highly accurate.

Key ingredients of the algorithm are:

- QR factorization with pivoting,
- QR factorization,
- one-sided Jacobi method with tiling-based pivoting.

Tiling

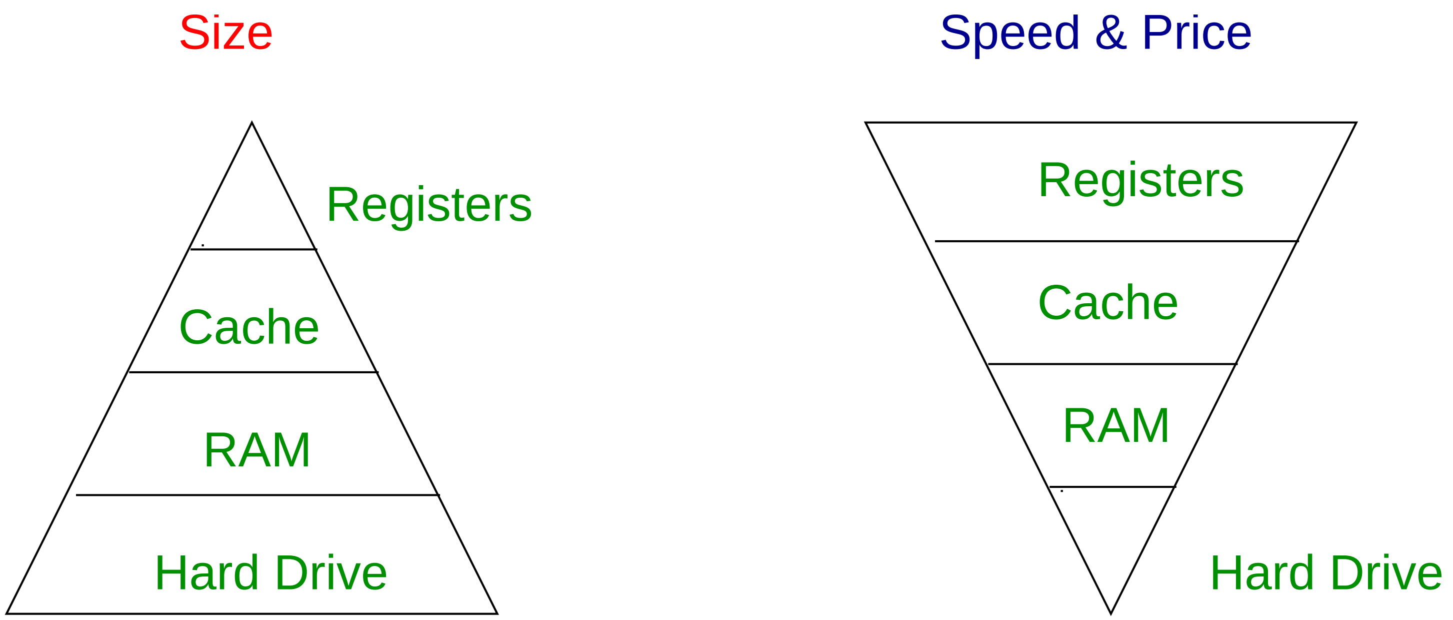
Example: choice of pivoting positions for $n = 8$ and block-size $nb = 3$:

$$\begin{bmatrix} \bullet & 1 & 2 & 4 & 5 & 6 & 13 & 14 \\ & \bullet & 3 & 7 & 8 & 9 & 15 & 16 \\ & & \bullet & 10 & 11 & 12 & 17 & 18 \\ & & & \bullet & 19 & 20 & 22 & 23 \\ & & & & \bullet & 21 & 24 & 25 \\ & & & & & \bullet & 26 & 27 \\ & & & & & & \bullet & 28 \\ & & & & & & & \bullet \end{bmatrix}$$

Ideas

1. Compare Givens QR factorisation with tiling and the standard BLAS 3 Householder implementation,
2. Improve the Demmel-Veselić implementation of the highly accurate algorithm for positive definite eigenvalue problem (make it faster) – fast Cholesky with pivoting + one sided Jacobi with tiling.

Memory hierarchy



Data traffic between RAM and Cache in done by moving consecutive blocks of memory (pages).

Conclusion: use data in cache as much as posible

BLAS

Basic Linear Algebra Subroutines

level	operands	example	data	flop
BLAS 1	vector, vector	**ddot**, **daxpy**	$O(n)$	$O(n)$
BLAS 2	matrix, vector	$\alpha Ax + \beta y$	$O(n^2)$	$O(n^2)$
BLAS 3	matrix, matrix	$\alpha AB + \beta C$	$O(n^2)$	$O(n^3)$

ddot: $d = x^T y = \sum_i x_i y_i$
daxpy: $y \leftarrow \alpha\, x + y \quad (y_i \leftarrow \alpha\, x_i + y_i)$

Conclusion: use matrix operations as much as possible
(or achieve similar effect with tiling)

It matters

Intel Xeon (em64t) has ~5,000 Mflops peak with Intel Math Kernel Library (mkl). For `ddot` and `daxpy` we obtain

	$a(:,i) \cdot a(:,i+1)$	$a(:,i) \cdot a(i,:)$	$a(i,:) \cdot a(i+1,:)$
`-O4`	502	166	173
`mkl`	573	165	173

	`daxpy_1`	`daxpy_1n`
`-O4`	312	136
`mkl`	312	135

Conclusion: approach data column-wise

It matters a lot

m	n	Mflops (`-O4`)	Mflops (`mkl`)
4	4	71	125
32	16	636	1612
32	32	540	2856
64	32	781	3571
64	64	729	4347
128	4	442	1190
128	64	854	**4542**
128	128	818	4340

Matrix multiplication $A_{mn} \cdot B_{nn}$ with `DGEMM`

QR factorization

$$A = QR = \begin{bmatrix} R_0 \\ 0 \end{bmatrix}, \qquad Q \text{ orthogonal}, \quad R \text{ upper triangular}$$

Example for $m = 5$ and $n = 3$:

$$\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Implementation with Householder reflectors

$$Hx = \left(I - 2\frac{vv^T}{v^T v} \right) x = x - v\frac{2(v^T x)}{v^T v}.$$

This requires $O(6n)$ flop. Similarly,

$$\beta = -\frac{2}{v^T v}, \quad w = \beta A^T v \quad HA = A + vw^T,$$

which requires $O(n^2)$ flop. Operation count for R is

$$\sum_{i=1}^{n} 4\, i^2 \approx \frac{4}{3} n^3.$$

The same holds for Q if we compute (otherwise it is $O(2n^3)$)

$$Q_n, \quad Q_{n-1} \cdot Q_n, \quad Q_{n-2} \cdot (Q_{n-1} \cdot Q_n), \cdots$$

Block algorithm

Good: we are accessing data column-wise
Bad: we are not using BLAS 3.

Solution: use block transformations:

- Dietrich (1976): $H_k = I - 2\,V_k(V_k^T V_k)^{-1} V_k^T$.
- Bischof and Van Loan (1986): WY representation:

$$H_k = I + W_k Y_k^T, \qquad A \;\leftarrow\; Q_k^T A = A + Y_k(W_k^T A)$$

The operation count increases by factor $(1 + k/n)$.

DGEQRF takes 0.4 seconds $\longrightarrow$

$$((4/3) \cdot 1000^3)/0.4 = 3,333 \text{ Mflops}$$

Givens rotation

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

$$r = \mathrm{sign}(y)\sqrt{x^2 + y^2}, \quad c = \frac{x}{r}, \quad s = \frac{y}{r}$$

Computation of c, s and r is implemented in `srotg` and `drotg`.

Rotation is implemented in `srot` and `drot`.

QR with Givens rotations

$$\begin{array}{c}\rightarrow\\\rightarrow\\ \\ \\ \\ \end{array}\begin{bmatrix}\bullet & \bullet & \bullet\\ 0 & \bullet & \bullet\\ \times & \times & \times\\ \times & \times & \times\\ \times & \times & \times\end{bmatrix},\quad \begin{array}{c}\rightarrow\\ \\ \rightarrow\\ \\ \\ \end{array}\begin{bmatrix}\oplus & \oplus & \oplus\\ 0 & \times & \times\\ 0 & \oplus & \oplus\\ \times & \times & \times\\ \times & \times & \times\end{bmatrix},\quad \begin{array}{c}\rightarrow\\ \\ \\ \rightarrow\\ \\ \end{array}\begin{bmatrix}\odot & \odot & \odot\\ 0 & \times & \times\\ 0 & \times & \times\\ 0 & \odot & \odot\\ \times & \times & \times\end{bmatrix},\cdots$$

Operation count for R is

$$\sum_{i=1}^{n} 6\,i\,(i-1) \approx 2\,n^3 \text{ flop}$$

Implementation

Bad: we are accessing data row-wise
Bad: we are not using BLAS 3
Bad: operation count is too large ($2\,n^3$ v.s. $4\,n^3/3$)

Implementation

Bad: we are accessing data row-wise
Bad: we are not using BLAS 3
Bad: operation count is too large ($2\,n^3$ v.s. $4\,n^3/3$)

Solution: work on the transposed matrix – compute

$$A^T = R^T Q^T$$

Implementation

Bad: we are accessing data row-wise
Bad: we are not using BLAS 3
Bad: operation count is too large ($2\,n^3$ v.s. $4\,n^3/3$)

Solution: work on the transposed matrix – compute

$$A^T = R^T Q^T$$

Solution: use tiling - REUSE DATA IN CACHE

Implementation

Bad: we are accessing data row-wise
Bad: we are not using BLAS 3
Bad: operation count is too large ($2\,n^3$ v.s. $4\,n^3/3$)

Solution: work on the transposed matrix – compute

$$A^T = R^T Q^T$$

Solution: use tiling - REUSE DATA IN CACHE

Solution: use fast self-scaling rotations (Anda and Park) - BUT NOT ON QUAD CORE

Fast rotations

Standard:

$$\begin{bmatrix} 1 & \beta \\ -\alpha & 1 \end{bmatrix} \begin{bmatrix} \delta & \\ & \delta \end{bmatrix}, \qquad \begin{bmatrix} \beta & 1 \\ -1 & \alpha \end{bmatrix} \begin{bmatrix} \delta & \\ & \delta \end{bmatrix},$$

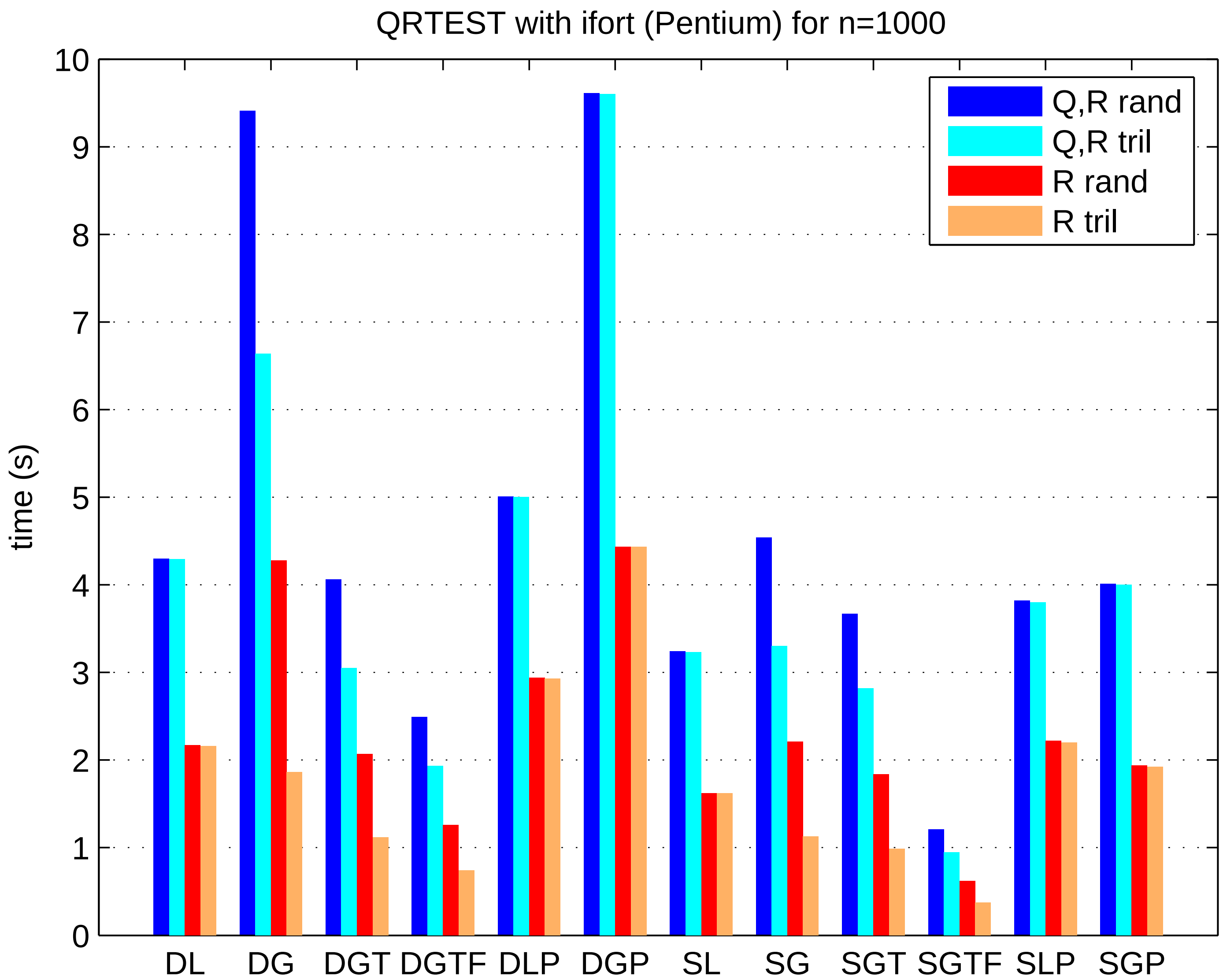
δs are accumulated in the vector $\mathbf{d}$.

Self-scaling: for example, for $\theta \leq \pi/4$ and $d_i \geq d_j$

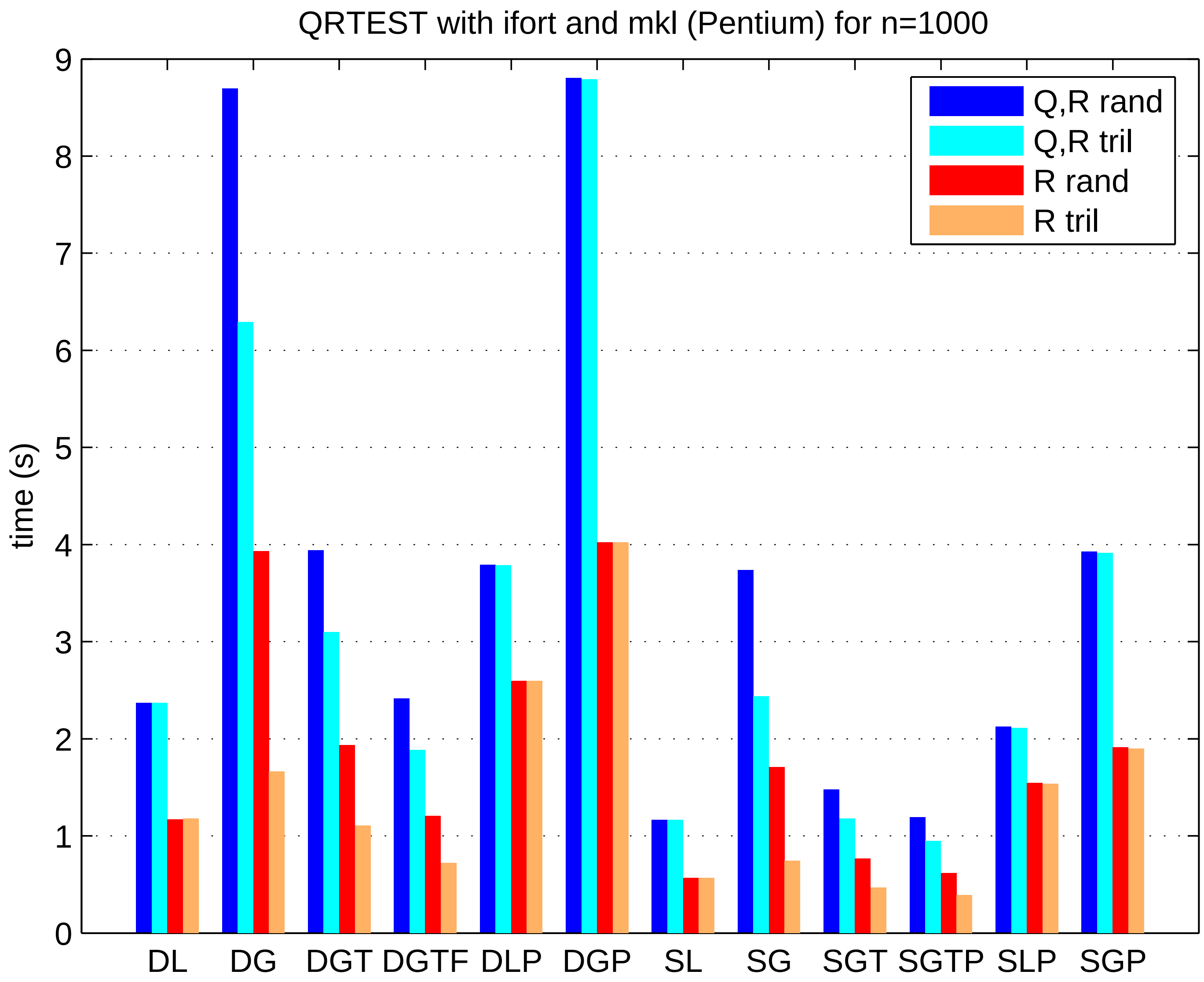
$$\begin{bmatrix} 1 & 0 \\ -\alpha & 1 \end{bmatrix} \begin{bmatrix} 1 & \beta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1/\delta & \\ & \delta \end{bmatrix}$$

There exist three more variants. Operation count is now $4n^3/3$ flop.

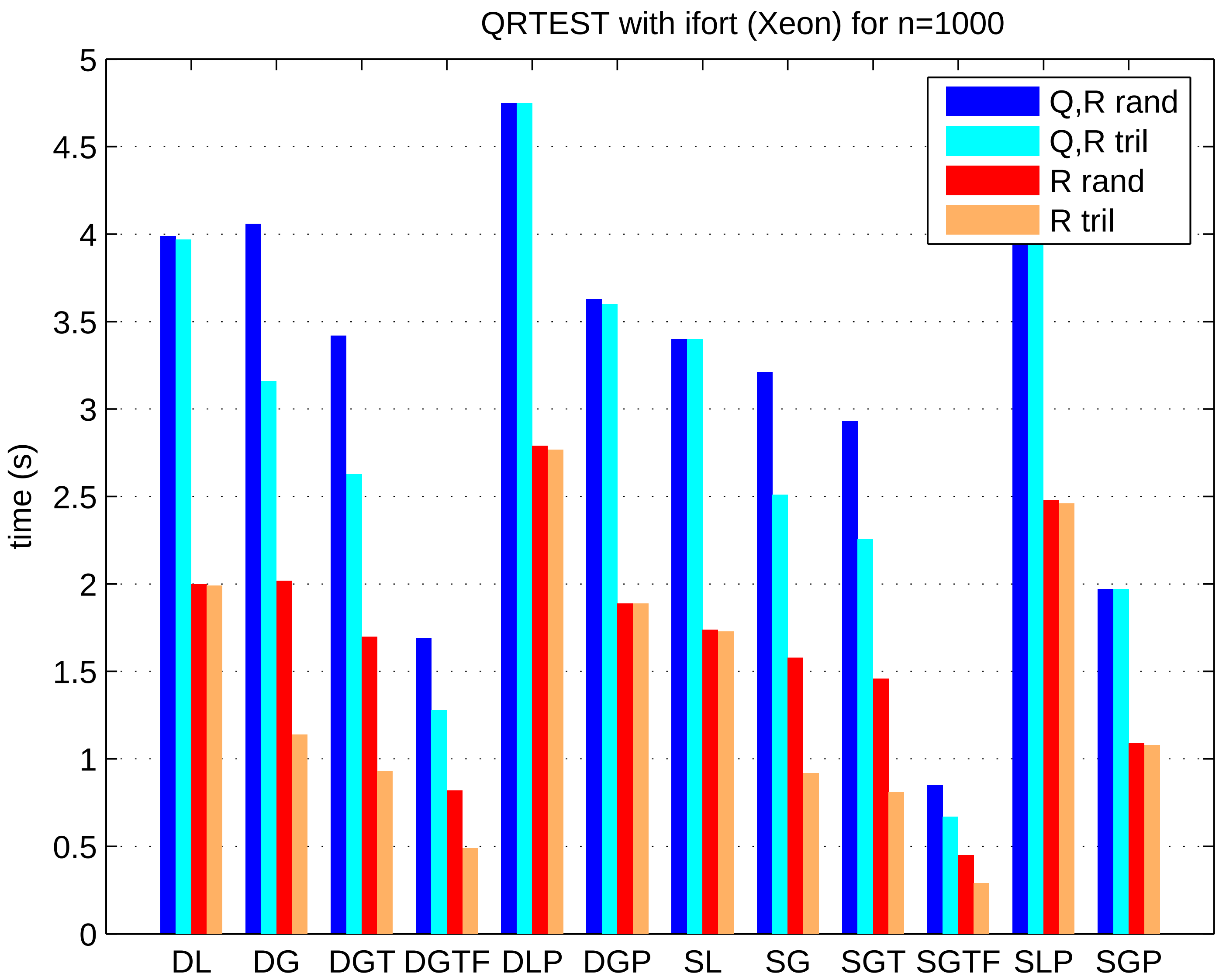
Experiments Pentium



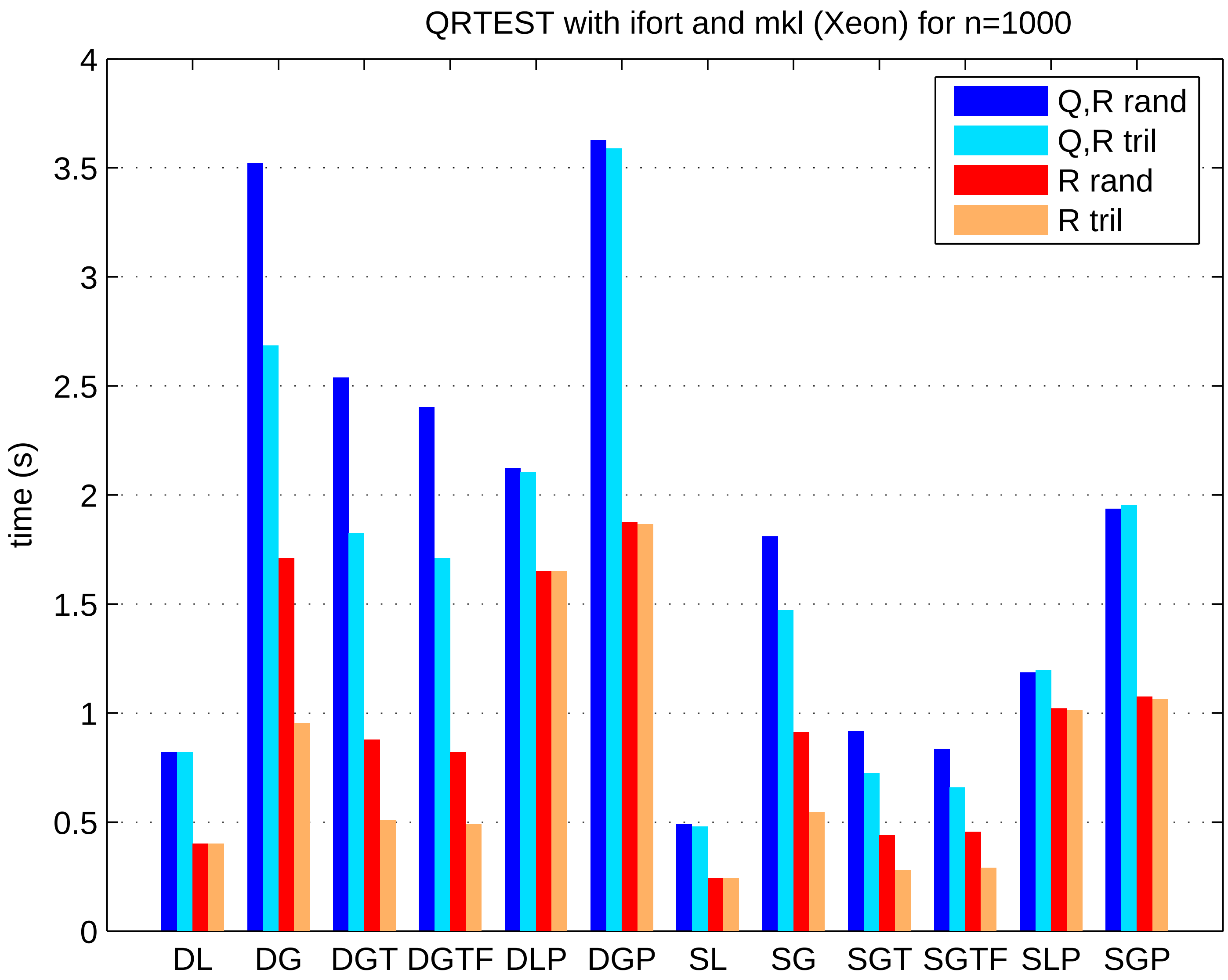
Experiments Pentium (mkl)



Experiments Xeon

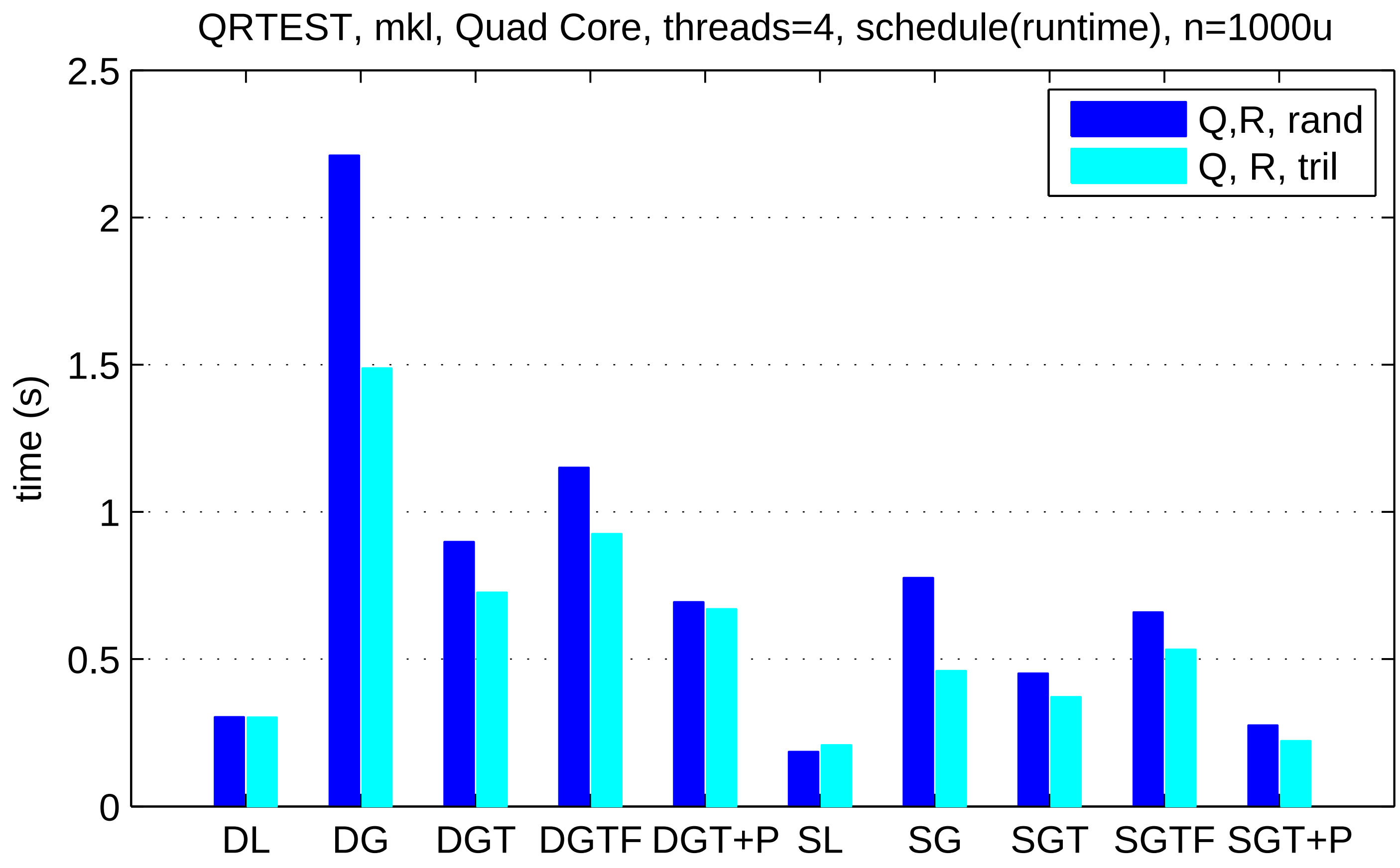


Experiments Xeon (mkl)



Experiments Xeon Quad Core (mkl)

Parallelism is needed!



Parallelism

$$\begin{bmatrix} \times & & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & & \times & 0 & \times & 0 & \times & 0 & \times & 0 \\ \times & & \times & 0 & 0 & 0 & \times & 0 & 0 & 0 \\ \times & & \times & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \times & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Should be ideal, but it is not - there is not enough control of memory access in OpenMP implementation.

CONCLUSION

Results depend on architecture and compiler.

Quad Core processors are new issue. Nvidia - unknown!

Givens rotations are:

- comparable in speed with the Householder reflectors,
- simpler to implement.

Plane rotations with tilings and parallel strategy should be condsidered for other problems.

Eigenvalue computations

$$A\,\mathbf{x} = \lambda\,\mathbf{x}, \quad A = A^T \;\rightarrow\; Q^T A Q = \Lambda,\; Q^T Q = I$$

QR method:

- tridiagonalization with Householder reflectors
- iterate { $T = QR$ (factorize), $T = RQ$ (multiply) }

Jacobi method (1845.): iterate

$$\begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & \times & \times \\ a_{12} & a_{22} & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \begin{bmatrix} c & s & 0 & 0 \\ -s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \bar{a}_{11} & 0 & \bullet & \bullet \\ 0 & \bar{a}_{22} & \bullet & \bullet \\ \bullet & \bullet & \times & \times \\ \bullet & \bullet & \times & \times \end{bmatrix}$$

High relative accuracy

QR computes:

$$|\delta\lambda| \leq \varepsilon|\lambda|\kappa(A).$$

For A positive definite, Jacobi computes:

$$|\delta\lambda| \leq \varepsilon\,\lambda\,\kappa(A_S).$$

($\kappa(A) = \|A\|\,\|A^{-1}\|$, $A_S = DAD,\;\; D = \mathrm{diag}(A)^{-1/2}$.)

Bad: Jacobi is several times slower than QR.

Solution: two-step algorithm (Demmel & Veselić, 1989):

- Cholesky factorization $A = LL^T$
- one-sided Jacobi on L

One-sided Jacobi

Diagonalize $L^T L$ by applying only transformations on L,

$$L_{k+1} = L_k U_k.$$

Here

- c and s are computed from the 2×2 submatrix of $(LU_k)^T (LU_k)$ (1 scalar product).
- L_k converges to a matrix with orthogonal columns.
- Let $U = \prod U_k$. Then $U^T L^T U L = \Lambda$.
- Let $Q = LU\Lambda^{-1/2}$. Then $Q^T AQ = \Lambda$.

One-sided Jacobi

Diagonalize $L^T L$ by applying only transformations on L,

$$L_{k+1} = L_k U_k.$$

Here

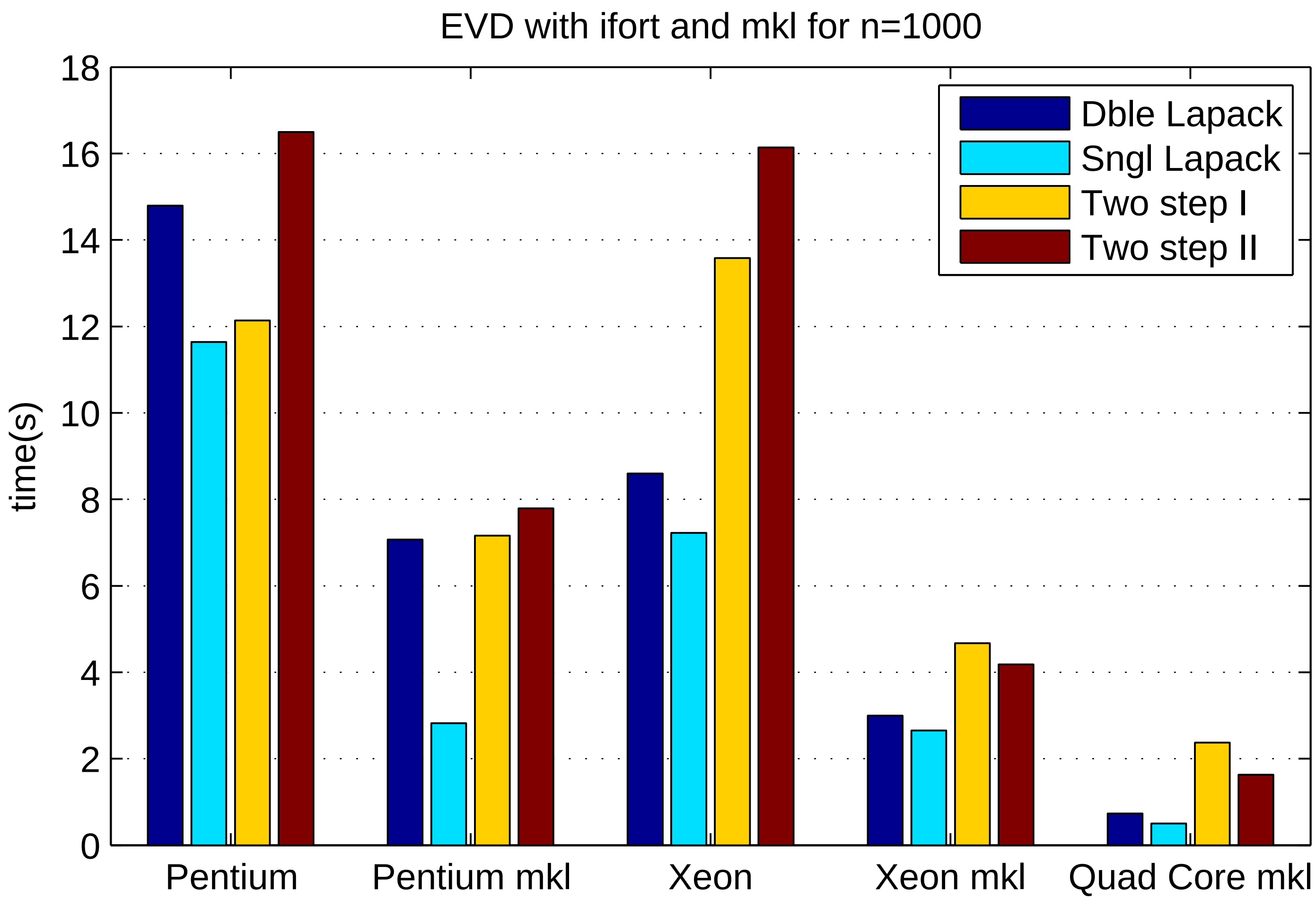
- c and s are computed from the 2×2 submatrix of $(LU_k)^T(LU_k)$ (1 scalar product).
- L_k converges to a matrix with orthogonal columns.
- Let $U = \prod U_k$. Then $U^T L^T U L = \Lambda$.
- Let $Q = LU\Lambda^{-1/2}$. Then $Q^T A Q = \Lambda$.

Bad: two-step algorithm is still slower than QR.

Implementation

- Use Cholesky with pivoting - this has a diagonalizing effect and makes Jacobi part faster (Demmel & Veselić).
  We use block & pivoting version by Lucas (2004) - very fast!
- One-sided Jacobi accesses data column-wise.
  We add tiling (Drmač & Veselić) and fast rotations.
- c and s are computed in double precision - this helps speed and accuracy.

Experiments



Two-step II: 1-4 threads – 2.00s, 1.84s, 1.73s, 1.62s, respectively – there is place for improvement!