PERFORMANCE AND ACCURACY OF LAPACK'S SYMMETRIC TRIDIAGONAL EIGENSOLVERS

JAMES W. DEMMEL† OSNI A. MARQUES‡ BERESFORD N. PARLETT† AND CHRISTOF VÖMEL‡

Abstract. We compare four algorithms from the latest LAPACK 3.1 release for computing eigenpairs of a symmetric tridiagonal matrix. These include QR iteration, bisection and inverse iteration (BI), the Divide-and-Conquer method (DC), and the method of Multiple Relatively Robust Representations (MR).

Our evaluation considers speed and accuracy when computing all eigenpairs, and additionally subset computations. Using a variety of carefully selected test problems, our study includes a variety of today's computer architectures.

Our conclusions can be summarized as follows. (1) DC and MR are generally much faster than QR and BI on large matrices. (2) MR almost always does the fewest floating point operations, but at a lower MFlop rate than all the other algorithms. (3) The exact performance of MR and DC strongly depends on the matrix at hand. (4) DC and QR are the most accurate algorithms with observed accuracy $O(\sqrt{n}\varepsilon)$. The accuracy of BI and MR is generally $O(n\varepsilon)$. (5) MR is preferable to BI for subset computations.

Key words. LAPACK, symmetric eigenvalue problem, inverse iteration, Divide & Conquer, QR algorithm, MRRR algorithm, accuracy, performance, benchmark.

AMS subject classifications. 15A18, 15A23.

1. Introduction.

One goal of the latest 3.1 release [25] of LAPACK [1] is to produce the fastest possible symmetric eigensolvers subject to the constraint of delivering small residuals and orthogonal eigenvectors.

For an input matrix A that may be dense or banded, one standard approach is the conversion to tridiagonal form T, then the eigenvalues and eigenvectors of T are found, and last the eigenvectors of T transformed to eigenvectors of A.

Depending on the situation, all the eigenpairs or just some of them may be desired. LAPACK, for some algorithms, allows selection by eigenvalue indices ('find $\lambda_i, \lambda_{i+1}, ...\lambda_j$, where $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ are all the eigenvalues in increasing order', and their eigenvectors) or by an interval ('find all the eigenvalues in $[a, b]$ and their eigenvectors').

This paper analyzes the performance and accuracy of four algorithms:

1. QR iteration, in LAPACK's driver STEV (QR for short),
2. Bisection and Inverse Iteration, in STEVX (BI for short),
3. Divide and Conquer, in STEVD (DC for short),
4. Multiple Relatively Robust Representations, in STEVR (MR for short)

Section 2 gives a brief description of these algorithms with references.

For a representative picture of each algorithm's capacities, we developed an extensive set of test matrices [7], broken into two classes: (1) 'practical matrices' based on reducing matrices from a variety of practical applications to tridiagonal form, and generating some other tridiagonals with similar spectra, and (2) synthetic 'testing matrices' chosen to have extreme distributions of eigenvalues or other properties designed to exercise one or more of the algorithms, see Section 3.1 for details. The timing and

†Mathematics Department and Computer Science Division, University of California, Berkeley, CA 94720, USA. {demmel@cs,parlett@math}.berkeley.edu

‡Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA. {oamarques,cvoemel}@lbl.gov

accuracy tests were performed on a large set of current computing platforms which are described in Section 3.2.

Ideally one of these algorithms would be the best in all circumstances. In reality, the performance of an algorithm may depend on the matrix, the platform, and possibly underlying libraries like BLAS, so we do need to judge very carefully. To study and illuminate some aspects of the astonishing variability in behavior is the goal of this paper.

Section 4 presents performance results when computing all eigenpairs. Its first part, Section 4.1, consists of an overall summary of performance on practical matrices across all investigated architectures. DC and MR are usually much faster than QR and BI for these matrices. Section 4.2 looks at one architecture in detail and shows that MR almost always does the fewest floating point operations, but at a lower MFlop rate than all the other algorithms. For certain matrix classes for which there is a great deal of deflation, DC becomes much faster. Section 4.3 further illustrates the dependence of algorithm performance on certain matrix characteristics by closely studying the behavior on selected synthetic test problems.

The performance of subset computations is analyzed in Section 5. Only BI and MR allow the computation of subsets at reduced cost. We show that MR beats BI on average and identify matrices with subsets where one algorithm wins over the other.

Section 6 shows that QR and DC are the most accurate algorithms, measured both in terms of producing pairwise orthogonal eigenvectors and small residuals norms. MR is less accurate but still achieves errors of size $O(n\varepsilon)$, where n is the dimension and ε is machine epsilon. Depending on the matrix and platform, it is known that BI may completely fail to guarantee orthogonal eigenvectors [9], though this is rare and did only occur in a few subset calculations with our test matrices.

Summary and conclusions are given in Section 7.

2. Description of Algorithms. Table 2.1 gives an overview of LAPACK's symmetric eigensolvers.

Algorithm	Driver	Subsets	Workspace	References
QR algorithm (QR)	STEV	N	real: $2n-2$	[6, 21, 28]
Divide & Conquer (DC)	STEVD	N	real: $1+4n+n^2$, int.: $3+5n$	[3, 4, 22]
Bisection/Inverse Iter. (BI)	STEVX	Y	real: $8n$, int.: $5n$	[9, 24]
MRRR algorithm (MR)	STEVR	Y	real: $18n$, int.: $10n$	[8, 29, 30, 11, 12, 14]

TABLE 2.1

LAPACK codes for computing the eigenpairs of a symmetric matrix of dimension n. See also [1, 2]. The driver column lists the LAPACK driver name. The subset column indicates whether the algorithm can compute subsets at reduced cost. With respect to memory, we note that QR uses the least, then MR and BI, and DC uses the most. Note that the workspace that is reported for DC corresponds to the case COMPZ = 'I'. The workspace that is reported for BI is for SYEVX, the driver that combines STEBZ and STEIN.

2.1. QR Iteration (QR). QR applies a sequence of similarity transforms to the tridiagonal T until its off-diagonal elements become negligible and the diagonal elements have converged to the eigenvalues of T. It consists of a bulge-chasing procedure that implicitly includes shifts and only uses plane rotations while preserving the tridiagonal form [28]. The plane rotations are accumulated to find the eigenvectors of T. The overall complexity of the method is $3bn^3 + \mathcal{O}(n^2)$, where b denotes the average number of bulge chases per eigenvalue, see [6]. No part of the bulge-chasing procedure currently makes use of higher level BLAS.

2.2. Bisection and Inverse Iteration (BI). Bisection based on Sturm sequences requires $\mathcal{O}(nk)$ operations to compute k eigenvalues of T. If the distance between the eigenvalues is large enough (relative to $\|T\|$), then computing the corresponding eigenvector by inverse iteration also is a $\mathcal{O}(nk)$ process. If however the eigenvalues are not well separated, Gram-Schmidt orthogonalization is employed to try to achieve numerically orthogonal eigenvectors. In this case the complexity of the algorithm increases to $\mathcal{O}(nk^2)$. In the worst case where almost all eigenvalues of T are 'clustered', the complexity can increase to $\mathcal{O}(n^3)$. Furthermore, from the accuracy point of view this procedure is not guaranteed to be reliable, see [6, 9]. Neither bisection nor inverse iteration make use of higher level BLAS.

2.3. Divide and Conquer (DC). The Divide & Conquer method can be described in terms of a binary tree where each node corresponds to a submatrix and its eigenpairs, obtained through recursively dividing the matrix in halves, see the exposition in [6].

The tree is processed bottom up, starting with submatrices of size 25 or smaller.[1] DC uses QR to solve the small eigenproblems and then computes the eigenpairs of a parent using the already computed eigenpairs of the children.

A parent's eigenvalues can be computed as solutions of a secular equation. The eigenvector computation consists of two steps. The first one is a relatively inexpensive scaling step. The second one, which is most of the work, multiplies the eigenvectors of the current matrix by the eigenvector matrix accumulated so far. This step uses the BLAS3 routine DGEMM. In the worst case, DC is an $\mathcal{O}(n^3)$ algorithm. On practical matrices studied in Table 4.1 of Section 4.1, the effective exponent is less than three.

The complexity of the eigenvector computation can sometimes be reduced substantially by a a process called *deflation*. If a submatrix eigenvalue nearly equals another, or certain entries in the submatrix eigenvector are small enough, the matrix column can be excluded from the BLAS 3 operation. In Table 4.3 from Section 4.3, we will see that for some matrices, deflation may occur for most eigenpairs, substantially accelerating the computation.

2.4. Multiple Relatively Robust Representations (MR). MR is a sophisticated variant of inverse iteration that avoids Gram-Schmidt orthogonalization and thus becomes an $\mathcal{O}(n^2)$ algorithm. The algorithm can be described in terms of a (generally irregular) *representation tree*. The root node describes the entire spectrum of T and the children define gradually refined eigenvalue approximations. See the references in Table 2.1 for the details. The overall complexity of the algorithm depends on the clustering of the eigenvalues. If some eigenvalues of T agree to d digits on average, then the algorithm has to do work proportional to dn^2. The algorithm uses a random perturbation to ensure with high probability that eigenvalues cannot be too strongly clustered, see [13] for details. MR cannot make use of higher level BLAS.

3. The Testing Environment.

3.1. Description of Test Matrices. In this section, we give a brief overview of the collection of test matrices used in this study. A more detailed description of the testing infrastructure is given in [7]. We focus on two types of matrices.

The first class of tridiagonals stems from important applications and thus are relevant to a group of users. For the smaller matrices, the tridiagonal form of the

[1] In a future release, the current fixed threshold 25 will be tunable for each platform to get the highest performance.

sparse matrices was obtained with LAPACK's tridiagonal reduction routine `sytrd`. For the larger matrices we generated tridiagonals by means of a simple Lanczos algorithm without reorthogonalization which, in finite precision, tends to produce copies of eigenvalues as clusters.

- Matrices obtained from G. Fann using the NWChem computational chemistry package [20, 19]: these matrices have clustered eigenvalues that require a large number of reorthogonalizations in BI. This motivated the development of the MR which can cope well with this type of matrix, see [8, 10].
- Examples from sparse matrix collections, including matrices from the BCSSTRUC1 set in [16, 17, 18] and matrices from the Alemdar, NASA, and Cannizzo sets in [5]. These matrices, coming from a variety of applications including power system networks, shallow wave equations, and finite-element problems, were chosen for their spectrum which typically consists of a part with eigenvalues varying almost 'continuously' and another one with several isolated large clusters of eigenvalues of varying tightness.

The second class of matrices are synthetic 'test matrices' that exhibit the strengths, weaknesses, or idiosyncrasies of a particular algorithm. This class includes distributions that are already used in LAPACK's tester, synthetic distributions, and matrices that are used in [8]. Furthermore, it includes Wilkinson matrices [31, 23] and glued Wilkinson matrices (see for example [13]). A detailed description of these matrices is given later, in Table 4.3 of Section 4.3.

3.2. Description of Test Platforms. In order to reliably quantify accuracy and performance and also to detect architecture-specific issues, we perform tests on a large number of today's computer systems including a variety of superscalar ones (Power 3 & 5, Xeon, Opteron), an EPIC (Itanium 2), and a vector computer (X1). Table 3.1 summarizes the architectures, compilers, and timers used for our experiments.

Architecture	Symbol	(MHz)	OS	Compiler	Timer	BLAS
Power 3	SP3	375	AIX	IBM xlf90 -O3	PAPI	ESSL
Power 5	SP5	1900	AIX	IBM xlf90 -O3	PAPI	ESSL
Sun UltraSparc 2i	SUN	650	Solaris	SUN f90 forte 7.0 -O4	CPU_TIME	SUNPERF
MIPS R12000	SGI	600	IRIX	MIPS pro 7.3.1.3m -O2	ETIME	SCS
Itanium 2	ITN2	1400	Linux	Intel ifort 9.0 -O2	ETIME	MKL
Pentium 4 Xeon	P4	4000	Linux	Intel ifort 9.0 -O3	ETIME	MKL
Cray X1	X1	800	UNICOS/mp	Cray ftn 5.4.0.4 -O2	CPU_TIME	LIBSCI
Opteron	OPT	2200	Linux	Pathscale pathf90 2.1 -O3	CPU_TIME	ACML

TABLE 3.1
Platforms, timers, and BLAS libraries used for testing.

3.3. Important metrics. For a tridiagonal matrix T and computed eigenvectors $Z = [z_1\ z_2 \cdots z_m]$ and corresponding eigenvalues $\Lambda = (\lambda_1 \cdots \lambda_m)$, $m \leq n$, we compute both the loss of orthogonality

$$O(Z) = \max_{i \neq j} \frac{|z_i^T z_j|}{n\varepsilon}, \tag{3.1}$$

and the largest residual norm

$$R(\Lambda, Z) = \max_i \frac{||Tz_i - \lambda_i z_i||}{||T||n\varepsilon}. \tag{3.2}$$

The factor $n\varepsilon$ in the denominators of (3.1) and (3.2) is used for normalization. For an algorithm to be satisfactorily accurate, both of these metrics should be bounded by a 'modest constant' for all matrices.

Because of its prominent role in DC, we also measure the amount of deflation encountered during the algorithm. We define the fraction of deflated eigenvalues fr_{defl} as the total number of deflations over the total number of submatrix eigenvalues, over all submatrices encountered while running DC. If fr_{defl} is close to 1, nearly all eigenvalues and eigenvectors were computed with little work, and if fr_{delf} is close to 0, then the maximum amount of floating point work was done, mostly in calls to DGEMM.

4. Performance when computing all eigenpairs.

4.1. Comparison on practical matrices across architectures. Table 4.1 summarizes the full spectrum comparison over all architectures. DC and MR are usually much faster than QR and BI for large practical matrices (dimension 500 or larger). The median ratio runtime(QR)/runtime(MR) varies from 1.8 to 69 across platforms (10 to 69 omitting the Cray X1), and is as large as 710. The median ratio runtime(BI)/runtime(MR) varies from 2.0 to 7.1 across platforms, and is as large as 310. MR is faster than DC on large practical matrices (i.e. the median value of the ratio runtime(DC) over runtime(MR) exceeds one) on 5 of our 8 platforms, and is slower than DC on 3 platforms. MR ranges from 12x faster than DC to 40x slower on these matrices (12x faster to 17x slower omitting the Cray X1).

Performance Summary for Large ($n \geq 500$) Practical Matrices													
Mach	Slope of time trends				Minimum, median, maximum for time ratios								
	QR	BI	DC	MR	QR/MR			BI/MR			DC/MR		
					min	med	max	min	med	max	min	med	max
SP3	3.3	2.1	2.8	2.5	1.5	10	48	.66	4.1	75	.10	1.3	2.8
SP5	3.0	2.6	2.5	2.3	1.5	12	110	.66	3.8	150	.067	1.1	7.0
SUN	3.8	2.0	2.6	2.4	11	70	710	.91	4.7	180	.35	2.5	4.6
SGI	3.5	3.2	2.7	2.3	2.0	26	180	.75	7.1	310	.17	1.9	11
ITN2	3.0	2.4	2.5	2.3	1.6	15	110	.63	3.3	72	.060	.82	6.0
P4	3.0	2.6	2.5	2.4	1.6	15	92	.50	2.9	91	.058	.86	4.4
X1	2.4	2.0	1.9	2.2	.56	1.8	5.2	.86	3.2	16	.024	.24	.66
OPT	2.9	2.9	2.5	2.2	4.3	41	190	.74	6.3	300	.18	2.1	12

TABLE 4.1

Performance summary when computing all eigenpairs. The 'slope of time trend' refers to the fitting of a straight 'trend' line to the timing data on a log-log plot. If the running time satisfied the simple formula $t = c \cdot n^e$ for some constants c and e, then the measured values of its runtime would lie on the straight line $\log t = \log c + e \cdot \log n$ with slope e when plotted on a log-log plot.

4.2. Performance Details for Practical Matrices on the Opteron. This section studies in-depth the situation on one target architecture, the Opteron. In this section, we call a matrix small whenever its dimension is less than $n = 362$, which marks the largest n for which an n-by-n double precision matrix (of eigenvectors) can fit in the Opteron's 1MB L3 cache.[2]

[2]Note that in this section, because of the cache, we call slightly more matrices 'large' than in Section 4.1 which looks at all architectures simultaneously.

Performance summary on Practical Matrices on Opteron							
		$n < 363$			$n \geq 363$		
Metric	Alg(s)	Min	Median	Max	Min	Median	Max
Time	QR/MR	.57	1.3	4.5	2.6	38	190
Ratios	BI/MR	.75	1.3	4.8	.74	5.8	300
	DC/MR	.34	.46	1.6	.18	2.0	12
Flop	QR/MR	2.4	4.8	12	9.0	74	390
Count	BI/MR	1.2	2.1	6.4	1.1	9.8	380
Ratios	DC/MR	.72	1.0	3.3	.5	8.3	66
GFlop	QR	1.9	2.2	2.5	1.0	1.4	2.0
Rates	BI	.66	.92	1.1	.6	.9	1.3
	DC	1.1	1.4	1.7	1.5	2.9	4.1
	MR	.55	.58	.8	.5	.6	.8
GFlop	QR/MR	2.6	3.5	4.1	1.5	2.1	3.5
Ratios	BI/MR	.83	1.6	2.0	.9	1.6	2.3
	DC/MR	1.7	2.3	2.7	2.5	4.4	6.9

TABLE 4.2
Performance Summary for Practical Matrices on Opteron.

Table 4.2 summarizes the performance of the four algorithms on the Opteron, for eigenvector matrices that fit in cache ($n < 363$) and those that do not ($n \geq 363$).

Figures 4.1 and 4.2 show the run time and flop counts, respectively, of all algorithms on a log-log plot.

The color and symbol code used for all plots is as follows: QR data is blue, using '+', BI data is magenta, using 'x', DC data is red, using 'o', and MR data is black, using diamonds to mark data points.

The straight lines in the plots are least-squares fits to the data of the same color, for $n \geq 363$. The slopes of these lines are shown in the legend of each plot in parentheses after the name of the corresponding algorithm.

The slopes indicate that QR is an $O(n^{2.9})$ algorithm measured by time, and an $O(n^{3.0})$ algorithm measured by flop counts, reasonably matching the expected $O(n^3)$.

The same is true of BI, although we see that the BI data is actually rather more spread out. This is because BI does $O(n)$ flops on eigenvectors with well-separated eigenvalues, and up to $O(n^2)$ work per eigenvector on matrices with tightly clustered eigenvalues.

MR is $O(n^{2.2})$ measured either way, and has the lowest exponent of any algorithm, though slightly higher than the anticipated $O(n^2)$. Given the spread out nature of the data, it is not clear how significant the '.2' part of the slope is.

Interestingly, DC is $O(n^{2.5})$ measured using time and $O(n^{2.8})$ measured using flop counts. As we will see, this is because the MFlop rate for DC increases considerably with dimension. Note that the runtimes for $n < 363$ are increasing somewhat more slowly than these slopes for larger matrices would indicate; these are the dimensions where the output matrix fits in the L3 cache.

Flop counts offer an interesting insight. MR always does fewer flops than QR and BI, up to 390x and 380x times fewer, respectively. MR does up to 66x fewer flops than DC, and never more than twice as many, with a median of 8.3x fewer flops than DC for large matrices. Figure 4.3 shows the flop counts of each algorithm relative to the one of MR. PAPI [15] was used to obtain the flop counts.

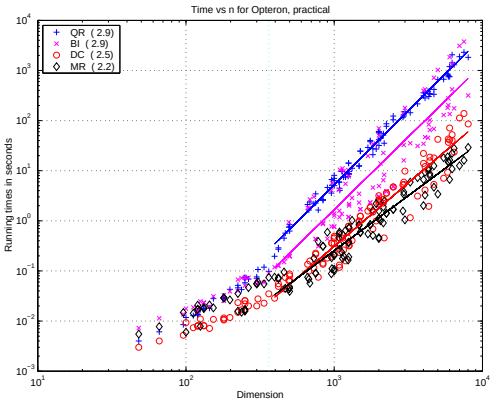

FIG. 4.1. *Runtime of all algorithms on Opteron. The slopes of the least squares fit, shown in parentheses, are computed from the larger matrices.*

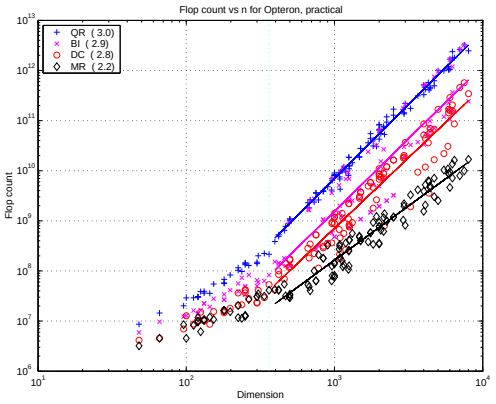

FIG. 4.2. *Flop counts of all algorithms on Opteron. The slopes of the least squares fit, shown in parentheses, are computed from the larger matrices.*

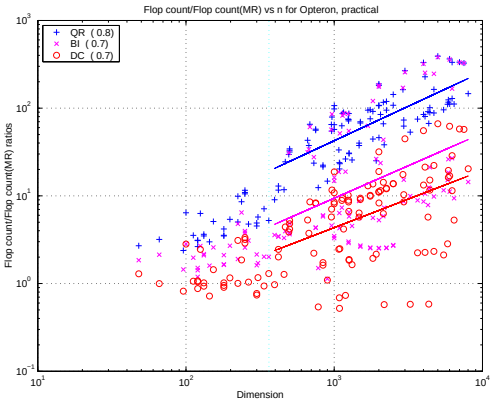

FIG. 4.3. *Flop counts divided by flop counts of MR for Practical Matrices on Opteron.*

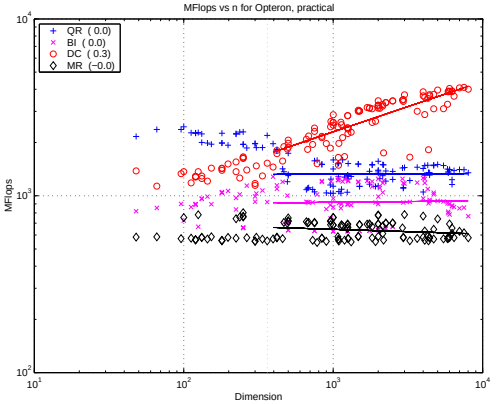

FIG. 4.4. *MFlop Rates for Practical Matrices on Opteron.*

Figure 4.4 shows the MFlop rates. MR generally is the slowest algorithm by this metric. Thus MR does the fewest flops but at higher cost. Indeed, an inspection shows that the number of divides MR performs always exceeds a fixed significant nonzero fraction of the total number of floating point operations, see [27, 14].

It is also natural to ask why QR's MFlop rate drops when n increases past 362, BI's and MR's remains roughly the same for all n, and DC's MFlop rate increases continuously with n. In the case of QR, the algorithm updates pairs of columns of the eigenvector matrix during the course of a QR sweep. This means that many sweeps touch the whole eigenvector matrix, performing just a few floating point operations per memory reference. This BLAS1-like operation on the eigenvector matrix explains the drop in MFlop rate when the eigenvector matrix no longer fits in L3 cache. In contrast, BI and MR compute one eigenvector at a time, so as long as a few vectors of length n fit in cache, there will be minimal memory traffic and the MFlop rate should remain constant.

DC's flop count and MFlop rate are more complicated, and depend on two phe-

nomena unique to DC: the use of BLAS3 to update the eigenvector matrix, and deflation, as discussed in section 2.3. If there is little deflation, most of the flops will be performed in calls to DGEMM on large matrices, so the MFlop rate should increase, as long as the larger matrices on which DGEMM is called result in faster MFlop rates. On the other hand, if there is a lot of deflation, DC will perform many fewer flops, but they will be scalar and not run fast. In the case of practical matrices, the fraction of deflated eigenvalues, fr_{defl}, never exceeds .502, and has a median value of .125. So we expect DC to perform many flops as shown by the slope, $O(n^{2.8})$. It is thus crucial for DC to use the fastest available BLAS library. Table 4.1 shows that on the Cray X1, DC on practical matrices behaves like an $O(n^{1.9})$ algorithm; this is because the speed of BLAS3 increases very rapidly as size of these matrices increases.

4.3. Performance details for synthetic matrices on the Opteron. In this section, we study in-depth how the performance of an algorithm can depend on the matrix at hand, focusing on the differences between DC and the MR. The goal is to push algorithms to their extremes, in the positive and negative sense. Table 4.3 lists the matrices considered. For more details and references, see [7].

Property	Line Color	Line Type	Symbol	Fraction deflated (DC)	
				Minimum	Maximum
More strongly clustered eigenvalues	*blue*		S		
$n-1$ evs at $1/\kappa$, 1 ev at 1		solid	S1	.98	1
$n-1$ evs at 1, 1 ev at $1/\kappa$		dashed	S2	.88	.98
Weakly clustered eigenvalues	*red*		W		
Evs at 1 and $1/\kappa \cdot [1:n-1]$		solid	W1	.34	.81
Evs at $1/\kappa$, 2, and $1+1/\sqrt{\kappa} \cdot [1:n-2]$		dashed	W2	.01	.05
Evs at $1+100/\kappa \cdot [1:n]$		dash-dot	W3	.06	.09
Geometric distributions	*green*		G		
Exactly geometric		solid	G1	.16	.36
Randomly geometric		dashed	G2	.16	.19
Uniform distributions	*black*		U		
Exactly uniform		solid	U1	0	.03
Randomly uniform		dashed	U2	0	.03
Wilkinson W_{m+1}	*cyan*	solid	Wi	.35	.84
Glued Wilkinson		dashed	GW	.59	.78

TABLE 4.3

Selected synthetic test matrices from [7]. Matrices that are generated from a given eigenvalue distribution using the LAPACK tester have a trailing 'p' or 'n' indicating that all eigenvalues are positive or are randomly negated, respectively. κ means the matrix condition number. By default, we choose $\kappa = 1/\varepsilon$, with ε being the working accuracy. For the strongly clustered eigenvalues, we use both $\kappa = 1/\varepsilon$ or $\kappa = 1/\sqrt{\varepsilon}$, indicated by an additional trailing 'e' and 's', respectively. As an example, 'S1ne' refers to a matrix from class S1 with randomly negated eigenvalues and $\kappa = 1/\varepsilon$. The last two rows refer to Wilkinson-type matrices.

The last two columns of Table 4.3 show the fraction of deflations in DC as defined in Section 3.3. For some matrix classes (S,W1,Wi,GW), the fraction deflated is significant, sometimes even close to 1, while for others (U,W2,W3), there are nearly no deflations.

For cases with a significant amount of deflation, we expect a noticeable speedup for DC. Moreover, Wilkinson and glued Wilkinson matrices are known to be notoriously difficult for MR due to strongly clustered eigenvalues [13]. Thus we expect to see MR perform poorly on these classes of matrices. To verify our intuition, we show least-square-fits of time and flop counts normalized by n^2 separately for each matrix class. Figures 4.5 and 4.6 show the data for DC, and Figures 4.7 and 4.8 for MR. Note that the vertical axes in Figures 4.5 and 4.7 are the same, going from 10^{-8} to

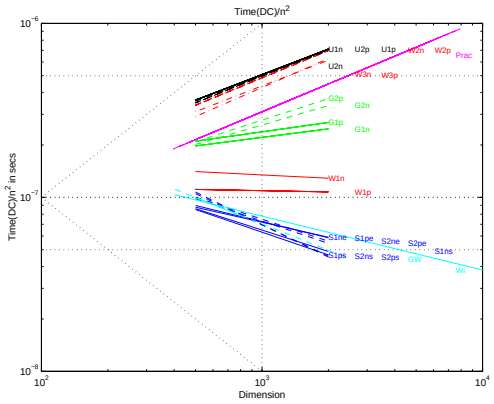

FIG. 4.5. *Performance trend lines of DC for runtime divided by n^2 on Opteron.*

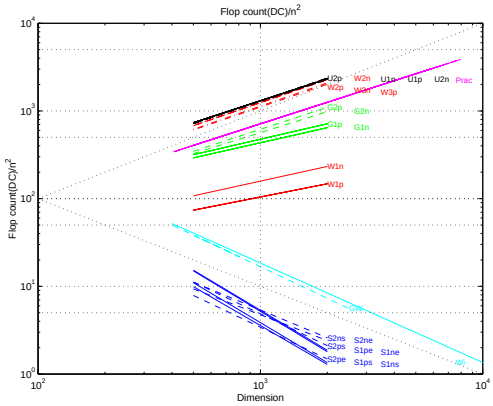

FIG. 4.6. *Performance trend lines of DC for flop counts divided by n^2 on Opteron.*

10^6. However, the vertical axis for MR in Figure 4.8 is ten times lower than the one for DC in Figure 4.6. For reference, we also added 'Prac', the practical matrices from Section 4.2. The dotted black lines indicate the slopes +1 and -1, in order to better recognize slopes of the trend lines.

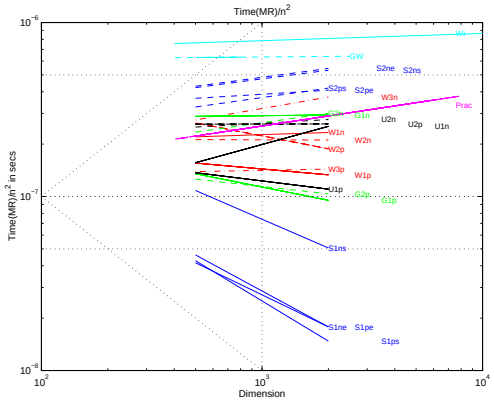

FIG. 4.7. *Performance trend lines of MR for runtime divided by n^2 on Opteron.*

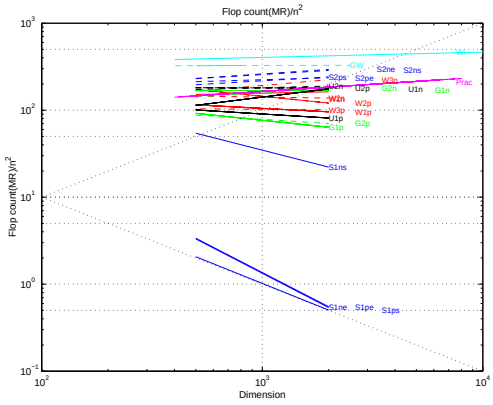

FIG. 4.8. *Performance trend lines of MR for flop counts divided by n^2 on Opteron.*

The figures exhibit how much the performance of both algorithms depends on the matrix class and can vary dramatically.

Figure 4.6 shows that DC either does close to an $O(n^3)$ flops (trend lines nearly parallel to the upper diagonal dotted black line) or does close to $O(n)$ flops (the other trend lines), when deflation is extreme. It is interesting to compare to the runtime of DC shown in Figure 4.5. As noted in Section 4.2, the majority of flops in DC is done using BLAS 3 so that the trend lines for the timings do not completely align with those for the flop counts.

The nearly horizontal trend lines of MR in Figure 4.8 indicate that the algorithm does close to $O(n^2)$ flops for the corresponding matrix classes. Runtime trends do fairly well correspond to the flop trends here.

We now report on some of the classes individually. Our examples are classes

where

- both DC and MR are much faster than on practical matrices, Figure 4.9 showing 'Strongly Clustered' Matrices with Positive Eigenvalues (S1pe),
- MR is much faster than DC, Figure 4.10 showing 'Uniformly Distributed' Matrices with Positive Eigenvalues (U2pe), and
- DC is much faster than MR, Figure 4.11 showing Glued Wilkinson Matrices (GW).

Each of the the Figures has six subplots. The rows, from top to bottom, show run time, flop counts, and MFlop rate. The left column shows the data (except for the last one, the MFlop rate) normalized by n^2, the right column presents the data relative to MR results.

For the 'strongly clustered' matrices in Figure 4.9, the fraction of eigenvalues deflated in DC is at least 88% and often 100%. This makes DC appear to be much faster than $O(n^2)$. In fact both DC and MR perform much faster for these matrices than they do on practical matrices of the same size.

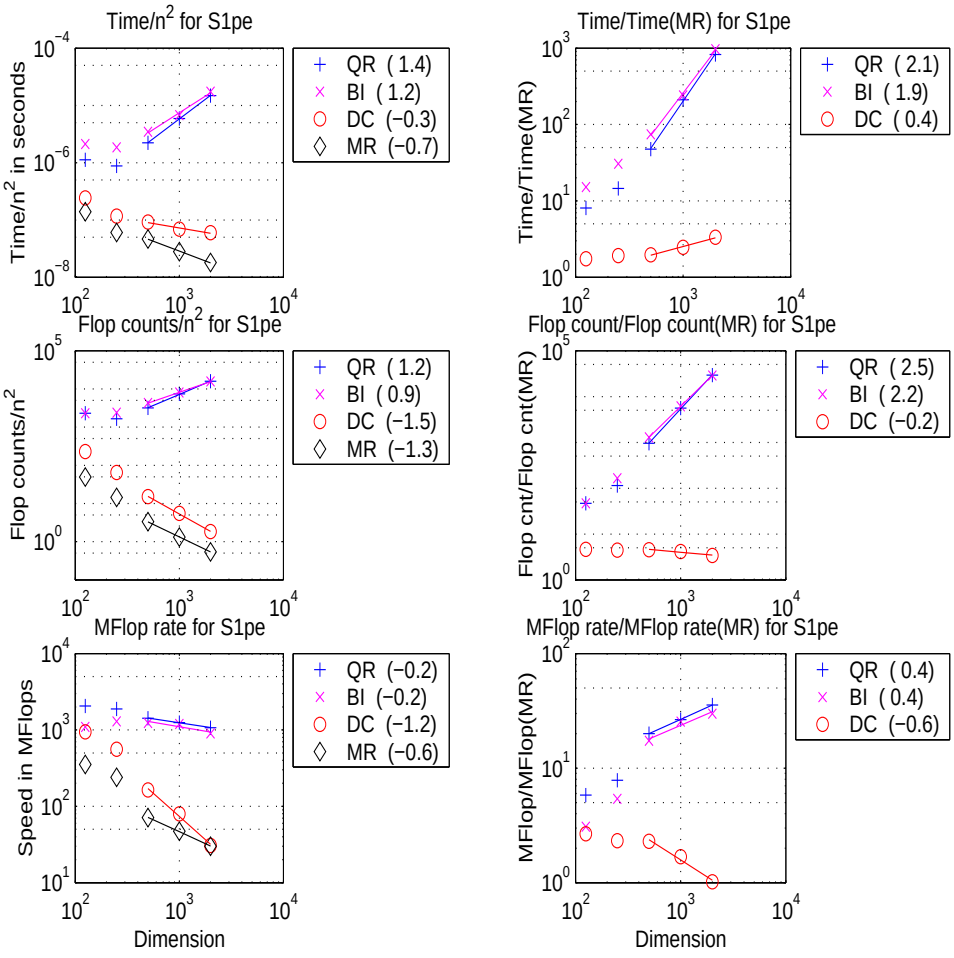

FIG. 4.9. *Performance data for Strongly Clustered Matrices with Positive Eigenvalues (S1pe) on Opteron.* $\kappa = 1/\varepsilon$.

DC runs uniformly slower on the 'uniformly distributed' matrices than on practical matrices. One example is shown in Figure 4.10. Note that the fraction deflated in DC is less than 3%. This means that the algorithm performs $O(n^3)$ work, at the speed of DGEMM. Several other matrix classes with similarly few deflations are given in [7]. Classical orthogonal polynomials such as Chebyshev, Legendre, Laguerre, and Hermite that are defined by three-term recurrence give rise to symmetric tridiagonal matrices with very small amounts of deflation in DC. It is interesting that on the other hand, these matrices pose no difficulties for MR: their eigenvalues are not very strongly clustered. For such matrices, as for the Uniformly distributed ones in Figure 4.10, MR can run much faster than DC.

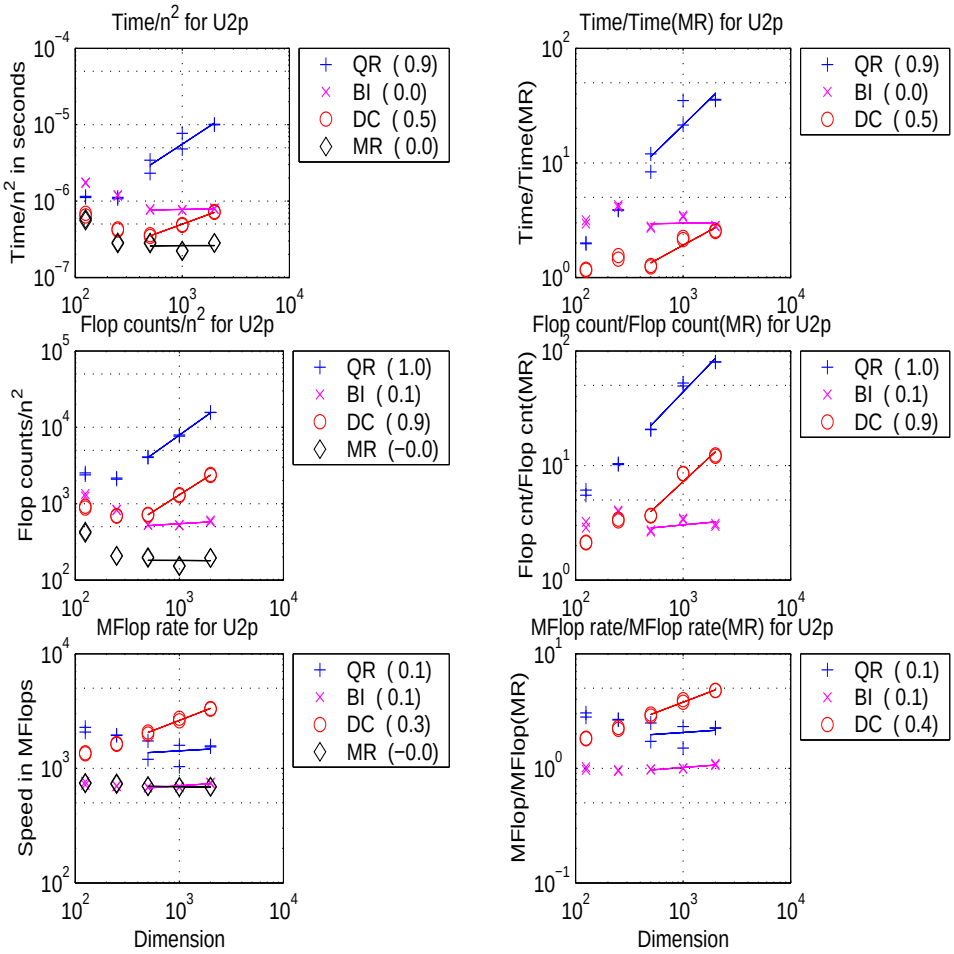

FIG. 4.10. *Performance data for Uniformly Distributed Matrices with Positive Eigenvalues (U2pe) on Opteron.*

Finally, the classes on which DC performs nearly fastest, and on which MR performs worst, are the Wilkinson and Glued Wilkinson matrices. The results for the latter class are shown in Figure 4.11. The difficulties of MR for these matrices are well understood: since the eigenvalues of glued matrices come in groups of small size but extreme tightness, the representation tree generated by the MR algorithm is very

broad and the overhead for the tree generation is considerable, see [13, 7]. On top of the difficulties of MR, the fraction deflated in DC is $\in [59\%, 78\%]$, that is DC is extraordinarily efficient and even faster than for practical matrices.

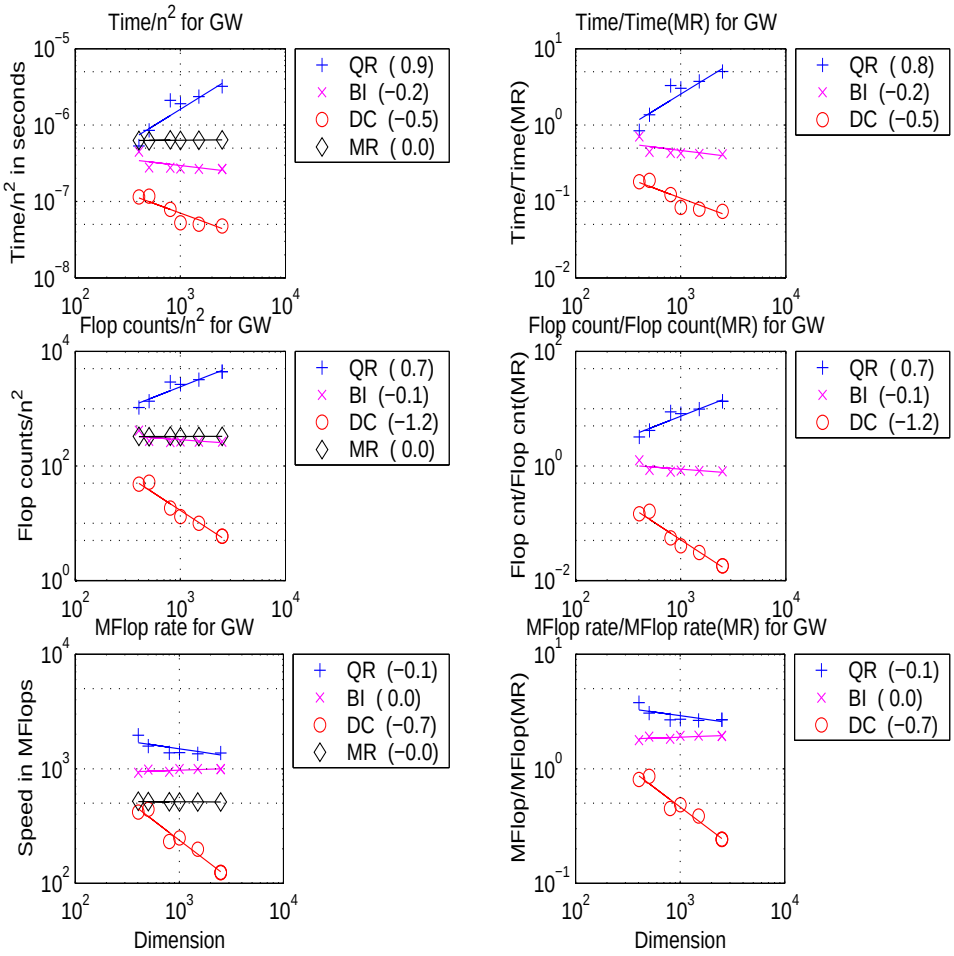

FIG. 4.11. *Performance data for Glued Wilkinson Matrices (GW) on Opteron.*

5. Subset computations: performance. Table 5.1 shows timing statistics of BI relative to MR, for all test matrices. For each matrix, we randomly chose 15 subsets by index range ('find eigenvalues IL:IU') and 15 subsets by intervals ('find eigenvalues in [VL,VU]').

Performance Summary for Subset Computations						
Mach	By index			By interval		
	min	med	max	min	med	max
SP3	.24	1.3	8.5	.22	1.3	9.0
SP5	.22	1.2	6.5	.20	1.2	6.8
SUN	.38	1.7	25.2	.32	1.7	27.4
SGI	.30	1.3	15.2	.29	1.4	14.2
ITN2	.24	1.1	4.7	.22	1.1	4.9
P4	.19	1.1	7.2	.16	1.1	7.8
X1	.39	1.4	4.3	.31	1.5	4.5
OPT	.30	1.3	21.1	.28	1.3	16.7

TABLE 5.1

Performance summary when computing subsets of eigenpairs, either by index ('eigenvalues IL:IU') by interval ('eigenvalues in [VL,VU]'). Shown are the results of BI relative to MR.

One can see that there are subsets for which BI is up to six times faster than MR, whereas there are others intervals where it is 27 times slower than MR. [3] This shows that performance depends on the subset. The medians tell that MR on average is faster and thus preferable to BI for subsets.

It is interesting to investigate what matrices show the biggest differences in runtime. MR does significantly better than BI on subsets of G. Fann's practical matrices. This should not be a surprise as it is known BI require a large number of reorthogonalizations for these matrices, see the remarks in Section 3.1 and [8, 10].

BI's relatively best performances occur in two different cases. First, there are the 'very easy' tests where BI does not require any reorthogonalization and both BI and MR are very fast. More interesting, in a pathological sense, are the Wilkinson and Glued Wilkinson matrices. As noted in [13], MR requires a substantial amount of work to deal with the extremely strong clustering of the eigenvalues and its overhead turns out to be more significant than the reorthogonalization required by BI.

6. Accuracy.

6.1. Residual norm and loss of orthogonality. Using the metrics for orthogonality loss and residual norm in (3.1) and (3.2), respectively, the worst case residuals and losses of orthogonality for all matrices and all platforms are reported in Table 6.1.

Some plots detailing the accuracy on the Opteron are given in Figure 6.1.

The important trends are that the errors decrease as n increases for DC and QR. Given the n in the denominators of the above formulas, this means that the DC and QR errors do not increase proportionally to n. This is confirmed by the slopes of the trend lines which are shown in the legends. In general, inverse iteration and MR do not achieve the same level of accuracy as QR and DC, their accuracy is $O(n\varepsilon)$ in general. For matrices that split into smaller blocks, the block size governs the orthogonality loss rather than the matrix dimension.

[3] There is some variation in the maximum ratios across the machines which we cannot explain satisfactorily, so the largest ratios should be considered with a grain of salt.

Worst case error for all matrices								
Mach	Residual				Orthogonality loss			
	QR	BI	DC	MR	QR	BI	DC	MR
SP3	.23	100	.13	22	.34	140	.25	70
SP5	.30	59	.13	18	.39	70	.25	163
SUN	.20	331	.11	14	.45	440	.19	92
SGI	.30	210	.13	14	.46	280	.19	160
ITN2	.30	240	.13	29	.45	320	.19	190
P4	.30	39	.13	33	.38	53	.19	140
X1	.30	34	.11	80	.46	45	.19	160
OPT	.30	100	.11	14	.46	130	.19	160

TABLE 6.1

Summary of all accuracy results. Reported is the worst result that was observed on any practical or testing matrix. Residual norms and level of orthogonality are given as multiples of $n\varepsilon$, see Section 3.3 on error metrics.

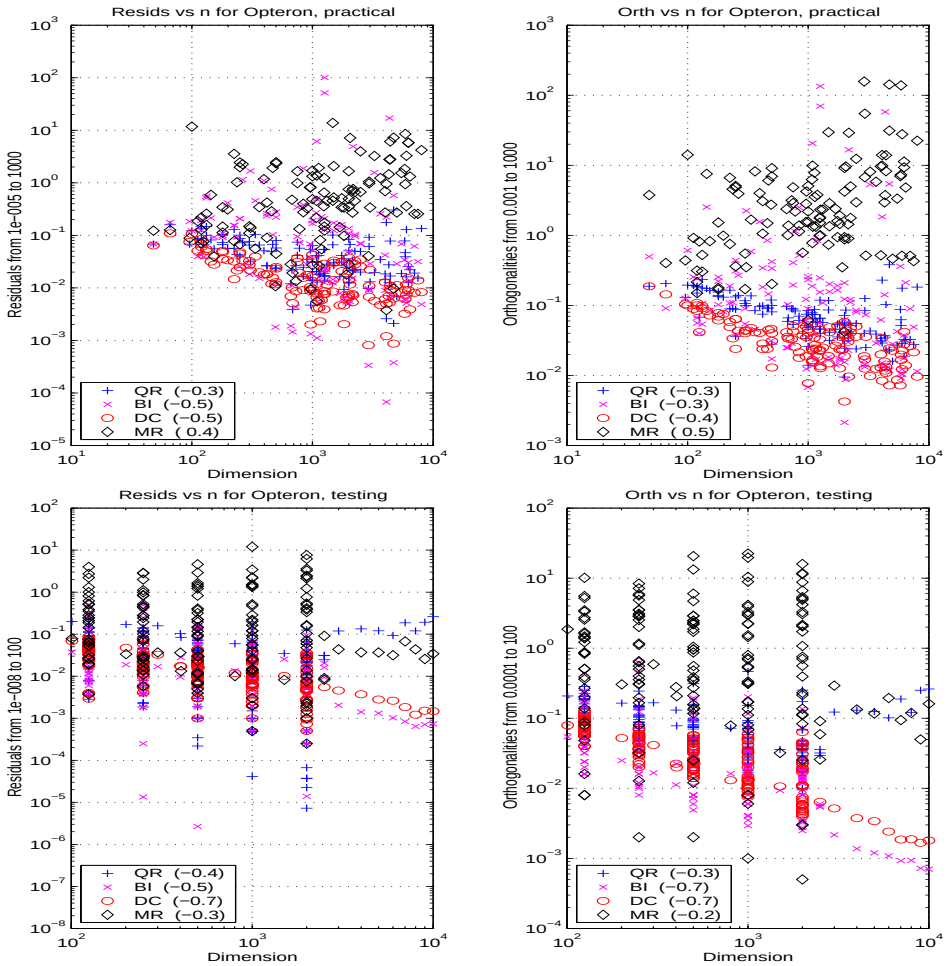

FIG. 6.1. *Residuals and Losses of Orthogonality for all matrices on Opteron. (Top: all practical matrices. Bottom: all synthetic 'testing' matrices. Note the difference in vertical scales on top and bottom.)*

6.2. Reliability: comparing MR from LAPACK 3.1 to version 3.0. For the LAPACK 3.1 release, we have done extensive development on MR. It now allows the computation of subsets of eigenpairs [26]. Moreover, the new algorithm is significantly more reliable than the version in LAPACK 3.0, which had very large errors on a significant subset of our test matrices.

Figure 6.2 shows that the version of MR tested here is more accurate than the old MR from LAPACK 3.0, which not only had numerous large errors as shown in the plot, but failed to return any answer on 22 of our test matrices, including 9 practical matrices.

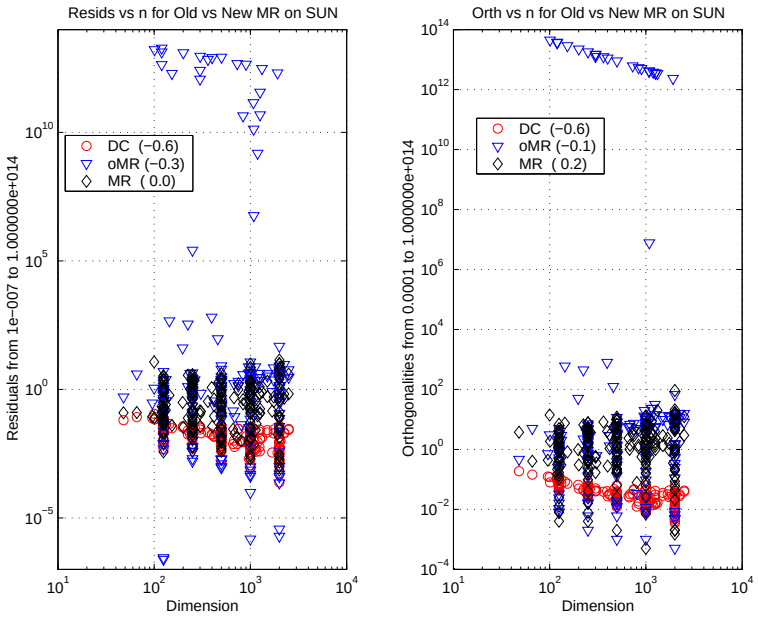

FIG. 6.2. *Residual norms and loss of orthogonality. Failures of the LAPACK 3.0 version of MR have been corrected for version 3.1.*

There are two reasons for the improved reliability and accuracy of the 3.1 version of MR. First, it includes a remedy to the recently discovered problem that for a tridiagonal where no off-diagonal entries satisfy the splitting criterion, the eigenvalues can still be numerically indistinguishable down to the underflow threshold, see [13]. Second, the internal accuracy threshold on relative gaps has been tightened. This threshold is directly proportional to the upper bounds on numerical residual and orthogonality of the computed vectors, see [14]. Instead of a threshold of one over the matrix dimension, a fixed threshold of 10^{-3} in double precision is used. This makes the new MR more accurate on larger matrices.

7. Summary and conclusions. In preparation for the the latest LAPACK 3.1 release, we performed a systematic study of performance and accuracy of the computation of eigenpairs for a symmetric tridiagonal matrix. Our evaluation considers speed and accuracy of QR iteration (QR), bisection and inverse iteration (BI), the Divide-and-Conquer method (DC), and the method of Multiple Relatively Robust Representations (MR) when computing all, or a subset of, eigenpairs of a variety of matrices on today's computer architectures. Our conclusions are as follows.

1. DC and MR are generally much faster than QR and BI on large matrices. MR is faster than DC on large practical matrices (i.e. the median value of the ratio runtime(DC) over runtime(MR) exceeds one) on 5 of our 8 platforms, and is slower than DC on 3 platforms. For matrix classes for which there is a great deal of deflation, DC becomes much faster.
2. Using hardware performance counters to count floating point operations on the Opteron, we discover that MR almost always does the fewest floating point operations, but at a lower MFlop rate than all the other algorithms. This is because it performs more divides than the other algorithms: the number of divides MR performs always exceeds a fixed significant nonzero fraction of the total number of floating point operations, whereas the fraction of divides for the other algorithms approaches zero as the dimension grows. DC has a MFlop rate that grows significantly with n (as proportionally more operations are done using BLAS3 DGEMM operations). This increase in DC's MFlop rate is enough to make DC look like an $O(n^{2.5})$ algorithm as determined empirically by fitting a straight line to the log of the running time, even though it is an $O(n^{2.8})$ algorithm as determined by the operation count. QR's MFlop rate drops when the eigenvector matrix is too large to fit in cache. Its use of memory, repeatedly sweeping over the eigenvector matrix performing Givens rotations, with just 1.5 flops per memory reference, is inefficient. BI's complexity depends on the amount of reorthogonalization, that is the eigenvalue clustering.
3. QR and DC are the most accurate algorithms, measured both in terms of producing pairwise orthogonal eigenvectors and small residuals norms $\|Tx-\lambda x\|$. MR is less accurate but still achieves errors of size $O(n\varepsilon)$, where n is the dimension and ε is machine epsilon, never more than $190n\varepsilon$ loss of orthogonality and $80n\varepsilon$ residuals for any matrix on any platform. Depending on the matrix and platform, it is known that BI may completely fail to guarantee orthogonal eigenvectors [9], though this is rare and did not occur on any of our test matrices.
4. MR is preferable to BI for subset computations. Independent of the architecture, the median of the relative time of BI/MR exceeds one on all architectures.
5. The LAPACK 3.1 version of MR addresses some reliability issues of version 3.0.

For computing all eigenpairs of a symmetric tridiagonal matrix, QR, BI and MR use the least memory ($n^2 + O(n)$), whereas DC uses about twice as much ($2n^2 + O(n)$). Looking at the dense symmetric eigenvalue problem, QR needs $n^2 + O(n)$, MR and BI need $2n^2 + O(n)$, and DC needs $3n^2 + O(n)$. If memory is not an obstacle, the choice between DC and MR is matrix-dependent. However, unless the performance differences are substantial, they will be masked in the dense case by reduction to tridiagonal form and backtransformation of the eigenvectors. DC always is the algorithm of choice when the superior accuracy matters. When computing a subset of the eigenvalues and vectors, MR is the algorithm of choice over BI.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, 3. edition, 1999.

[2] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and Henk van der Vorst. *Templates for the solution of algebraic eigenvalue problems - A practical guide*. SIAM, Philadelphia, 2000.
[3] J. Bunch, P. Nielsen, and D. Sorensen. Rank-one modification of the symmetric eigenproblem. *Numer. Math.*, 31:31–48, 1978.
[4] J. J. M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.*, 36:177–195, 1981.
[5] T. A. Davis. University of florida sparse matrix collection. NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996, and NA Digest, vol. 97, no. 23, June 7, 1997.
[6] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, USA, 1997.
[7] J. W. Demmel, O. A. Marques, B. N. Parlett, and C. Vömel. A Testing Infrastructure for Symmetric Tridiagonal Eigensolvers. Technical report LBNL-61831, Lawrence Berkeley National Laboratory, 2006.
[8] I. S. Dhillon. *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*. PhD thesis, University of California, Berkeley, California, 1997.
[9] I. S. Dhillon. Current inverse iteration software can fail. *BIT*, 38:4:685–704, 1998.
[10] I. S. Dhillon, G. Fann, and B. N. Parlett. Application of a new algorithm for the symmetric eigenproblem to computational quantum chemistry. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1997.
[11] I. S. Dhillon and B. N. Parlett. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Linear Algebra and Appl.*, 387:1–28, 2004.
[12] I. S. Dhillon and B. N. Parlett. Orthogonal eigenvectors and relative gaps. *SIAM J. Matrix Anal. Appl.*, 25(3):858–899, 2004.
[13] I. S. Dhillon, B. N. Parlett, and C. Vömel. Glued matrices and the MRRR algorithm. *SIAM J. Sci. Comput.*, 27(2):496–510, 2005. Revised version of LAPACK Working Note 163.
[14] I. S. Dhillon, B. N. Parlett, and C. Vömel. The design and implementation of the MRRR algorithm. *ACM Trans. Math. Software*, 32(4):533–560, 2006.
[15] J. J. Dongarra, S. Moore, P. Mucci, K. Seymour, D. Terpstra, and H. You. Performance Application Programming Interface (PAPI).
[16] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.
[17] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (release I). Technical Report RAL-TR-92-086, Atlas Centre, Rutherford Appleton Laboratory, 1992.
[18] I. S. Duff, R. G. Grimes, and J. G. Lewis. The Rutherford-Boeing Sparse Matrix Collection. Technical Report RAL-TR-97-031, Atlas Centre, Rutherford Appleton Laboratory, 1997. Also Technical Report ISSTECH-97-017 from Boeing Information & Support Services and Report TR/PA/97/36 from CERFACS, Toulouse.
[19] E. Apra et al. NWChem, a computational chemistry package for parallel computers, version 4.7. Technical report, Pacific Northwest National Laboratory, Richland, WA. USA, 2005.
[20] R. A. Kendall et al. High Performance Computational Chemistry: An overview of NWChem a distributed parallel application. *Computer Phys. Comm.*, 128:260–283, 2000.
[21] G. H. Golub and C. van Loan. *Matrix Computations*. The John Hopkins University Press, Baltimore, Maryland, 3. edition, 1996.
[22] M. Gu and S. C. Eisenstat. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM J. Matrix Anal. Appl.*, 16(1):172–191, 1995.
[23] N. J. Higham. Algorithm 694: A Collection of Test Matrices in MATLAB. *ACM Trans. Math. Software*, 17(3):289–305, 1991.
[24] I. C. F. Ipsen. Computing an eigenvector with inverse iteration. *SIAM Review*, 39(2):254–291, 1997.
[25] Lapack 3.1. http://www.netlib.org/lapack/lapack-3.1.0.changes, 2006.
[26] O. A. Marques, B. N. Parlett, and C. Vömel. Computations of Eigenpair Subsets with the MRRR algorithm. *Numerical Linear Algebra with Applications*, 13(8):643–653, 2006.
[27] O. A. Marques, E. J. Riedy, and C. Vömel. Benefits of IEEE-754 Features in Modern Symmetric Tridiagonal Eigensolvers. *SIAM J. Sci. Comput.*, 28(5):1613–1633, 2006.
[28] B. N. Parlett. *The Symmetric Eigenvalue Problem*. SIAM Press, Philadelphia, PA, 1998.
[29] B. N. Parlett and I. S. Dhillon. Fernando's solution to Wilkinson's problem: an application of double factorization. *Linear Algebra and Appl.*, 267:247–279, 1997.
[30] B. N. Parlett and I. S. Dhillon. Relatively robust representations of symmetric tridiagonals. *Linear Algebra and Appl.*, 309(1-3):121–151, 2000.
[31] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, Oxford, 1965.