# The Algorithm of Multiple Relatively Robust Representations for Multi-Core Processors

M. Petschow and P. Bientinesi

Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111 is gratefully acknowledged.

# The Algorithm of Multiple Relatively Robust Representations for Multi-Core Processors

Matthias Petschow[1] and Paolo Bientinesi[1]

RWTH Aachen University, 52062 Aachen, Germany,
`petschow@aices.rwth-aachen.de`,
WWW home page: `http://www.aices.rwth-aachen.html`

**Abstract.** The algorithm of Multiple Relatively Robust Representations (MRRR or MR$^3$) computes $k$ eigenvalues and eigenvectors of a symmetric tridiagonal matrix in $O(nk)$ arithmetic operations. Large problems can be effectively tackled with existing distributed-memory parallel implementations of MRRR; small and medium size problems can instead make use of LAPACK's routine `xSTEMR`. However, `xSTEMR` is optimized for single-core CPUs, and does not take advantage of today's multi-core and future many-core architectures. In this paper we discuss some of the issues and trade-offs arising in the design of MR$^3$–SMP, an algorithm for multi-core CPUs and SMP systems. Experiments on application matrices indicate that MR$^3$–SMP is both faster and obtains better speedups than all the tridiagonal eigensolvers included in LAPACK and Intel's Math Kernel Library (MKL).

**Keywords:** MRRR algorithm, tridiagonal eigensolver

## 1 Introduction

Given a hermitian matrix $A \in \mathbb{C}^{n\times n}$, the eigenproblem is finding solutions to the equation $Av = \lambda v$ with $\|v\| = 1$, where $\lambda \in \mathbb{R}$ is called an *eigenvalue* and $v \in \mathbb{C}^n$ an associated *eigenvector*. An eigenvalue together with an associated eigenvector are said to form an *eigenpair*. The eigenproblem has $n$ solutions, and since $A$ is hermitian, all the eigenvalues are real and the eigenvectors can be chosen mutually orthogonal.

Computing all the eigenpairs is equivalent to finding a matrix factorization $A = V\Lambda V^*$, where $\Lambda \in \mathbb{R}^{n\times n}$ is a diagonal matrix containing the eigenvalues as elements, and $V \in \mathbb{C}^{n\times n}$ is a unitary matrix whose columns are the associated eigenvectors. In this paper we focus on the parallel computation of a subset or all the eigenpairs of a real symmetric tridiagonal matrix.

Several efficient and accurate methods exist for the symmetric tridiagonal eigenproblem. Among them, Bisection and Inverse Iteration (BI) [1], the QR algorithm (QR) [2, 3], Divide & Conquer (DC) [4, 5], and the algorithm of Multiple Relatively Robust Representations (MRRR) [6]. Until the introduction of the latter, the computation of all the eigenpairs required $O(n^3)$ flops in the worst case. With the MRRR algorithm it is instead possible to compute all the eigenpairs in

$O(n^2)$ flops. Moreover, similar to the method of Inverse Iterations, MRRR allows the computation of a subset of the eigenpairs at reduced cost: $(nk)$ flops for $k$ eigenpairs. In fact, the MRRR algorithm can be seen as a sophisticated variant of Inverse Iteration that does not require explicit orthogonalization, hence the quadratic complexity. An informative and detailed performance analysis of LAPACK's [7] implementations of the four algorithms can be found in [8].

As multi-core architectures have replaced uni-processors, our goal is to explore how MRRR, already the fastest sequential algorithm, can make efficient use of today's multi-core and future many-core CPUs. A representative example is given in Fig. 1 *(left)*, where the execution time for a matrix of size $n = 4,289$ from quantum chemistry is shown as a function of the number of threads used. We present results for four routines: MKL's DC (`DSTEDC`), both MKL's and LAPACK's sequential MRRR (`DSTEMR`), and MR$^3$–SMP, the multi-core variant of the MRRR algorithm that we present in this paper.[1] BI with 487 seconds and QR with timings between 167 and 61 seconds are much slower and are not shown in the graph.[2] While DC casts most of the work in terms of `DGEMM` and can take advantage of parallelism by multi-threaded BLAS [9], MRRR's `DSTEMR` is sequential and therefore does not exploit any of parallelism of multi-core processors. As a result, DC can become faster than the sequential MRRR as the amount of available parallelism increases.

While tridiagonal matrices are common in applications, they play a much bigger role as part of dense and banded eigensolvers. In these cases, when many of the eigenpairs are to be computed, the most common approach to solve the eigenproblem consists of three stages: 1) Reduction of $A$ to a real symmetric tridiagonal matrix $T = U^*AU$ via a unitary matrix $U \in \mathbb{C}^{n \times n}$; 2) Solution of the symmetric tridiagonal eigenproblem $Tz = \lambda z$; 3) Back-transformation of the eigenvectors of $T$ into those of $A$ by $v = Uz$.

When MRRR is used for the tridiagonal stage, the reduction becomes the computational bottleneck in this procedure, requiring about $\frac{16}{3}n^3$ floating point operations (flops), only half of which can be cast in terms of fast BLAS-3 kernel routines. In contrast, the about $8n^3$ flops required by the back-transformation stage can be performed efficiently, since the computation can be casted almost entirely in terms of BLAS-3 routines.

Although negligible in a sequential execution, the time spent in the tridiagonal eigensolver becomes significant when multiple cores are used. Fig. 1 *(right)* shows the fraction of the total execution time that is spent on each of the three stages of the dense symmetric eigenproblem of size $n = 4,289$ for a varying number of threads. In the single-threaded execution, the tridiagonal eigensolver is indeed negligible, while in the multi-threaded executions it can take up to 40% of the total execution time. In contrast, with 24 threads our multi-core parallel algorithm MR$^3$–SMP accounts for about 7% of the execution time.

[1] More detailed information about the parameter of the experiment can be found section 4.

[2] For the timings of BI and QR Intel MKL 10.2 was used.

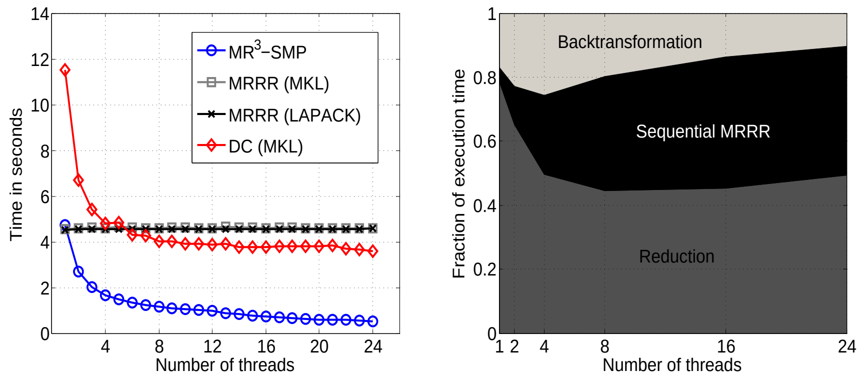

**Fig. 1.** *Left:* Timings as function of the number of threads used in the computation. Qualitatively, the graph is typical for the applications matrices that we tested. The Divide & Conquer algorithm becomes equally fast or even faster than the sequential MRRR algorithm. MR$^3$–SMP however is faster and obtains better speedups than DC. *Right:* Fraction of time spent in the solution of the corresponding dense symmetric eigenproblem for the reduction, tridiagonal eigenproblem, and back-transformation.

The paper is organized as follows: Section 2 contains a brief discussion of the MRRR algorithm. In Section 3 the design of the MRRR algorithm for shared-memory computer systems, MR$^3$–SMP, is described. In Section 4 the results of experiments evaluating the performance of MR$^3$–SMP are shown.

## 2 The MRRR Algorithm

In this section the algorithm of Multiple Relatively Robust Representations is briefly discussed. A detailed description and justification of the algorithm can be found in [6] and references therein.

Without loss of generality, the symmetric tridiagonal matrix $T \in \mathbb{R}^{n \times n}$ is assumed to be irreducible. That is, no off-diagonal element is smaller in magnitude than a certain threshold that warrants setting it to zero. Possible quantities for such a threshold are discussed in [10]. Otherwise, when $T$ is reducible, each diagonal block can be treated separately.

The algorithm starts by computing a factorization of $T$, called a *Relatively Robust Representation* (RRR). From this factorization eigenvalue approximations are computed and finally the associated eigenvectors together with the possibly refined eigenvalue.

An RRR is a representation of $T$ that has the property that small relative changes in its non-trivial entries only cause small relative changes in a specific set of eigenvalues [11]. Such an RRR is given by bidiagonal factorizations $T - \sigma I = LDL^T$ with $\sigma \in \mathbb{R}$, where $L$ is lower unit bidiagonal and $D$ is diagonal. The condition for the factorization to be an RRR is given in [11]. However, in the special case that $T - \sigma I = LDL^T$ is definite, the factorization is guaranteed to be an RRR for all eigenvalues.

After computing an RRR for all desired eigenvalues $\lambda_j$, it is possible to compute approximations $\hat{\lambda}_j$ to high relative accuracy, that is $|\lambda_j - \hat{\lambda}_j| = O(\varepsilon|\hat{\lambda}_j|)$, where $\varepsilon$ denotes the machine precision. This can be achieved using $O(n)$ flops via bisection using a differential quotient-difference transform [12]. In the definite case, all eigenvalues can be computed to high relative accuracy via the dqds-algorithm in $O(n^2)$ arithmetic operations [13].

Once an eigenvalue $\hat{\lambda}_j$ is computed to high relative accuracy, the associated eigenvector $\hat{z}_j$ is computed by solving $(LDL^T - \hat{\lambda}_j I)\hat{z}_j = \gamma_r e_r$, where the right hand side of the system is a multiple of the $r$-th standard basis vector. One of the features of the MRRR algorithm is that it is possible to compute an eigenvector $\hat{z}_j$, with $\|\hat{z}_j\|_2 = 1$, such that the residual norm satisfies

$$\|(LDL^T - \hat{\lambda}_j I)\hat{z}_j\|_2 = O(n\varepsilon|\hat{\lambda}_j|) \ . \tag{1}$$

By the gap theorem of Davis and Kahan [14] the error angle to the true eigenvector $z_j$ is bounded by

$$|\sin \angle(\hat{z}_j, z_j)| \leq \frac{O(n\varepsilon)}{\mathrm{relgap}(\hat{\lambda}_j)} \ , \tag{2}$$

where the relative gap of $\hat{\lambda}_j$ is defined as $\mathrm{relgap}(\hat{\lambda}_j) := \min_{i \neq j}(|\hat{\lambda}_j - \lambda_i|/|\hat{\lambda}_j|) = \mathrm{gap}(\hat{\lambda}_j)/|\hat{\lambda}_j|$. This definition and (2) imply that, provided the gap of an eigenvalue $\hat{\lambda}$ is of the same order as its magnitude, the computed eigenvector $\hat{z}_j$ has an error angle of $O(n\varepsilon)$ to the real eigenvector of the RRR. In practice, when $\mathrm{relgap}(\hat{\lambda}_j) \geq tol$ the eigenvalue is called a *singleton* and the eigenvector is computed from the RRR.

Eigenvalues that are not singletons form *clusters*. For each cluster a new RRR for the eigenvalues in the cluster is computed via the differential stationary qds (dstqds) transform [12]: $L_i D_i L_i^T = LDL^T - \sigma_i I$. The parameter $\sigma_i \in \mathbb{R}$ is thereby chosen in a way that at least one of the eigenvalues in the cluster becomes a singleton. The shifted eigenvalues $\hat{\lambda}_j - \sigma_i$ must be refined to relative accuracy with respect to the new RRR. Full accuracy is only needed for the singletons, so that eigenvectors with small relative residual norm can be computed. This procedure is than applied recursively until all eigenvectors are computed.

A main feature of the MRRR algorithm is that, although the eigenvectors might be computed from different RRRs, they are numerically orthogonal and no Gram-Schmidt procedure for orthogonalization has to be invoked. This property is discussed in detail in [6], where it is shown that the computed quantities satisfy

$$\|(T - \hat{\lambda}_j I)\hat{z}_j\| = O(n\varepsilon\,\|T\|) \text{ and } |\hat{z}_k^T \hat{z}_j| = O(n\varepsilon), \ \ k \neq j \ . \tag{3}$$

## 3 The MRRR Algorithm for Multi-Core Processors

In this section we discuss the design of MR$^3$–SMP, a parallel version of the MRRR algorithm specifically designed for multi-core and shared-memory architectures. One of the salient features of such systems is the capability of communication among processors at low costs thanks to shared caches and memory.

As a consequence, both redundancy and costly data exchanges, characteristic to distributed-memory parallelizations [15, 16], now can and should be avoided in favor of a fine grain parallelism.

MR$^3$–SMP is based on the routine `DSTEMR` of LAPACK Version 3.2 and makes use of *POSIX threads* for parallelization. A detailed description of `DSTEMR` and its design criteria can be found in [17].

## 3.1 Parallelization Strategy

MR$^3$–SMP achieves parallelism by dynamically dividing the computation into independent tasks. The tasks are placed into a work queue and can be executed by many threads in parallel. This form of parallelism may produce a bigger overhead than a static division of the work, but it is flexible and attains good load balancing among the processors.

After computing the root representation[3], the initial eigenvalue approximations are either computed by the dqds-algorithm or by parallel bisection, depending on the amount of parallelism available and the number of eigenvalues to compute. Bisection is used, when the number of cores $c$ is greater than $12 \cdot \#eigenvalues/n$ [15]. The dqds-algorithm computes the eigenvalues to full accuracy, while bisection only does so when only the eigenvalues are desired.

The computation of the eigenvectors and a gradual refinement of the eigenvalues can be represented in form of a representation tree. The associated work can be naturally divided into tasks: Each node consists of an index set $\Gamma_p$ of associated eigenpairs and depends on the RRR of its parent node. The computation that has to be executed depends entirely on the size of the index set $|\Gamma_p|$. In the case of $|\Gamma_p| = 1$ the node is a leaf node and represents a singleton. Otherwise, in the case $|\Gamma_p| > 1$, the node is considered a cluster. For both cases, singletons and clusters, we created a task type. A third task type is introduced to add the ability of splitting the work of refining eigenvalues into tasks. The three task types will be called *S-task*, *C-task* and *R-task* subsequently. The work associated to each task is discussed next:

1. ***S-task:*** As described in Section 2, the eigenvectors associated to singletons can be computed immediately. This leads to the following computational task: For a set of singletons $\Gamma_s \subseteq \Gamma_p$, compute the eigenvalues to high relative accuracy and the associated eigenvectors. This is done via Inverse Iteration with twisted factorizations and Rayleigh Quotient correction [17].
2. ***C-task:*** A task is created for each cluster $\Gamma_c \subset \Gamma_p$: Compute a new RRR for the eigenvalues in the cluster and refine the eigenvalues to relative accuracy with respect to the new RRR until a distinction between singletons and clusters is possible. At this point the new representation can be used to partition the computation into tasks recursively, that is creating S-tasks and C-tasks with $\tilde{\Gamma}_s \subseteq \Gamma_c$ and $\tilde{\Gamma}_c \subset \Gamma_c$, respectively.

[3] If the input matrix is reducible, there will be multiple root nodes and representation trees.

3. ***R-task:*** R-tasks are created when it is advantageous to split and parallelize the refinement of eigenvalues forming a cluster. The R-tasks are therefore created during the execution of a C-task, after computing the new RRR of the cluster. The computation involved in an R-task is: Given an RRR, refine a subset of $\Gamma_i \subset \Gamma_c$ to relative accuracy with respect to the RRR via bisection.

## 3.2 The Work Queue

In order to execute the tasks in parallel, even in cases of different spectra, a work queue is established, in which the required computation is enqueued in form of the three types of tasks. Each of the tasks can then be processed by any of the processor's cores.

The work queue consists of three levels, one for each task type and implemented as a FIFO queue, with different priorities. R-tasks have the highest priority, followed by S-tasks. The lowest priority is associated to C-tasks. During the computation of the eigenvectors, each thread in the thread pool is dequeuing tasks from the work queue, processing tasks with higher priority first.

The computation is initialized by treating the root node as a special C-task. In this case there is no need to compute an RRR and refine its eigenvalues. Since the root representation is not overwritten, no special care has to be taken to resolve the data dependency of the newly created tasks at $depth = 1$ in the representation tree. Therefore the tasks are created fast and fill up the work queue. To achieve the same for clusters at higher depth, the parent RRR is copied into the output eigenvector matrix $Z$ for cluster tasks. The task is therefore created faster than computing the RRR for the cluster first and store it in $Z$, as it is done in `DSTEMR`.

The organization of the work queue is among other things motivated to bound the amount of memory required during the computation. In the case of a single thread, the order of computation complies with the `DSTEMR` routine: at each level all the singletons are processed before the clusters.

## 3.3 An Example Matrix

Fig. 2 shows the execution traces of an exemplary eigenvector computation. The examined matrix of size $n = 12,387$ comes from a frequency response analysis of automobile bodies. Computing the eigenvectors with 16 threads took about 3.3 seconds. In the timeline graph green, blue and yellow sections correspond to the processing of S-tasks, C-tasks, and R-tasks, respectively. Everything considered as parallization overhead is colored red.

On average, each thread spends about 66% of the execution time in computing the eigenvectors of singletons, 19% in computing new RRRs of clusters and to refine the associated eigenvalues, and additionally 15% for refining eigenvalues. Almost no overhead occurs during the computation, due to dynamic task scheduling.

The first time that the refinement of eigenvalues is split via R-tasks, a cluster of size $8,871$ is encountered by the bottommost thread. Since the cluster contains a large part of the eigenvectors that are still to be computed, the refinement of its eigenvalues is split among all the threads. The number of eigenvalues to refine within a task is reduced in size when the tasks are created, so that load balancing among all the threads is achieved. The procedure of splitting the refinement among all threads is repeated two more times during the computation. Later during the computation there are also examples where the refinement of eigenvalues is split, but the computation is not distributed among all threads.

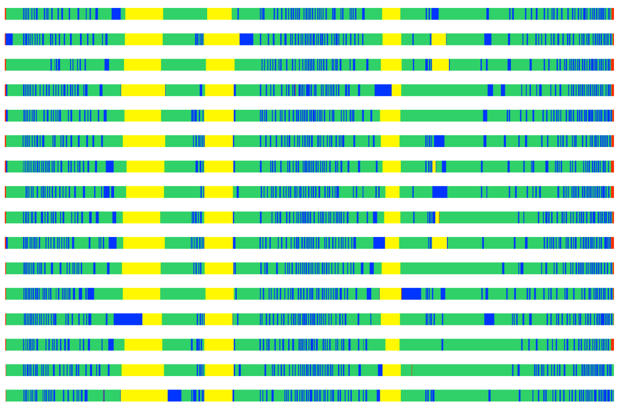

**Fig. 2.** Execution traces for a matrix of size $n = 12,387$, arising in a finite-element model of an automobile body. The colors green, blue, and yellow represent time spent in the execution of S-tasks, C-tasks, and R-tasks, respectively.

## 4 Experimental Results

In this section we present timing results for MR$^3$–SMP. All the tests were run on a SMP system comprising 4 *Intel Xeon 7460 Dunnington* multi-core processors. Each processor has 6 cores and a clock rate of 2.66 GHz. The Intel compilers[4] *icc* and *ifort*, with optimization level 3 enabled, were used for LAPACK[5] and MR$^3$–SMP. LAPACK's `DSTEMR` was linked to MKL's BLAS. For the all MKL routines Version 10.2 was used. MR$^3$–SMP was linked to the reference BLAS implementation.

In Table 1 we show timing results for a set of application matrices. In order make a fair comparison to DC, we computed all the eigenpairs. However, MR$^3$–SMP allows the computation of eigenpair subsets at reduced cost.

[4] Version 11.1.

[5] Version 3.2.1.

While MKL's and LAPACK's MRRR are sequential, MKL's DC and MR$^3$–SMP can make use of parallelism and the results for 1, 12, and 24 threads are shown. As can be seen in Fig. 1, the timings for MKL's and LAPACK's `DSTEMR` are almost identical and therefore only the results of the latter are shown.

**Table 1.** Execution times in seconds for a set of matrices arising in applications. The first four matrices are from quantum chemistry and the last four arise in finite element models.

| **Matrix** | **Size** | **DC** | | | **MRRR** | **MR$^3$–SMP** | | |
|---|---|---|---|---|---|---|---|---|
| | | 1 | 12 | 24 | seq. | 1 | 12 | 24 |
| SiOSi$_6$ | 1,687 | 0.95 | 0.40 | 0.40 | 0.52 | 0.55 | 0.14 | 0.12 |
| ZSM-5 | 2,053 | 1.47 | 0.58 | 0.56 | 0.94 | 0.97 | 0.23 | 0.15 |
| Juel.k1b | 4,237 | 11.57 | 3.69 | 3.60 | 4.49 | 4.72 | 0.97 | 0.51 |
| Auto.a | 7,923 | 63.50 | 19.32 | 17.72 | 19.51 | 20.53 | 3.69 | 1.92 |
| Auto.b | 12,387 | 219.56 | 65.70 | 33.61 | 56.43 | 59.10 | 11.63 | 4.84 |
| Auto.c | 13,786 | 233.94 | 70.94 | 36.32 | 54.27 | 60.31 | 10.63 | 5.46 |
| Auto.d | 16,023 | 324.47 | 98.83 | 92.02 | 90.27 | 97.51 | 20.56 | 8.31 |

With the exception of the last matrix, DC executed with 24 threads is faster than the sequential MRRR. In fact, for the last matrix DC was faster than MKL's `DSTEMR`, which took about 94 seconds. As shown in Table 2, in all tests MR$^3$–SMP is both faster and obtains better speedup than `DSTEDC`.

**Table 2.** Speedup of the total execution time of routine `DSTEDC` and MR$^3$–SMP. The reference for `DSTEDC` is its single threaded execution and for MR$^3$–SMP is the sequential `DSTEMR`. The last column shows the factor $\tau$ by which MR$^3$–SMP is faster than DC.

| **Matrix** | **Size** | **DC** (MKL) | **MR$^3$–SMP** | $\tau$ |
|---|---|---|---|---|
| SiOSi$_6$ | 1,687 | 2.4 | 4.3 | 3.3 |
| ZSM-5 | 2,053 | 2.6 | 6.3 | 3.7 |
| Juel.k1b | 4,237 | 3.2 | 8.8 | 7.0 |
| Auto.a | 7,923 | 3.6 | 10.2 | 9.2 |
| Auto.b | 12,387 | 6.5 | 11.7 | 6.9 |
| Auto.c | 13,786 | 6.4 | 9.9 | 6.6 |
| Auto.d | 16,023 | 3.5 | 10.9 | 11.0 |

Although the computation was executed on 24 cores, the speedup of MR$^3$–SMP against the sequential `DSTEMR` is far from 24. Why the obtained speedup is nonetheless close to the optimal is discussed through the example of matrix *Auto.b*. As it can be seen in Fig. 3 *(left)*, the fast but sequential dqds-algorithm is used to compute the initial eigenvalue approximations: it requires about 7.3

seconds, and it is used with up to 12 cores. If the dqds-algorithm were always be used, independently of the number of available cores, according to Amdahl's law the total speedup would be limited to $56.4/7.3 \approx 7.7$. This limit can be observed in Fig. 3 *(right)* for the total speedup. Instead, in MR$^3$–SMP bisection is used for more than 12 cores. The computation time for the eigenvalues decreases, but the input of the eigenvector computation changes. When in the initial eigenvalue approximation we force to always use either the dqds-algorithm or bisection, Fig. 3 *(right)* shows good scalability of the eigenvector computation. Notice that the graph of the total speedup in Fig. 3 *(right)* is not yet at a flat asymptote, and greater speedups can be expected with more parallelism.

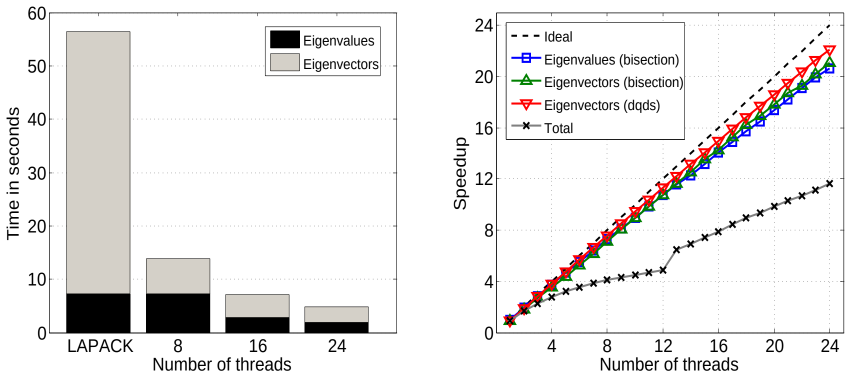

**Fig. 3.** *Left:* Time spent in the computation of the eigenvalues and eigenvectors for the matrix *Auto.b* of size 12,387. *Right:* Speedup for the eigenvalue and eigenvector computation. The total speedup is naturally limited since the sequential dqds-algorithm is used in the initial eigenvalue approximation with up to 16 cores.

For the sake of brevity, accuracy results are omitted, but we remark that in all tests the accuracy of MR$^3$–SMP is comparable to that of LAPACK's sequential routine `DSTEMR`.

## 5 Conclusion

We presented a design to adapt the algorithm of Multiple Relatively Robust Representations to shared-memory computer systems. The result, MR$^3$–SMP, is an algorithm specifically tailored for current multi-core and future many-core architectures, as well as SMP systems made out of them. We compared MR$^3$–SMP with all tridiagonal eigensolver contained in LAPACK and Intel's MKL on a set of matrices arising in real applications: in all cases MR$^3$–SMP resulted the fastest algorithm and attained the best speedups.

## Acknowledgment

The authors wish to thank the Center for Computing and Communication RWTH Aachen for the resources to run all the experiments, Diego Fabregat and Roman Iakymchuck for discussions. Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111 is gratefully acknowledged.

## References

1. Wilkinson, J.: The Calculation of the Eigenvectors of Codiagonal Matrices. Comp. J. 1 (2), 90–96 (1958)
2. Francis, J.: The QR Transform - A Unitary Analogue to the LR Transformation, Part I and II. The Comp. J. 4, (1961/1962)
3. Kublanovskaya, V.: On some Algorithms for the Solution of the Complete Eigenvalue Problem. Zh. Vych. Mat. 1, 555–572 (1961)
4. Cuppen, J.: A Divide and Conquer Method for the Symmetric Tridiagonal Eigenproblem. Numer. Math. 36, 177–195 (1981)
5. Gu, M., Eisenstat, S. C.: A Divide-and-Conquer Algorithm for the Symmetric Tridiagonal Eigenproblem SIAM J. Matrix Anal. Appl. 16 (1), 172–191 (1995)
6. Dhillon, I., Parlett, B.: Multiple Representations to Compute Orthogonal Eigenvectors of Symmetric Tridiagonal Matrices. Linear Algebra Appl. 387, 1–28 (2004)
7. Anderson, E, Bai, Z., Bishop, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., Sorensen, D.: LAPACK Users' Guide. SIAM, 2nd Edition (1995)
8. Demmel, J., Marques, O., Parlett, B., Vömel, C.: Performance and Accuracy of LAPACK's Symmetric Tridiagonal Eigensolvers. SIAM J. Sci. Comp. 30, 1508–1526 (2008)
9. Dongarra, J., Du Cruz, J., Duff, I., Hammarling, S.: A Set of Level 3 Basic Linear Algebra Subprograms. ACM Trans. Math. Software 16, 1–17, 1990.
10. Demmel, J., Dhillon, I., Ren, H.: On the Correctness of some Bisection-like Parallel Eigenvalue Algorithms in Floating Point Arithmetic. Elec. Trans. Num. Anal. 3, 116–149 (1995)
11. Parlett, B., Dhillon, I.: Relatively Robust Representations of Symmetric Tridiagonals. Linear Algebra Appl. 309, 121–151 (1999)
12. Dhillon, I., Parlett, B.: Orthogonal Eigenvectors and Relative Gaps. SIAM J. Matrix Anal. Appl. 25, 858–899 (2004)
13. Parlett, B., Marques, O.: An Implementation of the DQDS Algorithm (Positive Case). Linear Algebra Appl. 309, 217–259 (1999)
14. Parlett, B.: The Symmetric Eigenvalue Problem. Prentice-Hall (1980)
15. Bientinesi, P., Dhillon, I., van de Geijn, R.: A Parallel Eigensolver for Dense Symmetric Matrices Based on Multiple Relatively Robust Representations. SIAM J. Sci. Comp. 21, 43–66 (2005)
16. Vömel, C.: ScaLAPACK's MRRR Algorithm. ACM Trans. on Math. Software 37 (1), Article 1 (2010)
17. Dhillon, I., Parlett, B., Vömel, C.: The Design and Implementation of the MRRR Algorithm. ACM Trans. on Mathem. Software 32, 533–560 (2006)