Aachen Institute for Advanced Study in Computational Engineering Science

Preprint: AICES-2010/10-2

11/October/2010

MR3–SMP: A Symmetric Tridiagonal Eigensolver for Multi-Core Architectures

M. Petschow and P. Bientinesi

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

MR^3–SMP: A Symmetric Tridiagonal Eigensolver for Multi-Core Architectures

M. Petschow[a], P. Bientinesi[a]

[a]*RWTH Aachen, Aachen Institute for Advanced Study in Computational Engineering Science, 52062 Aachen, Germany*

Abstract

The computation of eigenvalues and eigenvectors of symmetric tridiagonal matrices arises frequently in applications; often as one of the steps in the solution of Hermitian and symmetric eigenproblems. While several accurate and efficient methods for the triadiagonal eigenproblem exist, their corresponding implementations usually target uni-processors or large distributed memory systems. Our new eigensolver `MR`^{3}`-SMP` is instead specifically designed for multi-core and many-core general purpose processors, which today have effectively replaced uni-processors. We show that in most cases `MR`^{3}`-SMP` is faster and achieves better speedups than state-of-the-art eigensolvers for uni-processors and distributed-memory systems.

Keywords: MRRR algorithm, tridiagonal eigensolver, eigenvalue, eigenvector, multi-core

1. Introduction

The role of the symmetric tridiagonal eigenproblem is twofold: The problem is both relevant in its own right for many applications, and it lies at the heart of all direct methods for computing eigenvalues and eigenvectors of dense and banded Hermitian and symmetric matrices [1]. The problem consists of finding solutions to the equation

$$Tz = \lambda z \ , \tag{1}$$

where $T \in \mathbb{R}^{n \times n}$ is a given symmetric tridiagonal matrix, and $\lambda \in \mathbb{R}$ and $z \in \mathbb{R}^n$ are the sought after *eigenvalue* and *eigenvector*, respectively. Together, eigenvalue and eigenvector form an *eigenpair*; since T is symmetric, Eqn. (1) is satisfied by n distinct eigenpairs, and the computed eigenvectors form an orthogonal basis: $z_j^T z_i = 0$ for $j \neq i$.

Email addresses: `petschow@aices.rwth-aachen.de` (M. Petschow), `pauldj@aices.rwth-aachen.de` (P. Bientinesi)

The first stable method that computes k eigenpairs in only $O(nk)$ arithmetic operations is the algorithm of *Multiple Relatively Robust Representations* (MRRR) [2, 3]. Thanks to its low complexity, MRRR is one of the fastest algorithms available for the symmetric tridiagonal eigenproblem [4]. Extant implementations target both sequential processors—LAPACK's routines `xSTEMR` [5, 6]—and distributed memory architectures—ParEig [7], ScaLAPACK [8, 9]. While for large problems a distributed-memory environment is a necessity, small and medium size problems are usually handled by a single processor, which nowadays consists of multiple computing cores, or by shared-memory systems. However, neither the `xSTEMR` routines nor the implementations for distributed systems fully exploit the parallelism provided by multi-core architectures. In fact, the formers cannot take advantage of a multi-threaded Basic Linear Algebra Subprograms (BLAS) library [10] because of the nature of the MRRR algorithm, and the latter are penalized by the redundant computations they perform to avoid costly communication.

In this document we present a new parallel eigensolver, $\texttt{MR}^3\texttt{-SMP}$, especially designed to take advantage of the features of multi-core and SMP systems. Here we detail our parallelization strategy and show evidence that a task queue-based approach leads to remarkable results in terms of scalability and execution time: Compared with the fastest solvers available on both artificial and application matrices, $\texttt{MR}^3\texttt{-SMP}$ consistently results the fastest.

The rest of the paper is organized as follows: Algorithms for the tridiagonal eigenproblem, and MRRR in particular, are discussed in Sections 2 and 3, respectively. In Section 4 we introduce $\texttt{MR}^3\texttt{-SMP}$, and timing and accuracy results are given in Section 5. In Section 6 we provide some closing remarks.

2. Related Work

A number of research efforts are devoted to adapting the functionality of LAPACK to multi-core processors. Among them, the *Parallel Linear Algebra for Scalable Multi-core Architectures* (PLASMA) project [11] and the *Formal Linear Algebra Methods Environment* (FLAME) project [12, 13]. While these efforts produced routines for the solution of linear systems, at the moment they provide no support for the symmetric tridiagonal eigenproblem. Similar projects address the use of Graphics Processing Units (GPUs) for high-performance computations; in this context a prototype of the MRRR algorithm for GPUs was recently developed [14].

2.1. The Symmetric Tridiagonal Eigenproblem as Part of Dense Eigenproblems

The computation of eigenvalues and eigenvectors of symmetric tridiagonal matrices arises in the solution of dense and banded Hermitian (symmetric) eigenproblems, as it is common to solve these problems in three stages: 1) Reduction to real symmetric tridiagonal form; the cost of this stage is about $\frac{16}{3}n^3$ arithmetic operations for Hermitian and $\frac{4}{3}n^3$ for symmetric matrices, respectively. 2) Solution of the symmetric tridiagonal eigenproblem; cost: $O(nk)$ operations.

3) Eigenvectors back-transformation; cost: about $8n^2k$ arithmetic operations for Hermtian and $2n^2k$ for symmetric matrices, respectively.

Since only about half of the operations in the reduction can be performed by the efficient level-3 BLAS, for large matrices this stage becomes the computational bottleneck. Nonetheless, despite the lower complexity, in a parallel environment the solution of the tridiagonal eigenproblem can be, for small matrices, as expensive as the reduction. This is shown in Fig. 1 (*left*) for a dense symmetric matrix of size $n = 4{,}289$. As the matrix size increases, the reduction, as dictated by its complexity, becomes much more expensive than the solution of the tridiagonal eigenproblem. However, as shown in Fig. 1 (*right*), the tridiagonal eigenproblem remains the second most expensive stage, despite the higher complexity of the back-transformation. Therefore, the objective for a scalable solver is to make the execution time for the tridiagonal eigenproblem negligible, as the lower complexity dictates.

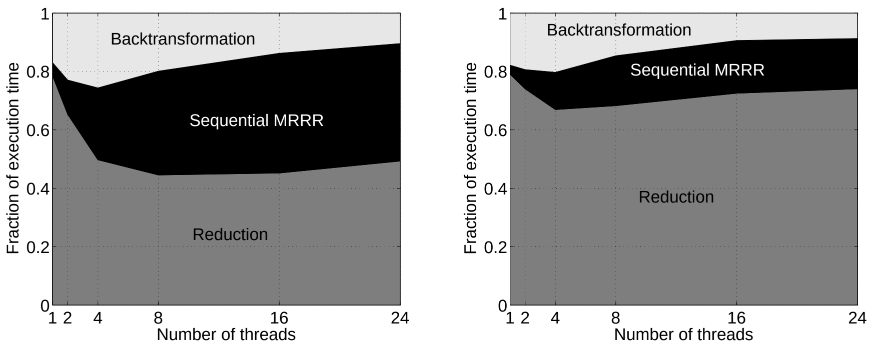

Figure 1: Solution of dense symmetric eigenproblems. Fraction of execution time spent in the three stages depending on the number of used threads. *Left:* Matrix size $n = 4{,}289$. *Right:* Matrix size $n = 7{,}923$.

2.2. Other Algorithms for the Symmetric Tridiagonal Eigenproblem

The MRRR algorithm is only one of the available methods for the solution of the symmetric tridiagonal eigenproblem. Among the most efficient and accurate ones, we mention Bisection and Inverse Iteration (BI) [15], QR [16, 17], and Divide & Conquer (DC) [18, 19].

Like the MRRR algorithm, the method of Inverse Iteration allows for the computation of a subset of eigenpairs at reduced cost, requiring $O(nk^2)$ arithmetic operations in the worst case. In contrast, the other methods can only compute all eigenpairs and may require $O(n^3)$ operations in the worst case [1]. For all these algorithms there exist both sequential and distributed memory implementations. A comprehensive discussion about existing parallel algorithms and their performance can be found in [7].

An informative and detailed performance analysis of LAPACK's implementations of the aforementioned four algorithms can be found in [4]. Here we summarize the salient results. In terms of accuracy, QR and DC are normally preferable over BI and MRRR; DC requires $O(n^2)$ additional memory and therefore much more than all the other algorithms, which only require $O(n)$ extra storage; DC and MRRR are *much faster* than QR and BI; despite the fact that MRRR uses the fewest floating point operations (flops), DC can be faster on certain classes of matrices. Whether DC or MRRR is faster depends on the spectral distribution of the input matrix.

When executed on multi-core architectures, which algorithm is faster additionally depends on the amount of available parallelism; indeed, if many cores are available, DC using multi-threaded BLAS might become faster than MRRR. In Fig. 2 we report representative timings for the computation of all eigenpairs as a function of the number of threads used;[1] the input matrix of size $n = 12{,}387$ comes from a finite-element model of an automobile body. Shown are LAPACK's DC and MRRR implementations, as well as the new `MR`^{3}`-SMP`. BI with about 2 *hours* and QR with more than 6 *hours* are much slower and not shown.

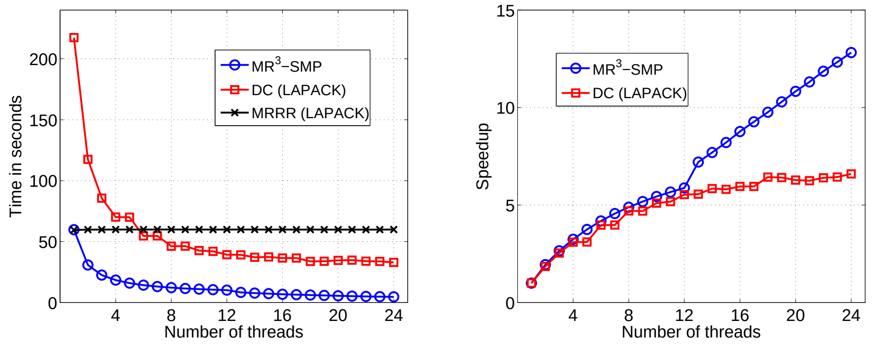

Figure 2: Execution of `MR`^{3}`-SMP` and LAPACK's DC and MRRR for a matrix of size 12,387. The algorithms are linked to a multi-threaded BLAS. *Left:* Timings. *Right:* Speedups. The slope of `MR`^{3}`-SMP`'s curve at 24 threads is still positive, indicating that more available hardware parallelism will yield even higher speedups.

While DC takes advantage of parallelism by using a multi-threaded BLAS library, the other LAPACK routines do not exploit multi-core architectures. Once enough hardware parallelism is available, DC becomes faster than the sequential MRRR algorithm. `MR`^{3}`-SMP` on the other hand is specifically designed for multi-core processors, and results to be faster and, as Fig. 2 (*right*) shows, more scalable than all the other LAPACK routines.

Readers familiar with the MRRR algorithm may directly jump to Section 4, where we provide a detailed discussion of `MR`^{3}`-SMP`.

[1]For all the timings LAPACK version 3.2.2 was used together with Intel's MKL BLAS version 10.2. More details about the parameters of the experiments are given in Section 5.

3. The MRRR Algorithm and its Representation Tree

The MRRR algorithm is an elaborated version of Inverse Iteration that eliminates the need for the Gram-Schmidt process to obtain numerically orthogonal eigenvectors. As a consequence, k eigenpairs are accurately computed in $O(nk)$ operations. Theoretical aspects of the algorithm and bounds for its accuracy can be found in [2, 3] and references therein. In this section we concentrate on the aspects of the algorithm that are important for devising a parallel computation strategy. In particular, we focus on the so called *representation tree* [2], and the different computational tasks that are encountered in the algorithm. An exemplary representation tree is illustrated in Fig. 3.

Without loss of generality, in the remainder of the paper we assume the tridiagonal input matrix T to be *irreducible*, that is, no off-diagonal element is smaller in magnitude than a certain threshold that warrants setting it to zero. If T is not irreducible, the problem can be broken up into a set of smaller irreducible ones.

The algorithm commences by computing a factorization of T that defines all or a set of the eigenvalues to high relative accuracy. Such a factorization is called a *Relatively Robust Representation* (RRR) [20]. Candidates for RRRs are bidiagonal factorizations of the form $LDL^T = T - \sigma I$, where $\sigma \in \mathbb{R}$ is called a *shift*, and the matrices L and D are unit lower bidiagonal and diagonal, respectively. An RRR is therefore specified by the $2n-1$ non-trivial entries of D and L. In the example in Fig. 3, the root node corresponds to an initial RRR $L_0 D_0 L_0^T = T - \sigma_0 I$. This representation defines the desired eigenvalues λ_i, with $i \in \Gamma = \{1, 2, \ldots, 9\}$, to high relative accuracy; this RRR is often referred as the *root representation.*

Given an RRR, it is possible to obtain approximations for the eigenvalues $\hat{\lambda}_i$ such that $|\hat{\lambda}_i - \lambda_i| = O(\varepsilon |\hat{\lambda}_i|)$, where ε denotes the machine precision. This can be achieved by bisection in $O(n)$ arithmetic operations per eigenvalue [21]. If the RRR is definite, all eigenvalues can instead be computed by the fast dqds algorithm, also in $O(n^2)$ operations [22]. Once an approximation of the eigenvalues is available, the MRRR algorithm proceeds by computing the corresponding eigenvectors. For each eigenvalue $\hat{\lambda}_i$ two cases must be distinguished, depending on whether it is *well separated* or *clustered* with respect to its neighbors.

1) $\hat{\lambda}_i$ is well separated, meaning that its relative gap *relgap* is greater or equal than a given parameter *tol*, where

$$\text{relgap}(\hat{\lambda}_i) := \min_{j \neq i} \frac{|\hat{\lambda}_i - \hat{\lambda}_j|}{|\hat{\lambda}_i|} = \frac{\text{gap}(\hat{\lambda}_i)}{|\hat{\lambda}_i|} \quad . \tag{2}$$

In this case, $\hat{\lambda}_i$ is also called a *singleton.*

2) $\hat{\lambda}_i$ is part of a cluster, that is, $\text{relgap}(\hat{\lambda}_i) < tol$. This condition leads to clusters of size two or more. The representation tree in Fig. 3 shows examples of both singletons and clusters. The children of the root node with indices 1 and 5 are well separated, while the eigenvalues with indices 2, 3 and 4, and with indices 6 to 9 form two clusters, of size three and four, respectively.

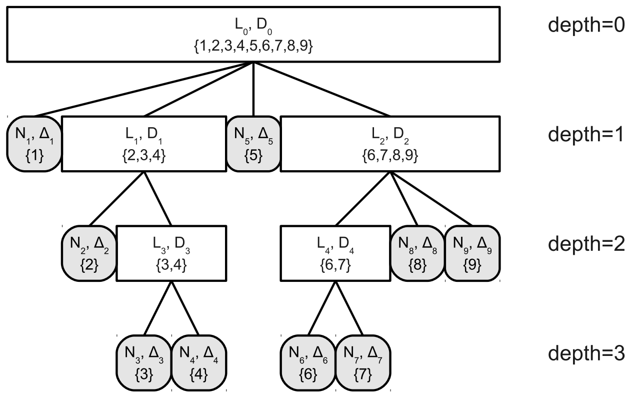

Figure 3: The unfolding of the MRRR algorithm can be pictured as a tree of relatively robust representations. The leaf nodes correspond to well-separated eigenvalues, while internal nodes represent clustered ones.

The distinction between separated and clustered eigenvalues corresponds to different execution paths. While for singletons the associated eigenvectors can be computed directly, due to numerical sensitivity, clusters require extra work before the eigenvector computation can be initiated.

For each cluster, the MRRR algorithm computes a new RRR $L_c D_c L_c^T = LDL^T - \sigma_c I$, and refines the eigenvalues in the cluster with respect to this new representation. Since the relative gap is not shift invariant, the shift σ_c can be chosen to guarantee that at least one refined eigenvalue becomes well separated. The computation of the new RRR is performed by the differential form of the stationary qd transform (dstqds) [23], a fast but inherently sequential $O(n)$ algorithm. The refinement of the eigenvalues in the cluster, performed by bisection, instead requires $O(n)$ flops per eigenvalue. For clustered eigenvalues the process is repeated until all the clusters are decomposed into singletons.

In our example in Fig. 3, the eigenvalues $\hat{\lambda}_2$, $\hat{\lambda}_3$, and $\hat{\lambda}_4$ of the root representation are clustered, as indicated by the box $\{2, 3, 4\}$ at depth one of the representation tree. As a consequence, a new RRR $L_1 D_1 L_1^T$ is found and the three eigenvalues are refined; in this new RRR, $\hat{\lambda}_2$ becomes now a singleton as the leaf $\{2\}$ at depth two displays, while $\hat{\lambda}_3$ and $\hat{\lambda}_4$ are still clustered. This process is repeated and a new RRR $L_3 D_3 L_3^T$ is computed, in which $\hat{\lambda}_3$ and $\hat{\lambda}_4$ now become singletons.

For each singleton, its corresponding eigenvector $\hat{z}_i$ is directly computed from the RRR by solving $(LDL^T - \hat{\lambda}_i I)\hat{z}_i = \gamma_r e_r$, where the right hand side of the system is the rth vector of the standard basis, scaled by γ_r. The solution is obtained through a twisted factorization $N \Delta N^T = LDL^T - \hat{\lambda}_i I$, in $O(n)$ flops. Details on the procedure can be found in [3].

This high level description of MRRR is summarized in Algorithm 1. The computation actually performed heavily depends on the spectral distribution of the matrix, the tolerance chosen to classify singletons, and on how the shift for computing a new RRR is chosen.

Algorithm 1 The MRRR Algorithm

```
Input: A symm. trid. matrix $T$; an index set $\Gamma$ of desired eigenpairs.
Output: The eigenpairs $(\hat{\lambda}_i, \hat{z}_i)$ with $i \in \Gamma$.

1: Form a work queue $Q$.
2: Compute $RRR_0$, an RRR for all $\lambda_i$ with $i \in \Gamma$.
3: Compute eigenvalues $\hat{\lambda}_i$, with $i \in \Gamma$.
4: Partition $\Gamma$ into subsets $\Gamma = \bigcup_k \Gamma_k$ according to their relative gap, and
   enqueue each $(\Gamma_k,\ RRR_0)$.
5: while $Q$ not empty do
6:     Dequeue a new task $(\tilde{\Gamma},\ RRR)$.
7:     if $|\tilde{\Gamma}| > 1$ then
8:         1. Compute $RRR_{new}$, an RRR for all $\hat{\lambda}_i$ with index $i \in \tilde{\Gamma}$.
9:         2. Refine $\hat{\lambda}_i$ with index $i \in \Gamma$ with respect to $RRR_{new}$.
10:        3. Partition $\tilde{\Gamma}$ into subsets $\tilde{\Gamma} = \bigcup_k \tilde{\Gamma}_k$ according to their relative
           gap and enqueue each $(\tilde{\Gamma}_i,\ RRR_{new})$.
11:    else
12:        Compute $\hat{\lambda}_i$ with $i \in \tilde{\Gamma}$ to high relative accuracy and its associated
           eigenvector $\hat{z}_i$.
13:    end if
14: end while
```

4. The MRRR Algorithm for Multi-Core Processors

In this section present the design of `MR`^{3}`-SMP`, a parallel version of the MRRR algorithm specifically tailored to multi-core processors and shared-memory architectures. Our C implementation is based on LAPACK's `DSTEMR` version 3.2 [6], and makes use of the *POSIX threads*. As several refinements to the MRRR algorithm have been proposed to improve its accuracy [24], we have designed `MR`^{3}`-SMP` in a modular fashion, so that algorithmic changes can be incorporated with minimal efforts.

4.1. Parallelization Strategy

The distributed-memory versions of MRRR aim at minimizing communication among processors while attaining an optimal balance of memory requirement [7, 9]. This approach comes at the expense of work load balancing, as the division of work is performed statically. Since on multi-core and shared-memory architectures the memory balance is not a concern, our objective is instead to identify the right computational granularity to balance the work load perfectly. In this section we illustrate how this is accomplished by dividing and scheduling the work dynamically.

Algorithm 1 is conceptually split into two parts, corresponding to Statements 2–4 and 5–14, respectively.

1. Computation of a root representation and initial approximation of eigenvalues;
2. Computation of eigenvectors, and final refinement of eigenvalues.

In the first part, the computation of a root RRR (as for each RRR subsequently) only costs $O(n)$ and is performed sequentially. The initial approximation of eigenvalues instead costs $O(n)$ per required eigenvalue, and can be performed in parallel using the bisection algorithm. Nonetheless, depending on the available parallelism, the sequential dqds algorithm is faster than bisection and should be preferred [7]. Assuming bisection is used to compute the eigenvalues to half machine precision and the dqds algorithm converges in about three iterations per eigenvalue, the dqds algorithm is preferable over bisection if the number of processors is smaller than or equal to $12 \cdot |\Gamma|/n$.

In the second part of Algorithm 1, our parallelization strategy consists of dividing the computation into tasks, that can be executed by multiple threads in parallel. As many different strategies in dividing and scheduling the computation can be employed, we will discuss possible variants in the following sections.

Although the eigenvalues are refined in the second part of the algorithm, for simplicity in the following we refer to *the eigenvalue computation* and *the eigenvector computation* as the first and the second part, respectively.

4.2. Dividing the Computation into Tasks

Since the representation tree characterizes the unfolding of the algorithm, it is natural to associate each node in the tree with a unit of computation. Because of this one-to-one correspondence, from now on we use the notion of task and node interchangeably. Corresponding to interior nodes and leaf nodes, we introduce two types of tasks, *C-tasks* and *S-tasks*. C-tasks deal with the processing of clusters and create other tasks, while S-tasks are responsible for the final eigenpair computation. C-tasks and S-tasks embody the two branches of the `if` statement (Line 7) in Algorithm 1: statements 8–10 and statement 12, respectively.

Irrespective of scheduling strategies, the granularity of this first approach prevents a parallel execution of tasks: In the presence of a large cluster, the threads might run out of executable tasks well before the large cluster is processed and broken up into smaller children tasks. As an example, assume that the eigenvalues λ_2 to λ_9 in Fig. 3 are all clustered together, and that we want to solve the problem on a 4-core processor; in this case, one of the four cores will tackle the singleton $\{1\}$, a second core will process the cluster, and a third and forth core will sit idle until the cluster is decomposed. Since the cluster computation is more expensive than the processing of a singleton, even the first core will have idle time, waiting for new available tasks.

To reduce the granularity of the computation, we introduce a third type of tasks, the *R-task*, that allows the decomposition of the eigenvalue refinement—the most expensive step in the C-tasks—originating immediately executable tasks. An R-task takes as input an RRR and a list of eigenvalues. As output, it returns the refined eigenvalues.

The computation associated with a C-task is summarized in Algorithm 2, where $\#left$ and $\#threads$ denote the number of eigenpairs not yet computed and the number of threads used, respectively. The algorithm directly invokes LAPACK's MRRR routines `DLARRF` and `DLARRB`; we point out that if these are amended, our parallelization of MRRR still stands, without modifications.

Algorithm 2 Process C-task

```
Input: An RRR, the eigenvalues $\hat{\lambda}_i$, and the index set $\Gamma_c$ of a cluster.
Output: S-tasks and C-tasks associated with the children of the cluster $\Gamma_c$.
1: Call subroutine ComputeNewRRRforCluster.
2: Call subroutine RefineEigenvaluesOfCluster.
3: Call subroutine SplitIntoClustersAndSingletons

SUBROUTINE: ComputeNewRRRforCluster
Input: An RRR, the eigenvalues $\hat{\lambda}_i$, and the index set $\Gamma_c$ of a cluster.
Output: $RRR_{new}$
1: Compute $RRR_{new}$, an RRR for $\lambda_i$ with $i \in \Gamma_c$ using DLARRF.

SUBROUTINE: RefineEigenvaluesOfCluster
Input: $RRR_{new}$, the eigenvalues $\hat{\lambda}_i$, and the index set $\Gamma_c$ of a cluster.
Output: Refined eigenvalues $\hat{\lambda}_i$ with $i \in \Gamma_c$.
1: if $|\Gamma_c| > \lceil \#left/\#threads \rceil$ then
2:     Decompose the refinement into R-tasks.
3: else
4:     Refine eigenvalues $\hat{\lambda}_i$ with $i \in \Gamma_c$ using DLARRB.
5: end if

SUBROUTINE: SplitIntoClustersAndSingletons
Input: $RRR_{new}$, , the eigenvalues $\hat{\lambda}_i$, and the index set $\Gamma_c$ of a cluster.
Output: S-tasks and C-tasks.
1: Partition $\Gamma_c$ into subsets $\Gamma_c = \bigcup_j \Gamma_j$ according to their relative gap.
2: Enqueue each ($\Gamma_j$, $RRR_{new}$).
```

As a guiding principle for load balance, we assure that every time a task is created, its size does not exceed $s_{max} = \lceil \#left/\#threads \rceil$ eigenpairs; the rationale being that from this moment on, each thread will be responsible for the computation of not more than s_{max} eigenpairs. As a consequence, the eigenvalues of a cluster bigger than this number will be refined in parallel by multiple threads.[2]

Vice versa, to avoid too fine a granularity, we extend S-tasks to compute the eigenpairs for multiple singletons from the same RRR; we call this strategy "bundling". Similarly to the R-tasks, we choose the maximum size of a bundle to be proportional to s_{max}. Bundling brings three advantages: better usage of cached data through temporal and spatial locality; fewer tasks and less contention for the work queues; the possibility of lowering the overall algorithm memory requirement, as explained in Subsection 4.3.1. Algorithm 3 summarizes the computation performed by an S-task, where k_1 and k_2 are appropriate constants, and RQC abbreviates Rayleigh Quotient Correction. This algorithm invokes LAPACK's MRRR auxiliary routine `DLAR1V`; the same comment as for Algorithm 2 applies here.

Algorithm 3 Process S-task

Input: An RRR (L, D), the eigenvalues, and the index set Γ_s of singletons.
Output: The eigenpairs $(\hat{\lambda}_i, \hat{z}_i)$ with $i \in \Gamma_s$.

```
1:  for each index $i \in \Gamma_s$ do
2:      while eigenpair $(\hat{\lambda}_i, \hat{z}_i)$ is not accepted do
3:          Invoke DLAR1V to solve $(LDL^T - \hat{\lambda}_i I)\hat{z}_i = \gamma_r e_r$.
4:          Record the residual norm $|\gamma_r|/\|\hat{z}_i\|_2$ and the RQC $|\gamma_r|/\|\hat{z}_i\|_2^2$.
5:          if residual norm $< k_1 n\varepsilon\,\mathrm{gap}(\hat{\lambda}_i)$ or RQC $< k_2\varepsilon|\hat{\lambda}_i|$ then
6:              Accept the eigenpair.
7:          end if
8:          if RQC does not improve $\hat{\lambda}_i$ then
9:              Use bisection to refine $\hat{\lambda}_i$ to high relative accuracy.
10:         end if
11:     end while
12:     Normalize eigenvector $\hat{z}_i \leftarrow \hat{z}_i/\|\hat{z}_i\|_2$.
13: end for
14: Decrement $\#left$ by $|\Gamma_s|$.
```

4.3. The Work Queues and Task Scheduling

Recently it has been shown that the computation of many dense linear algebra algorithms can be mapped efficiently to multi-core architectures through a task queue-based approach [13, 25]. Commonly, the number and type of tasks comprising the computation are known prior to the execution, independently of the particular input data. This makes it possible to schedule the tasks—either statically or dynamically—taking the dependencies into account. Conversely, the unfolding of the MRRR algorithm, i.e., the shape of the representation tree, heavily depends on the input matrix, and cannot be known prior to the execution; moreover, tasks are generated dynamically when non-leaf nodes are

[2]In practice, a minimum threshold prevents the splitting of small clusters.

processed. In this subsection we analyze data dependencies among tasks, and discuss how they may be scheduled for execution.

To differentiate the task priority, we form three work queues for *high*, *medium*, and *low* priority tasks, respectively. Each active thread polls the queues, from the highest to the lowest in priority, until an available task is found; the task then is dequeued and executed. The process is repeated until all eigenpairs are computed.

4.3.1. Task Data Dependencies

The output of Algorithm 1 forms a vector of k eigenvalues and a matrix of eigenvectors $Z \in \mathbb{R}^{n \times k}$. These data structures are shared among all the tasks involved in the computation. Since the index sets Γ_j—associated to C-tasks and S-tasks that can be executed in parallel—are disjoint, no data dependencies are introduced.

Dependencies between tasks only result from the management of RRRs. Each task in the work queues requires its parent RRR, which serves as input in Algorithms 2 and 3; therefore an RRR must be available until all its children tasks are processed. One possible solution is to dynamically allocate memory for each new RRR, pass the memory address to the children tasks, and keep the RRR in memory until all the children are executed. An example is given by node $\{6, 7, 8, 9\}$ in Fig. 3. The RRR (L_2, D_2) must be available as long as the children nodes $\{6, 7\}$, $\{8\}$, and $\{9\}$ have not been processed. This solution offers the following advantages: When multiple threads are executing the children of a C-task, only one copy of the parent RRR resides in memory and is shared among all threads; similarly, the required auxiliary quantities $dl(i) = D(i, i) \cdot L(i+1, i)$ and $dll(i) = D(i, i) \cdot L(i + 1, 1)^2$, for all $i = 1, \ldots, n-1$, are evaluated once and are shared; in architectures with shared caches, additional benefits might arise from re-use of cached data.

On the downside, this approach needs an extra $O(n^2)$ work space in the worst case. In order to reduce the memory requirement, we make the following observation. Each task can associate the portion of Z corresponding to Γ_c in Algorithm 2 or Γ_s in Algorithm 3 with local workspace, i.e., a section of memory not accessed by any other task in the work queues. Continuing with the example of node $\{6, 7, 8, 9\}$ at depth one of the tree in Fig. 3, the columns 6 to 9 are regarded as local storage of the corresponding task.

If the RRR required as input is stored in the task's local workspace, then we call such a task *independent*. If all the children of a task are independent, then the task's RRR is not needed anymore and can be discarded. Additionally, all the children tasks can be executed in any order, without any data dependency. We now illustrate that all the C-tasks can be made independent and that practically all S-tasks can be made independent too.

C-tasks: As a cluster consists of at least two eigenvalues, the corresponding portion of Z—used as local workspace—is always large enough to contain an RRR, identified by the diagonal entries of D and subdiagonal entries of L. We therefore use Z to store the parent RRR.[3] Unfortunately rendering the tasks independent comes with a small overhead due to: storing the parent RRR into Z; retrieving the parent RRR from Z; recomputing $dl(i)$ and $dll(i)$. Additionally, negative effects on performance might be caused by lack of re-used cached data.

S-tasks: The same approach is feasible for S-tasks whenever at least two singletons are bundled. Conversely, the approach cannot be applied in the extreme scenario in which a cluster is decomposed into smaller clusters plus only one singleton. The leaf node $\{2\}$ in Fig. 3 represents such an exception. All the other S-tasks may be bundled in groups of two or more, and therefore can be made independent. As before, this approach introduces some overheads: storing the parent RRR into Z; duplicating the RRR to make Z available for the eigenvectors; recomputing $dl(i)$ and $dll(i)$. A drawback of this approach is that when several S-tasks children of the same node are processed simultaneously, multiple copies of the same RRR reside in memory, preventing the re-use of cached data. In the example of node $\{6, 7, 8, 9\}$ in Fig. 3, storing the RRR (L_2, D_2) into columns 6 and 7, as well as columns 8 and 9, creates an independent C-task for the node $\{6, 7\}$ and an independent S-task $\{8, 9\}$, where the two singletons $\{8\}$ and $\{9\}$ are bundled.

We point out again that while working with independent tasks introduces some overhead, it brings great flexibility in the scheduling, as tasks can now be scheduled *in any order*.

4.3.2. Task Scheduling

Many strategies can be employed for the dynamic scheduling of tasks. As a general guideline, in a shared memory environment, having enough tasks to feed all the computational cores is paramount to achieve high-performance. In this section we discuss the implementation of two simple but effective strategies that balance task-granularity, memory requirement, and creation of tasks. In both cases we implemented three separate FIFO queues, to avoid contention when many processors are used. Also, both approaches schedule R-tasks with high priority, because they arise as part of the decomposition of large clusters, and originate work in the form of tasks.

C-tasks before S-tasks: All enqueued C-tasks and S-tasks are made independent. (In the rare event that a S-task cannot be made independent, the task is executed immediately without being enqueued.) As a consequence, all C-tasks and S-tasks in the work queues can be scheduled in any order. Since processing C-tasks creates new tasks that can be executed in parallel, we impose that C-tasks (medium priority) are dequeued before S-tasks (low priority). This ordering is a special case of many different strategies in which the execution of C-tasks and S-tasks are interleaved. But no other ordering will offer more executable tasks in the work queues, and we expect that it therefore will, for the scenario that all tasks are independent, attain the best performance.

[3]An alternative, used by `DSTEMR`, is to compute and store new RRR of the cluster.

S-tasks before C-tasks: No S-task is made independent. In order to obtain a similar memory requirement as in the first approach, we are forced to schedule the S-tasks before the C-tasks. The reason is that for each cluster an RRR must be kept in memory until all its children S-tasks are executed. The ordering assures that at any time S-tasks of no more than *#threads* clusters are in the queue. While the flexibility in task scheduling is reduced, all the overhead from making the S-tasks independent is avoided. In practice this second approach is slightly faster—about 5–8%—and is therefore used for all timings in Section 5.

4.4. Memory Requirement

`MR`^{3}`-SMP` and LAPACK's `DSTEMR` make efficient use of memory by using the eigenvector matrix Z as temporary storage. Additionally `DSTEMR` overwrites the input matrix T to store only the RRR of the currently processed node. This approach is not feasible in a parallel setting, as the simultaneous processing of multiple nodes requires storing the associated RRRs separately. Moreover, in our multi-threaded implementation, each thread requires its own work space. As a consequence, the parallelization and its resulting performance gain comes with the cost of a memory requirement slightly higher than in the sequential case.

While `DSTEMR` requires extra work space to hold $18n$ double precision numbers and $10n$ integers [4, 6], `MR`^{3}`-SMP` requires extra storage of about $(12n + 6n \cdot \#threads)$ double precision numbers and $(10n + 5n \cdot \#threads)$ integers. For a single thread the total work space requirement is therefore roughly the same as for `DSTEMR`, and the requirement increases linearly with the number of threads used for the computation. We remark that, thanks to shared data, the required memory is less than *#threads* multiplied by the memory needed for the sequential routine.

In Fig. 4 we show for two different kind of matrices, namely so called Hermite and symmetrized Clement matrices [26], the measured peak memory usage by `MR`^{3}`-SMP` compared to the estimate given above. The graph on the left shows, by setting *#threads* to 12, that the extra memory requirement is linear in the matrix size n. In the graph on the right the matrix size has a fixed value $n = 5{,}000$ and the dependence of the required memory with the number of threads is shown. Although the memory requirement increases with the number of threads, even if 24 threads are used, the required extra storage only makes up for less than 5% of the memory required by the output matrix Z.

Since the DC algorithm requires $O(n^2)$ double precision work space, the advantage of MRRR in requiring much less memory than DC is therefore maintained.

5. Experimental Results

We now turn the attention on timing and accuracy results of `MR`^{3}`-SMP`. The section is organized in two parts: In the first part we report on experiments with a set of matrices of known eigenspectra. The focus is on the scalability

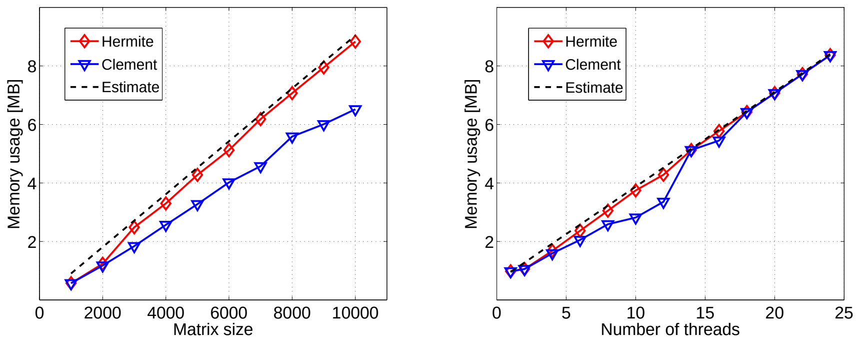

Figure 4: *Left:* Peak memory requirement of `MR`^{3}`-SMP` as a function of the matrix size using a fixed number of 12 threads. *Right:* Peak memory requirement for a fixed matrix size of $n = 5{,}000$ as a function of the number of threads used. The required memory has to be compared to the output eigenvector matrix requiring about 190 MB memory.

of the eigenvector computation, and on the scenario in which only a subset of the eigenpairs is requested. In the second part we look at matrices arising in scientific applications, and investigate the overall performance and accuracy of `MR`^{3}`-SMP`. We compare `MR`^{3}`-SMP` with ParEig—a solver for a distributed-memory systems—and all tridiagonal eigensolvers contained in LAPACK.

All tests were run on a SMP system made of four *Intel Xeon 7460 Dunnington* six-core processors, operating at a frequency of 2.66 GHz. We used LAPACK version 3.2.2 in conjunction with Intel's MKL BLAS version 10.2, and ParEig version 2.0 together with Open MPI version 1.4.2. All the routines were compiled with version 11.1 of the Intel compilers *icc* and *ifort*, with optimization level three enabled.

5.1. Artificial Matrices

5.1.1. Different Spectral Distributions

As discussed in Section 3, the representation tree of a matrix, and consequently the unfolding of the algorithm, depends on the spectral distribution of the matrix. For this reason it is instructive to inspect the behavior of `MR`^{3}`-SMP` with matrices with known eigenspectrum. In particular, we consider the following four eigenvalue distributions, defined in the interval $[\varepsilon, 1]$, and generated by *stetester* [26], a test software for symmetric tridiagonal eigensolvers:

1. *Uniform* distribution: $\lambda_i = \varepsilon + (i-1)(1-\varepsilon)/(n-1)$, $i = 1, \ldots, n$.
2. *Geometric* distribution: $\lambda_i = \varepsilon^{(n-i)/(n-1)}$, $i = 1, \ldots, n$.
3. *Random* values, *uniformly* distributed.
4. *Exponential* of random values, *uniformly* distributed.

In Fig. 5 we report on the execution time (*left*) and the speedup (*right*) limited to the eigenvector computation. For each matrix, in order to make sure that the initial eigenvalue approximation is the same, independently of the number

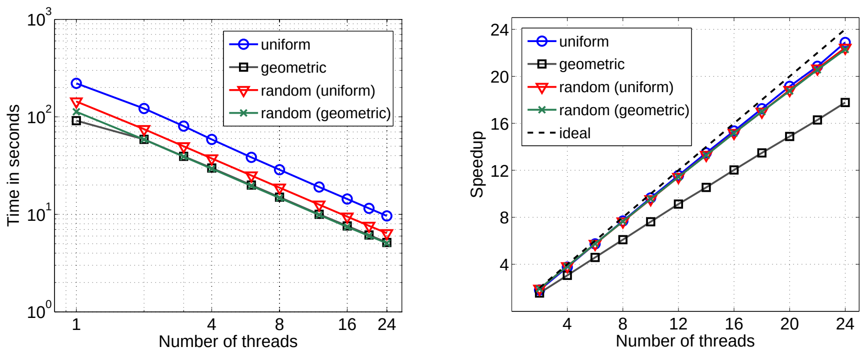

Figure 5: Eigenvector computation for matrices of size 20000 with known spectral distribution. *Left:* Execution time. *Right:* Speedup with respect to single-threaded DSTEMR.

of threads, we used the dqds algorithm. The timings for the single-threaded execution refer to LAPACK's MRRR routine DSTEMR. This is the reference time used for calculating the speedups. All matrices are of size 20,000. From the logarithmic scale it is easy to discern that the execution time decreases linearly as the number of threads used for the computation increase. The lack of curvature in the plots suggests that if more than 24 cores were available, a further decrease in time should be expected.

5.1.2. Subset Computations

An important feature of the MRRR algorithm is the ability of computing a subset of eigenpairs at reduced cost. We show that with a task-based parallelization the same is also true in the multi-threaded case. In Fig. 6 (*left*) we show MR^3-SMPs total execution time against the fraction of requested eigenspectrum. The test matrices are of size 10,001, and the computed subsets are on the left end of the spectrum. The graph on the right shows additionally the obtained speedup when compared with DSTEMR. All timings refer to the execution with 24 threads.

In all the experiments the execution time for a half/a quarter of the total eigenpairs is at most a half/a quarter of the time required to compute all eigenpairs. This indicates that MR^3-SMP is especially well suited for subsets computation. In all cases the corresponding speedup is higher for computing subsets than for finding all eigenpairs.

5.2. Application Matrices

Arguably, the most important test matrices come from applications. Here we show detailed timing and accuracy results for a set of tridiagonal matrices arising in different scientific and engineering problems. Since some of the eigensolvers in LAPACK do not support subset computation, in all the tests the entire eigenspectrum is computed.

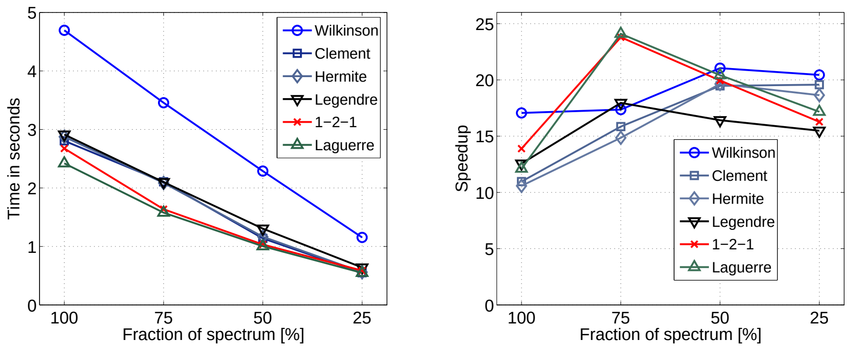

Figure 6: Computation of a subset of eigenpairs using `MR`^{3}`-SMP` with 24 threads. *Left:* Execution time. *Right:* Speedup with respect to `DSTEMR`.

5.2.1. MR^3*–SMP vs. LAPACK's eigensolvers*

In Table 1 we show timings for QR (`DSTEV`), BI (`DSTEVX`), MRRR (`DSTEMR`), as well as for DC (`DSTEDC`) and `MR`^{3}`-SMP`. The execution time for the first three routines is independent of the number of threads, which means that they cannot achieve parallelism through multi-threaded BLAS. Conversely, both DC and `MR`^{3}`-SMP` take advantage of multiple threads, through multi-threaded BLAS and a task queue-based approach, respectively. All the timings refer to the execution with 24 threads, that is, as many threads as cores available. The execution times for QR and BI are significantly higher than for the other algorithms, thus we omit the results for the largest matrices.

Matrix	**Size**	QR	BI	MRRR	DC	MR^3–SMP
ZSM-5	2,053	68.4	6.24	0.92	0.70	**0.15**
Juel-k2b	4,289	921	382	4.41	3.39	**0.52**
Auto-a	7,923	6,014	2,286	18.8	12.2	**1.88**
Auto-b	12,387	22,434	7,137	59.5	32.9	**4.65**
Auto-c	13,786	–	9,474	56.6	35.4	**5.34**
Auto-d	16,023	–	–	87.8	45.8	**7.76**

Table 1: Execution times in seconds for a set of application matrices.

In all cases `MR`^{3}`-SMP` is the fastest solver, being about one order of magnitude faster than DC.

For all matrices listed in Table 1, Fig. 7 (*left*) shows `MR`^{3}`-SMP`'s speedup in the total execution time. All the other routines do not attain any speedup. Regarding `MR`^{3}`-SMP`, every line up to 12 threads converges to a constant value, a phenomenon which can be better understood by looking at Fig. 7 (*right*) with the example of the matrix *Auto-b*. In the figure we plot the execution time for the initial eigenvalue approximations and the eigenvectors computation sepa-

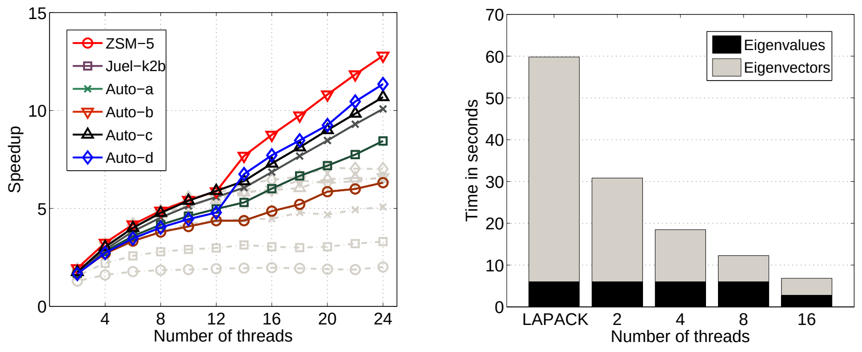

Figure 7: *Left:* Speedups for the total execution time of MR^3-SMP with respect to DSTEMR. The speedups for DSTEDC are shown in light gray. *Right:* Separate execution time for the initial eigenvalues approximation, and for the subsequent eigenvectors computation. The input matrix *Auto-b* is of size 12,387.

rately. When all the eigenpairs are requested, the sequential dqds algorithm is used to approximate the eigenvalues. Because of the sequential nature of the dqds algorithm, the maximal speedup is limited. When enough parallelism is available, bisection becomes faster than dqds; according to the criterion discussed in Subsection 4.1, we switch to bisection if more than 12 threads are used.

All the speedup curves in Fig. 7 (*left*) have positive slope at 24 threads, thus indicating that further speedups should be expected as the amount of available cores increases. In Fig. 7 (*left*) we also plot, in light gray, the speedups for DSTEDC. In all cases MR^3-SMP's speedups are higher.

5.2.2. *MR^3–SMP vs. ParEig*

In Table 2 we compare the timings of MR^3-SMP and ParEig. ParEig is designed for distributed-memory architectures, and uses the *Message-Passing-Interface* (MPI) for communication. Since ParEig is the fastest distributed-memory parallel symmetric tridiagonal eigensolver, we omit a comparison to other routines. Instead we refer to [7, 9] for comparisons of ParEig to other eigensolvers. In the experiments, we present ParEig's timings for 22 processes as they were consistently faster than the execution with 24 processes.

The results show that MR^3-SMP matches and even surpasses the performance of ParEig. For a direct comparison of MR^3-SMP with ParEig it is important to stress that, even though both routines implement the MRRR algorithm, several internal parameters are different. Most importantly, the minimum relative gap *tol*, which determines when an eigenvalue is to be considered a singleton, is set to $\min(10^{-2}, \frac{1}{n})$ and 10^{-3} in ParEig and MR^3-SMP, respectively. In order to make a fair comparison, in brackets we show the execution time of MR^3-SMP when using

Matrix	**Size**	MR^3-SMP	ParEig
ZSM-5	2,053	0.15 (0.16)	0.13
Juel-k2b	4,289	0.52 (0.49)	1.29
Auto-a	7,923	1.88 (1.62)	2.89
Auto-b	12,387	4.65 (3.93)	5.48
Auto-c	13,786	5.34 (3.02)	5.98
Auto-d	16,023	7.76 (5.69)	7.99

Table 2: Execution times in seconds for a set of matrices arising in applications. The timings in brackets are achieved if `MR`^{3}`-SMP` uses the same splitting criterion as ParEig.

the parameter *tol* as set in ParEig.[4] The numbers indicate that with similar tolerances, in the shared-memory environment `MR`^{3}`-SMP` outperforms ParEig.

5.2.3. Accuracy Results

In Fig. 8 we present accuracy results for the four fastest routines. On the left we show the normalized largest residual norm, given by

$$R = \max_i \frac{\|T\hat{z}_i - \hat{\lambda}_i \hat{z}_i\|_1}{\|T\ \|_1 n\varepsilon} \quad , \tag{3}$$

where $\varepsilon \approx 2.22 \cdot 10^{-16}$ denotes the machine precision. On the right we display the loss of orthogonality, defined as

$$O = \max_{i,j} \frac{|\hat{z}_j^T \hat{z}_i - \delta_{ij}|}{n\varepsilon} \quad , \tag{4}$$

where δ_{ij} is the Kronecker symbol.

For all the application matrices, the routines attain equally good residuals, without any clear winner. In terms of orthogonality, DC is—as expected—moderately more accurate than the three MRRR-based routines, which attain almost identical results.

6. Conclusions

We have presented `MR`^{3}`-SMP`, an eigensolver for symmetric tridiagonal matrices, specifically designed for multi-core and many-core architectures.[5] `MR`^{3}`-SMP` breaks the computation into tasks to be executed by in parallel multiple threads. The tasks are both created and scheduled dynamically. While a static division of work would introduce smaller overheads, the dynamic approach is flexible and

[4]Despite a slightly loss of performance, in `MR`^{3}`-SMP` we conservatively set $tol = 10^{-3}$ for accuracy reasons [9].

[5]The source code is available on request.

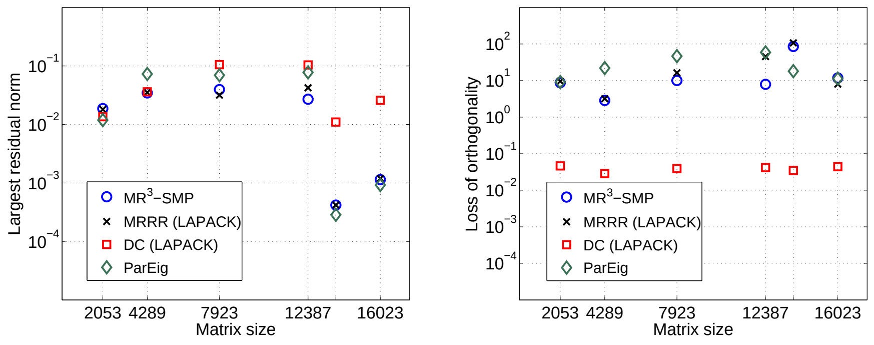

Figure 8: Accuracy of MR^3-SMP, LAPACK's MRRR and DC algorithm, and ParEig for a set of application matrices. *Left:* Largest residual norm as defined in Eqn. (3). *Right:* Loss of orthogonality as defined in Eqn. (4).

produces remarkable work load balancing. For small and medium size matrices, MR^3-SMP matches or outperforms ParEig, the fastest existing parallel eigensolver; compared with all the eigensolvers in LAPACK, MR^3-SMP is the fastest and most scalable.

Acknowledgment

The authors wish to thank the Center for Computing and Communication at RWTH Aachen for the resources, Diego Fabregat and Roman Iakymchuk for suggestions on an earlier draft of the paper. Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111 is gratefully acknowledged.

References

[1] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, H. van der Vorst, Templates for the Solution of Algebraic Eigenvalue Problems - A Practical Guide, SIAM, Philadelphia, 2000.

[2] I. Dhillon, B. Parlett, Multiple Representations to Compute Orthogonal Eigenvectors of Symmetric Tridiagonal Matrices, Linear Algebra Appl. 387 (2004) 1–28.

[3] I. Dhillon, B. Parlett, Orthogonal Eigenvectors and Relative Gaps, SIAM J. Matrix Anal. Appl. 25 (2004) 858–899.

[4] J. Demmel, O. Marques, B. Parlett, C. Voemel, Performance and Accuracy of LAPACK's Symmetric Tridiagonal Eigensolvers, SIAM J. Sci. Comp. 30 (2008) 1508–1526.

[5] E. Anderson, Z. Bai, C. Bishop, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorensen, LAPACK Users' Guide, SIAM, Philadelphia, 2nd edition, 1995.

[6] I. Dhillon, B. Parlett, C. Voemel, The Design and Implementation of the MRRR Algorithm, ACM Trans. on Mathem. Software 32 (2006) 533–560.

[7] P. Bientinesi, I. Dhillon, R. van de Geijn, A Parallel Eigensolver for Dense Symmetric Matrices Based on Multiple Relatively Robust Representations, SIAM J. Sci. Comp. 21 (2005) 43–66.

[8] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. S. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, ScaLAPACK Users' Guide, SIAM, Philadelphia, 1997.

[9] C. Voemel, ScaLAPACK's MRRR Algorithm, ACM Trans. on Math. Software 37 (2010).

[10] J. Dongarra, J. D. Cruz, I. Duff, S. Hammarling, A Set of Level 3 Basic Linear Algebra Subprograms, ACM Trans. Math. Software 16 (1990) 1–17.

[11] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov, Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects, Journal of Physics: Conference Series 180 (2009).

[12] F. G. V. Zee, P. Bientinesi, T. M. Low, R. A. van de Geijn, Scalable Parallelization of FLAME Code via the Workqueuing Model, ACM Trans. Math. Software 34 (2008) 10:1–10:29.

[13] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. V. Zee, E. Chan, Programming Matrix Algorithms-by-Blocks for Thread-Level Parallelism, ACM Trans. Math. Software 36 (2009).

[14] C. Lessig, P. Bientinesi, On Parallelizing the MRRR Algorithm for Data-Parallel Coprocessors, in: 8th Intern. Conf. on Parallel Proc. and Applied Math. (PPAM 2009).

[15] J. Wilkinson, The Calculation of the Eigenvectors of Codiagonal Matrices, Comp. J. 1 (1958) 90–96.

[16] J. Francis, The QR Transform - A Unitary Analogue to the LR Transformation, Part I and II, The Comp. J. 4 (1961/1962).

[17] V. Kublanovskaya, On some Algorithms for the Solution of the Complete Eigenvalue Problem, Zh. Vych. Mat. 1 (1961) 555–572.

[18] J. Cuppen, A Divide and Conquer Method for the Symmetric Tridiagonal Eigenproblem, Numer. Math. 36 (1981) 177–195.

[19] M. Gu, S. C. Eisenstat, A Divide-and-Conquer Algorithm for the Symmetric Tridiagonal Eigenproblem, SIAM J. Matrix Anal. Appl. 16 (1995) 172–191.

[20] B. Parlett, I. Dhillon, Relatively Robust Representations of Symmetric Tridiagonals, Linear Algebra Appl. 309 (1999) 121–151.

[21] J. Demmel, I. Dhillon, H. Ren, On the Correctness of some Bisection-like Parallel Eigenvalue Algorithms in Floating Point Arithmetic, Electronic Trans. Num. Anal. 3 (1995) 116–149.

[22] B. Parlett, O. Marques, An Implementation of the DQDS Algorithm (Positive Case), Linear Algebra Appl. 309 (1999) 217–259.

[23] H. Rutishauser, Der Quotienten-Differenzen-Algorithmus, Z. Angew. Math. Phys. 5 (1954) 223–251.

[24] P. Willems, On MR^3-type Algorithms for the Tridiagonal Symmetric Eigenproblem and Bidiagonal SVD, Dissertation, University of Wuppertal (2010).

[25] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, Parallel Tiled QR Factorization for Multicore Architectures, Concurrency and Computation: Practice and Experience 20 (2008) 1573–1590.

[26] O. Marques, C. Voemel, J. Demmel, B. Parlett, Algorithm 880: A Testing Infrastructure for Symmetric Tridiagonal Eigensolvers, ACM Trans. Math. Software 35 (2008).