Aachen Institute for Advanced Study in Computational Engineering Science

Preprint: AICES-2011/09-2

25/September/2011

High-Performance Solvers for Large-Scale Dense Eigenproblems

M. Petschow, E. Peise and P. Bientinesi

RWTH AACHEN UNIVERSITY

AICES

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

List of AICES technical reports: http://www.aices.rwth-aachen.de/preprints

HIGH-PERFORMANCE SOLVERS FOR LARGE-SCALE DENSE EIGENPROBLEMS*

M. PETSCHOW†, E. PEISE†, AND P. BIENTINESI†

Abstract.
We introduce a new collection of solvers – subsequently called EleMRRR – for large-scale dense eigenproblems. EleMRRR solves various types of problems: Hermitian-definite and symmetric-definite generalized eigenproblems; Hermitian and symmetric standard eigenproblems; symmetric tridiagonal eigenproblems. Among these, the latter is of particular importance as it is a solver on its own right, as well as the computational kernel for the other types; we present a fast and scalable tridiagonal solver based on the Algorithm of *Multiple Relatively Robust Representations* – referred to as PMRRR. Like the other EleMRRR solvers, PMRRR is part of the freely available Elemental library, and is designed to fully support both message-passing (MPI) and multithreading parallelism (SMP). As a result, the solvers can equally be used in pure MPI or in hybrid MPI-SMP fashion. A thorough performance study of EleMRRR and ScaLAPACK's solvers is conducted on two supercomputers. Such a study, performed with up to 4096 cores, provides precise guidelines to assemble the fastest solver within the ScaLAPACK framework; it also indicates that the EleMRRR outperforms even the fastest solvers built from ScaLAPACK's components.

1. Introduction. A *generalized eigenproblem* (GENEIG) is identified by the equation

$$Ax = \lambda Bx\,, \tag{1.1}$$

in which $A, B \in \mathbb{C}^{n\times n}$ are known, and $\lambda \in \mathbb{C}$ and $x \in \mathbb{C}^n$ are sought after. The scalar λ and the vector x are called an *eigenvalue* and an *eigenvector*, respectively, and together they form an *eigenpair*. Other formulations of the generalized eigenproblem are $ABx = \lambda x$ and $BAx = \lambda x$. If B is the identity matrix I, Eq. (1.1) reduces to a *standard eigenproblem* $Ax = \lambda x$ (STDEIG); additionally, if A is tridiagonal, the problem is referred to as the *tridiagonal eigenproblem* (TRDEIG). Different numerical methods have been devised to solve Eq. (1.1) in accordance to the amount of eigenpairs requested and the properties of A and B. The emphasis of this paper is on the efficient solution of three classes of dense eigenproblems on modern large distributed-memory machines:

- *Hermitian-definite (or symmetric-definite) GENEIG:* In Eq. (1.1) and its variants both A and B are Hermitian (or symmetric[1]), and at least one of the two is positive or negative definite. Without loss of generality, we assume that B is positive definite.
- *Hermitian (or symmetric) STDEIG:* A is Hermitian, and $B = I$.
- *Symmetric TRDEIG:* A is symmetric and tridiagonal, and $B = I$.

For these three classes of problems, all eigenvalues are real and n linear independent eigenvectors can be chosen to be mutually orthogonal with respect to the inner product induced by B. This means $x_i^H B x_j = 0$ if $i \neq j$ and $x_i^H B x_j = 1$ if $i = j$ [11]. If k eigenpairs are computed, the solutions can be represented in the form

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

†Aachen Institute for advanced study in Computational Engineering Science, RWTH Aachen University ({petschow,peise,pauldj}@aices.rwth-aachen.de).

[1]If both A and $B \in \mathbb{R}^{n\times n}$, all complex quantities become real and the following discussion remains true, provided the words Hermitian and unitary are replaced by symmetric and orthogonal, respectively.

$AX = BX\Lambda$ with $X^H BX = I$, where the eigenvectors are the columns of $X \in \mathbb{C}^{n\times k}$, and the eigenvalues are the entries of the diagonal matrix $\Lambda \in \mathbb{R}^{k\times k}$.

If B is sufficiently well-conditioned with respect to inversion, the traditional approach for computing all or a significant fraction of the eigenpairs of the Hermitian-definite GENEIG relies on a reduction-backtransformation procedure, corresponding to three nested eigensolvers: generalized, standard, and tridiagonal. A GENEIG is transformed into a STDEIG [31], which in turn is reduced to a symmetric TRDEIG; once the eigenvectors of the latter are computed, they are mapped back to those of the generalized problem via two successive backtransformations. Overall, the process for solving a GENEIG consists of six stages.

1. *Cholesky factorization:* B is factored into LL^H;
2. *Reduction to a STDEIG:* From $B = LL^H$, Eq. (1.1) can be rewritten as $L^{-1}AL^{-H}L^H x = \lambda L^H x$. Computing $M := L^{-1}AL^{-H}$ and assigning $y := L^H x$, the problem becomes $My = \lambda y$, indicating that the eigenvalues of the original and the standard problems are the same, while the eigenvectors are related through the Cholesky factor L;
3. *Reduction to a TRDEIG:* A unitary matrix U is computed such that $T := U^H MU$ is symmetric tridiagonal. $My = \lambda y$ can be rewritten as $TU^H y = \lambda U^H y$; assigning $z := U^H y$, the problem becomes $Tz = \lambda z$. This equation shows that the eigenvalues of the standard problem are the same as those of the tridiagonal one, while the eigenvectors are related through the unitary factor U;
4. *Tridiagonal solver:* All or a subset of k eigenpairs of T are computed: $TZ = Z\Lambda$, where the diagonal matrix $\Lambda \in \mathbb{R}^{k\times k}$ contains the eigenvalues and the columns of $Z \in \mathbb{C}^{n\times k}$ contain the corresponding eigenvectors;
5. *Backtransformation #1:* Since $z = U^H y$, the eigenvectors of the TRDEIG are backtransformed to those of the STDEIG by computing $Y := UZ$;
6. *Backtransformation #2:* Since $y = L^H x$, the eigenvectors of the STDEIG are backtransformed to the desired eigenvectors of the GENEIG by computing $X := L^{-H}Y$.

With slight modifications to Stages 2 and 6, the same 6-stage procedure also covers eigenproblems in the form $ABx = \lambda x$ and $BAx = \lambda x$. In the first case, the reduction and the backtransformation become $M := L^H AL$ and $X := L^{-H}Y$, respectively; in the second case, they become $M := L^H AL$ and $X := LY$.

The solution of the aforementioned types of large-scale eigenproblems is integral to a number of scientific disciplines, particularly vibration analysis [4] and quantum chemistry [21, 35, 23, 43]. An example of an application where the solution of the eigenproblem is the most time consuming stage is Density Functional Theory. There, as part of a simulation, one has to solve a set of equations in a self-consistent fashion; this is accomplished by an iterative process in which each iteration involves the solution of dozens or even hundreds of generalized eigenproblems. In many cases, the application requires the lower 5-25% eigenpairs of the spectrum, and the size of such problems is usually in the tens of thousands. In other applications, in which the simulations do not follow an iterative process, it is instead common to encounters only one single eigenproblem, but potentially of very large size (50k–100k or more) [26, 28] In both scenarios, the problem size is not limited by the physics of the problem, but only by memory requirements and time-to-solution.

When the execution time and/or the memory requirement of a simulation become limiting factors, scientists place their hopes on massively parallel supercomputers. With respect to execution time, the use of more processors would ideally result in faster solutions. When memory is the limiting factor instead, the increase of resources from large distributed-memory environments should lead to solving larger problems.

We study the performance of eigensolvers for both situations: increasing the number of processors while keeping the problem size constant (*strong scaling*), and increasing the number of processors while keeping the memory per node constant (*weak scaling*). The latter has often a greater significance as in "practice, the problem size scales with the number of processors [...] to make use of the increased facilities" [20].

In this paper we make the following contributions.

- Given the nested nature of the GENEIG, STDEIG and TRDEIG, the latter is both a solver in its own right and the computational kernel for STDEIG and GENEIG. We present a novel tridiagonal solver, PMRRR, based on the algorithm of Multiple Relatively Robust Representations (MRRR) [13, 14], which merges the distributed and multithreaded approaches first introduced in [5] and [36].[2] PMRRR is well suited for both single node and for large scale massively parallel computations. Experimental results indicate that PMRRR is currently the fastest tridiagonal solver available, outperforming all the solvers included in LAPACK [2], ScaLAPACK [7] and Intel's Math Kernel Library (MKL).
- We introduce EleMRRR (from Elemental and PMRRR), a set of distributed-memory eigensolvers for the GENEIG, STDEIG, and TRDEIG. EleMRRR provides full support for hybrid message-passing and multithreading parallelism. If multithreading is not desired, EleMRRR can be used in purely message-passing mode.
- The five stages of reduction and backtransformation in EleMRRR are based on the Elemental library for the development of distributed-memory dense linear algebra routines [37]. This library embraces a two-dimensional cyclic element-wise matrix distribution, and attains performance comparable or even superior to the well established ScaLAPACK and PLAPACK parallel libraries [7, 46]. For the reduction to standard form, in particular, a new algorithm is used which delivers higher performance and scalability [38].
- A thorough performance study on two high-end computing platforms is provided. This study accomplishes two objectives. On the one hand it contributes guidelines on how to build – within the ScaLAPACK framework – an eigensolver faster than the existing ones. This is of particular interest for computational scientists and engineers as each of the commonly used[3] routines `PZHEGVX`, `PDSYGVX`, `PZHEEVD`, and `PDSYEVD` presents performance penalties that can be avoided by calling a different sequence of subroutines and choosing the right settings. On the other hand, the study indicates that EleMRRR is scalable – both strongly and weakly – to a large number of processors, and outperforms all the standard ScaLAPACK solvers.

The paper is organized as follows: In Section 2 we discuss related work and give experimental evidence that some widely used routines fail to deliver the desired performance. In Section 3 we concentrate on EleMRRR, with emphasis on the tridiagonal

[2] PMRRR should not be confused with the distributed-memory solver introduced in [5].
[3] See for example [43, 10, 25, 45].

stage – PMRRR. We present a thorough performance study on two state-of-the-art high-performance computer systems in Section 4. We show that ScaLAPACK contains a set of fast routines that can be combined to avoid the aforementioned performance problems, and we compare the resulting routines to our solver EleMRRR.

2. A Study of Existing Solvers. In this section we give a brief overview of existing methods and discuss well-known issues of widely used routines available in the current version of the ScaLAPACK library. We discuss the generalized, standard and symmetric tridiagonal eigenproblems in succession.

2.1. The Generalized Eigenproblem. In some cases, even if A and B do not satisfy the assumptions for the Hermitian-definite GENEIG, the problem can still be transformed into one that exhibits the desired properties [11]. In general, if A and B are dense but non-Hermitian, or if B has poor conditioning with respect to inversion, instead of the aforementioned 6-stage approach, the QZ algorithm [32] should be used. Such an algorithm is much more time consuming, but does not put any restriction on the input matrices. A parallel distributed-memory implementation is discussed in [1].

The ScaLAPACK library contains routines for the three classes of eigenproblems we consider in this paper. A complete list of routine names for both the solvers and the individual stages is given in Table 2.1.

In particular, `PZHEGVX` and `PDSYGVX` are the double precision drivers for the Hermitian-definite and symmetric-definite GENEIGs, respectively.

Stage	Complex	Real	Description
1—6	`PZHEGVX`	`PDSYGVX`	GENEIG based on Bisection and Inverse Iteration
3–5	`PZHEEVX`	`PDSYEVX`	STDEIG based on Bisection and Inverse Iteration
3–5	`PZHEEV`	`PDSYEV`	STDEIG based on the QR algorithm
3–5	`PZHEEVD`	`PDSYEVD`	STDEIG based on the Divide and Conquer algorithm
3–5	`PZHEEVR`	`PDSYEVR`	STDEIG based on the MRRR algorithm
4	-	`PDSTEDC`	TRDEIG based on the Divide and Conquer algorithm
4	-	`PDSTEGR`	TRDEIG based on the MRRR algorithm
1	`PZPOTRF`	`PDPOTRF`	Cholesky factorization
2	`PZHEGST`	`PDSYGST`	Reduction to STDEIG
2	`PZHENGST`	`PDSYNGST`	Reduction to STDEIG (square grid of processes)
3	`PZHETRD`	`PDSYTRD`	Reduction to TRDEIG
3	`PZHENTRD`	`PDSYNTRD`	Reduction to TRDEIG (square grid of processes)
4	-	`PDSTEBZ`	Eigenvalues of a tridiagonal matrix using Bisection
4	`PZSTEIN`	`PDSTEIN`	Eigenvectors of a tridiagonal matrix using Inverse Iteration
5	`PZUNMTR`	`PDORMTR`	Backtransformation to STDEIG
6	`PZTRSM`	`PDTRSM`	Backtransformation to GENEIG (Type #1 or #2)
6	`PZTRMM`	`PDTRMM`	Backtransformation to GENEIG (Type #3)

TABLE 2.1
List of relevant ScaLAPACK routine names.

In Fig. 2.1 we report on the weak scalability for ScaLAPACK's `PZHEGVX`, computing the smallest 15% of the eigenpairs of GENEIGs in the form of $Ax = \lambda Bx$.[4]

[4]The timings were generated on the *Juropa* supercomputer; a detailed description of the architecture and the experimental setup is provided in Section 4. We used all default parameters. In particular the parameter *orfac* that indicates which eigenvectors should be re-orthogonalized during inverse iteration has the default value 10^{-3}. In practice it is possible to use a less restrictive choice, e.g. by setting *orfac* to $\min(10^{-3}, \text{const} \cdot n^{-1})$. In order to exploit ScaLAPACK's fastest reduction routines, the lower triangular part of the matrices is stored and referenced.

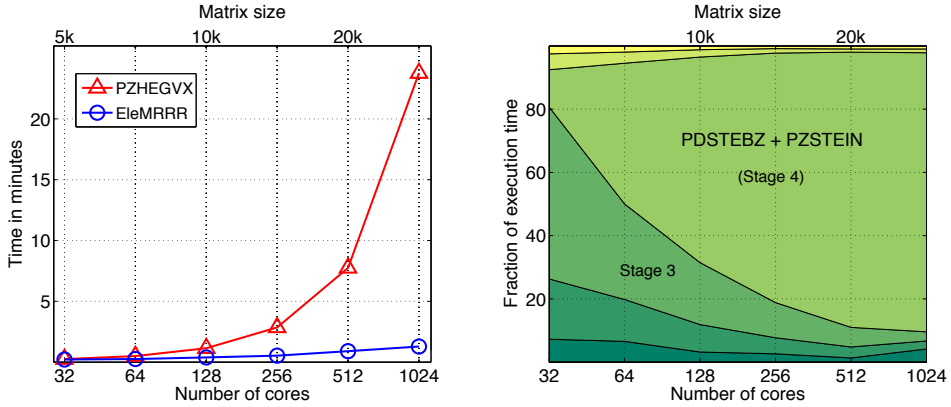

FIG. 2.1. *Weak scalability for the computation of the smallest 15% of the eigenpairs of GENEIGs in the form of $Ax = \lambda Bx$. As done commonly in practice [43], the eigenvectors are requested to be numerically orthogonal. Left: Total execution time of* `PZHEGVX` *and EleMRRR. Right: Fraction of the execution time spent in the six stages of* `PZHEGVX`*, from bottom to top (see Table 2.1).*

The left graph indicates that as the problem size and the number of processors increase, `PZHEGVX` does not scale nearly as well as the EleMRRR solver presenteted in Section 3. In the right graph we show the breakdown of the execution time for the six stages. With the exception of the tridiagonal solver – Stage 4 – the computation time of each stage is cubic in the matrix size n and independent of the input data. Conversely, the execution time of the tridiagonal eigensolver is greatly influenced by the method used and the properties of the input matrix, most importantly by how the eigenvalues are distributed within the eigenspectrum. It is evident that the routines `PDSTEBZ` and `PZSTEIN`, which implement the Bisection and Inverse Iteration (BI) tridiagonal eigensolver, are the main cause for the poor performance of `PZHEGVX`. For the problem of size 20,000, they are responsible for almost 90% of the compute time. BI's poor performance is a well understood phenomenon, e.g. [8], directly related to the effort necessary to re-orthogonalize eigenvectors corresponding to clustered eigenvalues. This issue led to the development of an improved version of Inverse Iteration, the MRRR algorithm, that avoids re-orthogonalization even when the eigenvalues are clustered. In addition to the performance issue, `PZHEGVX` also suffers from memory imbalances, as all the eigenvalues belonging to a cluster are computed on a single processor.

In light of the above considerations, the use of ScaLAPACK's routines based on Bisection and Inverse Iteration (BI) is not recommended.

We therefore do not provide further comparisons between EleMRRR and `PZHEGVX` or `PDSYGVX`. Instead, in Section 4 we illustrate how the performance of these drivers changes when the BI algorithm for the tridiagonal eigensolver is replaced with other – faster – methods available in ScaLAPACK, namely the Divide and Conquer algorithm (DC) and the MRRR algorithm [44, 47].

2.2. The Standard Eigenproblem. We restrict the discussion to dense Hermitian and symmetric problems. Such problems are originated by transforming GENEIGs into STDEIGs, as discussed in Section 1, and arise frequently in a variety of applications ranging from electrodynamics to macro-economics [39].

The standard approach to solve a STDEIG consists of three stages, corresponding

to stages 3–5 in the solution of a GENEIG: (a) Reduction to a symmetric TRDEIG; (b) Solving the tridiagonal eigenproblem; (c) Backtransformation of the eigenvectors. A detailed analysis of the three stage approach, concentrating on the first and third stage, can be found in [42].

An alternative approach is the Successive Band Reduction (SBR) [6]. The idea is to split the reduction to tridiagonal form in two (or more) stages. In all but the last stage, the matrix is reduced to banded form with strictly decreasing bandwidths. Unlike the direct reduction to tridiagonal form, these stages can take full advantage of the highly efficient kernels from the Basic Linear Algebra Subprograms (BLAS) library [17], thus attaining high-performance despite an increase of the operation count. The reduction is then completed with a final band-to-tridiagonal reduction.

The downside of such a strategy lays in the accumulation of the orthogonal transforms. Thus, when the eigenvectors are requested the SBR routines are not competitive. For this reason, SBR is normally used when only the eigenvalues are requested. In a recent publication it is suggested that on highly parallel systems the SBR approach might be faster than the direct reduction to tridiagonal form even when also the eigenvectors are computed [3]. Unfortunately the SBR approach is only compared with the `PDSYTRD` routine and not with the faster and more scalable `PDSYNTRD` [22].

ScaLAPACK offers a number of routines for the Hermitian (or symmetric) STDEIG, differing in the algorithm of choice for the TRDEIG. For the Hermitian case the routines `PZHEEVX`, `PZHEEV`, and `PZHEEVD` implement BI [48], the QR algorithm [27, 18], and the DC algorithm [9, 19], respectively. Additionally, `PZHEEVR` [47] is a publicly available routine (not yet included in ScaLAPACK) that is based on the parallel MRRR algorithm. All these routines have counterparts for the symmetric case.

Among the solvers presently available in ScaLAPACK, only `PZHEEVX` (BI) offers the possibility of computing a subset of the eigenspectrum. Because of this, the routine is widely used, even though, as highlighted in the previous section, it is highly non-scalable. Therefore we do not further discuss `PZHEEVX` and instead compare with the faster `PZHEEVR` (MRRR), which also allows for subset computation. Similarly, as the QR algorithm is known to be slower than DC [5], *the use of ScaLAPACK's routines based on QR is not recommended*; in the future experiments, we omit comparisons with QR-based routines.

We point now our attention to the widely used routine `PZHEEVD` (DC); as in the previous section, we report on the weak scalability. In Fig. 2.2 we show the results for `PZHEEVD` and EleMRRR in a similar experiment as shown for GENEIG in Fig. 2.1. All the eigenpairs are computed, since `PZHEEVD` does not allow for subset computations. While in the previous section we identified that BI was the cause for poor scalability, in `PZHEEVD` the solution of the tridiagonal eigenproblem takes less than 10% of the total execution time. Instead, as the matrix size increases, the reduction to tridiagonal form (`PZHETRD`) becomes the computational bottleneck, requiring up to 70% of the total time for the largest problem shown. From a comparison of Fig. 2.2 left and right, it is apparent that for large problems `PZHETRD` alone requires more time than the complete solution using EleMRRR.

Also included in ScaLAPACK is `PZHENTRD`, a routine for the reduction to TRDEIG especially optimized for square processor grids. The performance improvement with respect to `PZHETRD` is so dramatic that for this stage it is always convenient to limit the computation to a square number of processors and redistribute the matrix accordingly [22]. It is important to notice that the performance benefit of `PZHENTRD` can only be exploited if the lower part input matrix is stored, since otherwise the slow

routine PZHETRD is invoked.[5]

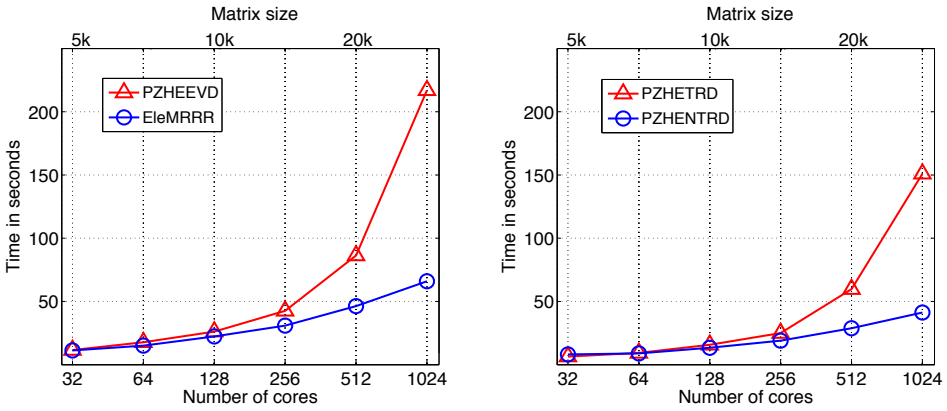

FIG. 2.2. *Weak scalability for the computation of all eigenpairs of STDEIGs using DC. Left: Total execution time of* PZHEEVD *and EleMRRR. Right: Execution time for ScaLAPACK's routines* PZHETRD *and* PZHENTRD*, providing the reduction to tridiagonal form. The former, used within the routine* PZHEEVD*, causes a performance penalty and accounts for much of the time difference compared with EleMRRR.*

> ScaLAPACK's reduction routines (PxxxNGST and PxxxNTRD) optimized for square grids of processors are to be preferred over the regular reduction routines, even for non-square grids; moreover, only the lower triangular part of the matrices should be referenced.

For this reason in Section 4, we only use the faster routines PZHENTRD and PZHENGST to build the fastest solver within the ScaLAPACK framework.

2.3. The Tridiagonal Eigenproblem. At the core of the reduction-backtransformation approach for GENEIGs and STDEIGs lies the symmetric tridiagonal eigenproblem. As seen in Section 2.1, this might account for a significant computational portion of the solution process. One of the main differences – and often the only difference – among solvers for GENEIGs and STDEIGs lays in the method used for the TRDEIG. In Section 2.2 we already mentioned four methods: BI, QR, DC, and MRRR. In the case of BI and the QR algorithm, we also gave experimental justification for not including further comparisons.

In contrast to all other stages of the GENEIG and STDEIG, the number of arithmetic operations of the tridiagonal eigensolver varies in accordance to the input data. For this reason, depending on the matrix entries, either DC or MRRR may be the faster. Fig. 2.3 provides an example of how performance is influenced by the input data. The algorithms are compared on two types of test matrices: "1–2–1" and "Wilkinson". The former contains ones on the subdiagonals and twos on the diagonal; its eigenpairs are known analytically. In the latter, the subdiagonals contain ones and the diagonal equals the vector $(m, m-1, \ldots, 1, 0, 1, \ldots, m)$ where $m = n-1/2$. Due to the phenomenon of deflation this matrix is known to favor the DC algorithm [9]. In the figure we also inlcude the timings for our novel solver PMRRR: For both matrix types it eventually becomes the fastest solver. A detailed discussion of PMRRR follows in the next section.

[5]Similar considerations also apply to reduction to standard form and the routines PZHEGST and PZHENGST, see [38].

We give now a short comparison of the basic functionalities of ScaLAPACK's routines `PDSTEDC` and the to-be-included `PDSTEGR`, which implement the tridiagonal DC and MRRR algorithms, respectively [44, 47].

In the worst case, `PDSTEDC` computes all eigenpairs at the cost of $O(n^3)$ floating point operations (*flops*); in practice though, the flop count is slightly lower (due to deflation), and since most of the computation is cast in terms of fast BLAS-3 kernels, the run time behavior follows $n^{2.5}$ [12].

While `PDSTEDC` cannot compute a subset of $k < n$ eigenpairs, the MRRR routine `PDSTEGR` returns the eigenpairs at the reduced cost of $O(nk)$ flops. In terms of orthogonality of the eigenvectors, normally DC is superior to MRRR. On the other hand, DC is more expensive also in terms of memory usage: It requires an extra $O(n^2)$ memory for work space.

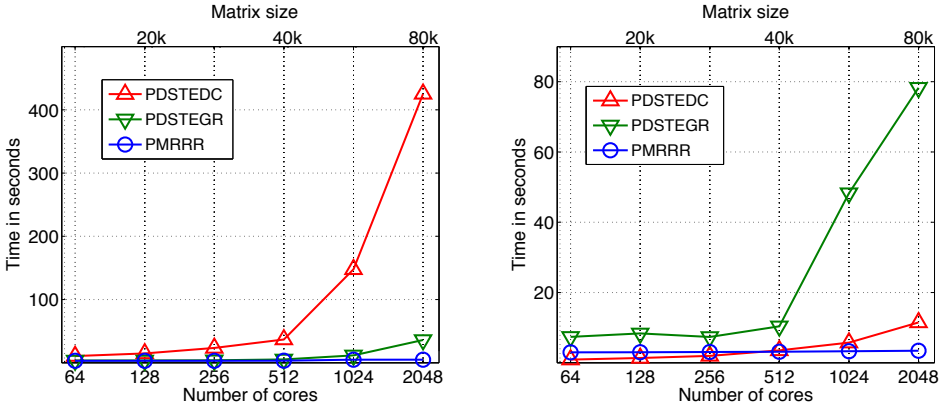

FIG. 2.3. *Weak scalability for the computation of all the eigenpairs of the TRDEIG for two different test matrix types. The left and right graphs have different scales. Left: "1–2–1" matrix; Right: "Wilkinson" matrix. In contrast to the results reported in [47], even when the matrices offer an opportunity for heavy deflation, our PMRRR becomes the fastest solver.*

> ScaLAPACK's tridiagonal eigensolver routines based on DC (`PDSTEDC`) and MRRR (`PDSTEGR`) are generally fast and reasonably scalable; depending on the target architecture and specific requirements from the applications, either one may be used. Specifically, if only a small subset of the spectrum has to be computed, in terms of performance, the MRRR-based solvers are to be preferred to DC.

In Section 4 we include experimental data on GENEIGs for both DC and MRRR. Further comparisons for all the aforementioned methods on uni-processors and distributed-memory systems can be found in [12, 5, 47].

3. Elemental's Generalized and Standard Eigensolvers. Elemental is a framework for dense matrix computations on distributed-memory architectures [37]. The main objective of the project is to ease the process of implementing matrix operations, without conceding performance or scalability. The code resembles a high-level description of the algorithm, so that the details relative to data distribution and communication are hidden. Performance is achieved through an elemental two-dimensional cyclic data distribution.

Elemental supports many frequently encountered dense matrix operations; these include the Cholesky, QR and LU factorizations, and linear systems. Recently, rou-

tines for Hermitian and symmetric eigenvalue problems have also been added. In particular, Elemental supports Hermitian-definite and symmetric-definite GENEIGs, Hermitian and symmetric STDEIGs, as well as skew-Hermitian and skew-symmetric STDEIGs. All the eigensolvers follow the classical reduction and backtransformation approach described in Section 1.

In terms of memory, Elemental's solvers require for the Hermitian GENEIG and STDEIG is about $(2n^2 + 1.6nk)/nproc$ and $(n^2 + 1.6nk)/nproc$ complex numbers per process, respectively. Similarly, for the symmetric case, the requirement per process is about $(2n^2 + 2.1nk)/nproc$ and $(n^2 + 2.1nk)/nproc$ real numbers, respectively.

In terms of accuracy, Elemental's solvers are equivalent to their sequential counterparts. We therefore do not report accuracy results.

Detailed discussions the reduction and backtransformation stages, both in general and within the Elemental environment, can be found in [42, 40, 22, 38].

For the solution of TRDEIGs, Elemental incorporates PMRRR, a new parallel variant of the MRRR algorithm. PMRRR combines two solvers: one based on message-passing parallelism [5, 47] and another based on multithreading [36]. As a consequence, the solver provides the user with multiple parallel programming models (multithreading, message-passing, a hybrid of both), so that the parallelism offered by modern distributed-memory architectures can be fully exploited. In the following two subsections we focus our attention on PMRRR, our tridiagonal solver.

3.1. PMRRR as the Core of Elemental's Eigensolver. The MRRR algorithm is a form of inverse iteration. Its salient feature is that the costly re-orthogonalization, necessary when the eigenvalues are clustered together, is entirely removed. As the experiment described in Fig. 2.1 shows, the re-orthogonalization may affect the execution time dramatically: in computing the lowest 15% of the eigenspectrum of a matrix of size 20,000 using 512 cores, inverse iteration takes about 404 seconds, while PMRRR requires less than 0.3 seconds.

To achieve this performance, the classical algorithm of inverse iteration was modified significantly. The basic inverse iteration requires the repeated solution of linear systems; given a computed eigenvalue $\hat{\lambda}_j$ and a starting vector $\hat{z}_j^{(0)}$ the iterative process

$$(T - \hat{\lambda}_j I)\hat{z}_j^{(i+1)} = \hat{z}_j^{(i)} \tag{3.1}$$

will, under some restrictions, result an an eigenvector approximation $\hat{z}_j$ [24]. The connection between inverse iteration and the MRRR algorithm is discussed in a number of articles [29, 16]. Here we only mention three main differences: (1) Instead of representing tridiagonal matrices by their diagonal and subdiagonal entries, MRRR uses *Relatively Robust Representations* (RRRs); these are representations with the property that small relative perturbations to the data result in small relative perturbations to the eigenpairs [34]; (2) the selection of a nearly optimal right-hand side vector $\hat{z}_j^{(0)}$ for a one-step inverse iteration [33]; and (3) the use of so-called *twisted factorizations* to solve linear systems [33].

In the following, we briefly discuss both the mathematical foundations of the MRRR algorithm, and how parallelism is organized in PMRRR.

3.1.1. The MRRR algorithm. At first, a representation RRR_0 is chosen so that it defines all the desired eigenpairs in a relatively robust way.[6] For example, a

[6]Without loss of generality, we will assume that T is irreducible: No off-diagonal element is smaller in magnitude than a certain threshold.

factorization of the form $L_0 D_0 L_0^T = T - \sigma I$, is an RRR for all the eigenpairs [34]; here, D_0 is diagonal, L_0 is lower unit bidiagonal, and σ is a scalar such that $T - \sigma I$ definite. As a second step, bisection is used to compute approximations to each eigenvalue $\hat{\lambda}_j$. At the cost of $O(n)$ flops, bisection guarantees that the eigenvalues have high relative accuracy [30].

At this point, for all the eigenvalues λ_j that have large relative gaps, it is possible to compute their corresponding eigenvectors: By using one step of inverse iteration with a twisted factorization, the error angle $\angle(\hat{z}_j, z_j)$ between the computed eigenvector $\hat{z}_j$ and true eigenvector z_j satisfies

$$|\sin \angle(\hat{z}_j, z_j)| \leq \frac{O(n\varepsilon)}{\text{relgap}(\hat{\lambda}_j)} \quad . \tag{3.2}$$

Here ε is the machine precision and $\text{relgap}(\hat{\lambda}_j) = \min_{i \neq j} |\hat{\lambda}_j - \hat{\lambda}_i| / |\hat{\lambda}_j|$ is the relative gap. The denominator in Eq. (3.2) is the reason for which the process can be applied only to well-separated eigenvalues. Such well-separated eigenvalues are called *singletons*. Indeed, for singletons the computed eigenvectors have a small error angle to the true eigenvector and consequently are numerically orthogonal.

In contrast, when a set of consecutive eigenvalues have small relative gaps – that is, they form a *cluster* – the current RRR cannot be used to obtain numerically orthogonal eigenvectors. Instead, one exploits the fact that the relative gap is not invariant to matrix shifts. The algorithm then proceeds by constructing a new RRR $\{L_c, D_c\}$ in a mixed relatively stable way: $L_c D_c L_c^T = L_0 D_0 L_0^T - \sigma_c I$. By shifting, the relative gaps are modified by a factor $|\hat{\lambda}_j| / |\hat{\lambda}_j - \sigma_c|$; thus σ_c is chosen close to one of the eigenvalues in the cluster, so that at least one of them becomes well-separated. Once the new RRR is established, the eigenvalues in the cluster are refined to high relative accuracy with respect to the new RRR. At this point, the shifted and refined eigenvalues are classified as singletons and clusters. Ideally, all of them have a large relative gap, that is are singletons, and a set of orthogonal eigenvector for a slightly perturbed invariant subspace of $L_0 D_0 L_0^T$ can be computed using $L_c D_c L_c^T$. In the case that there are clustered eigenvalues within the cluster, we must repeat the procedure recursively until all eigenpairs of the cluster are computed.

As a detailed discussion of the MRRR algorithm is outside the scope of this article, for further information we refer the readers to [33, 13, 15, 14, 49] and references therein.

3.1.2. PMRRR's parallelism. Our parallelization strategy consists of two layers: a global and a local one. At the global level, the k eigenvalues and eigenvectors are statically divided into equal parts and assigned to the processes. Since the unfolding of the algorithm depends on the eigenspectrum, it is still possible that the workload is not perfectly balanced among processes. At the local level (within each process), the computation is decomposed into tasks, to be executed in parallel by multiple threads. The processing of these tasks leads to the dynamic generation of new tasks and might involve communication with other processes. The newly generated tasks are then likewise enqueued.

When executed with *nproc* processes, the algorithm is started by broadcasting the input matrix and by redundantly computing the initial representation RRR_0. Once this is available, the computation of the approximations $\hat{\lambda}_j$ of k eigenvalues of $L_0 D_0 L_0^T$ is embarrassingly parallel: Each process is responsible for at most $epp = \lceil k/nproc \rceil$

eigenvalues;[7] similarly, each of the *nthread* threads within a process has the task to compute at most $ept = \lceil epp/nthread \rceil$ eigenvalues. The processes then gather all the eigenvalues, and the corresponding eigenpairs are assigned to the processes as desired.

Locally, the calculation of the eigenvectors[8] is split into computational tasks of three types: a set of singletons, clusters that require no communication, and clusters that require communication with other processes. The computation associated with each of the three types is detailed below.

1. *A set of singletons.* The corresponding eigenvectors are computed locally. No further communication among processes is necessary.
2. *A cluster requring no communication.* When the cluster contains eigenvalues assigned to only one process, no cooperation among processes is needed. The four necessary steps are the same as those for the cluster task in [36]: A new RRR is computed; the eigenvalues are refined to relative accuracy with respect to the new RRR; the refined eigenvalues are classified into singletons and clusters; the corresponding tasks are enqueued into the local work queue.
3. *A cluster requring communication.* When the cluster contains eigenvalues assigned to more than one process, inter-process communication is needed. In this case, all the processes involved perform the following steps: A new RRR is computed redundantly; the local portion of the eigenvalues is refined; the eigenvalues of the cluster are all gathered and classified, originating tasks corresponding to the three scenarios just described.

Multithreading support is easily obtained by having multiple threads dequeue and execute tasks. The tasks are dynamically generated: their number and size highly depends on the the spectral distribution of the input matrix; for this reason, the execution time for matrices of the same size may differ noticeably. By contrast, the memory requirement is matrix independent ($O(nk/nproc)$ memory per process), and perfect memory balance is achieved [5, 47].

Our tests show that the hybrid parallelization approach is equally or slightly faster than the one purely based on MPI. This is generally true for architectures with a high degree of inter-node parallelism and limited intra-node parallelism. By contrast, on architectures with a small degree of inter-node parallelism and high degree of intra-node parallelism, we expect the hybrid execution of PMRRR to be preferable to the pure MPI one.

4. Experiments. In the next two sections we present experimental results for the execution of GENEIGs on two state-of-art supercomputers at the Research Center Jülich, Germany: *Juropa* and *Jugene*.

4.1. Juropa. Juropa consists of 2208 nodes, each comprising two Intel *Xeon X5570 Nehalem* quad-core processors running at 2.93 GHz with 24 GB of memory. The nodes are connected by an *Infiniband QDR* network with a Fat-tree topology.

All tested routines were compiled using the *Intel compilers* (ver. 11.1) with the flag `-O3` and linked to the *ParTec's ParaStation MPI* library (ver. 5.0.23).[9] Generally we used a two-dimensional process grid $P_r \times P_c$ with $P_r = P_c$ whenever possible, and $P_c = 2P_r$ otherwise. The quantities P_r and P_c describe the number of rows and

[7] PMRRR (ver. 0.6) computes approximations to all n eigenvalues at this stage, such that $epp = \lceil n/nproc \rceil$.

[8] We only refer to the calculation of the *eigenvectors* here, although the eigenvalues are also modified in this part of the computation.

[9] Version 5.0.24 was used when support for multithreading was needed.

columns of the grid, respectively. If not stated otherwise, one process per core was employed.

The latest version of ScaLAPACK (ver. 1.8) was used in conjunction with Intel's MKL BLAS (ver. 10.2). From extensive testing, we identified that in all cases the optimal block size was close to 32; therefore we carried out the ScaLAPACK experiments only with block size of 16, 32, 48; the best result out of this pool is then reported.

Since no driver for the GENEIG that makes use of DC is available, we refer to ScaLAPACK's DC as the sequence of routines `PZPOTRF`–`PZHENGST`–`PZHENTRD`–`PDSTEDC`–`PZUNMTR`–`PZTRSM`, as listed in Table 2.1. Similarly, we refer to ScaLAPACK's MRRR as the same sequence with `PDSTEDC` replaced by `PDSTEGR`.

We stress that the slow routines `PZHEGST` and `PZHETRD`, for the reduction to standard and tridiagonal form, respectively, were *not* used, and instead replaced by the faster `PZHENGST` and `PZHENTRD`. Because of this, the DC timings do *not* correspond to the sequence `PZPOTRF`–`PZHENGST`–`PZHEEVD`–`PZTRSM`, as the STDEIG solver `PZHEEVD` makes use of `PZHETRD`. Instead, the MRRR timings do correspond to the sequence `PZPOTRF`–`PZHENGST`–`PZHEEVR`–`PZTRSM`, because the STDEIG solver `PZHEEVR` makes use of `PZHENTRD`. In order to make use of ScaLAPACK's fast reduction routines, only the lower triangular part of the matrices is referenced. This is because when referencing the upper triangular part, the routines `PZHENGST` and `PZHENTRD` are merely wrappers to the slower `PZHEGST` and `PZHETRD`, respectively.

Elemental (ver. 0.6) – incorporating PMRRR (ver. 0.6) – was used for the EleMRRR timings. In general, since Elemental does not tie the algorithmic block size to the distribution block size, different block sizes could be used for each of the stages. We do not exploit this fact in the reported timings. Instead the same block size is used for all stages. A block size of around 96 was in all cases optimal, therefore experiments were carried out for block sizes of 64, 96 and 128, but only the best timings are reported.[10]

Since the timings of TRDEIGs depend on the input data, so does the overall solver. In order to compare fairly different solvers, we fixed the tridiagonal input matrix by using the 1–2–1 type. The performance of every other stage is data independent. Moreover, since the output of the tridiagonal solvers has to be in a format suitable for the backtransformation, the MRRR-based routines have to undergo a stage of data redistribution. For this reason, in this section the timings for TRDEIGs include the cost for such a stage.

In next two subsections we show results for both strong and weak scaling. In all experiments the number of nodes are increased from 8 to 256. Since each node consists of two quad-core processors, this corresponds to 64 to 2048 cores.

4.1.1. Strong Scaling. We present timings of EleMRRR for the computation of all the eigenpairs of the generalized Hermitian eigenproblem $Ax = \lambda Bx$, where the problem size $n = 20{,}000$ is constant. The results are displayed in Fig. 4.1. As a reference, we include timings of ScaLAPACK.[11] The right side of the Figure shows the *parallel efficiency* – defined as $e_p = (t_{ref} \cdot p_{ref})/(t_p \cdot p)$ – where t_p denotes the

[10] The block size for matrix vector products were fixed to 32 in all cases. For the biggest matrices in the weak scaling experiment only the block size of 32 and 96 were used for ScaLAPACK and EleMRRR, respectively.

[11] We did not investigate the cause for the increased run time of ScaLAPACK using 1024 and 2048 cores. While most subroutines in the sequence are slower compared with the run time using 512 cores, `PZHENTRD` scales well up to 2048 cores.

execution time on p processors, and t_{ref} the execution time on p_{ref} processors. The reference is the execution using 8 nodes or 64 cores.

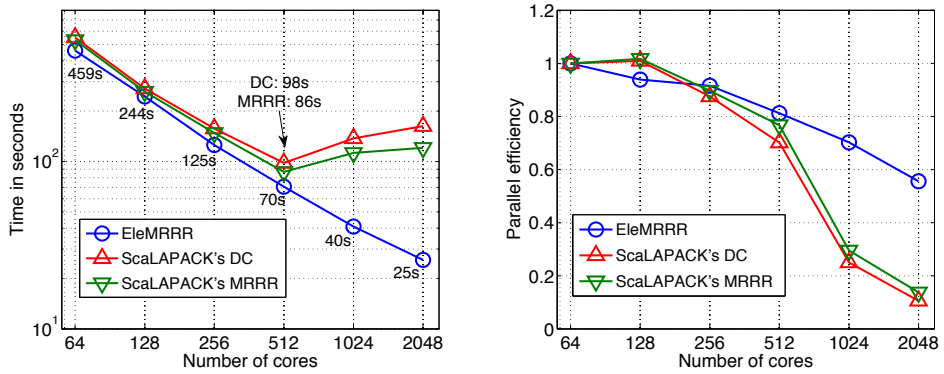

FIG. 4.1. *EleMRRR's strong scalability for the computation of all the eigenpairs for a GENEIG in the form $Ax = \lambda Bx$. Matrices A and B are of size 20,000. Left: Total execution time in a log-log scale. Right: Parallel efficiency; normalized to the exucution using 64 cores.*

Once the sequence of routines has been rectified as indicated earlier, the performance of ScaLAPACK's solvers with less than 512 cores becomes comparable to that of EleMRRR (Fig. 4.1): For 64 to 512 cores DC is about 10% to 40% slower than EleMRRR, while MRRR is about 7% to 20% slower. The advantage of EleMRRR mainly comes from the Stages 1, 2 and 6, i.e., those relative to GENEIG. The timings for the kernels of Stages 3–5 (STDEIG) are nearly identical, with DC slightly slower than both MRRR-based solutions.

The story for 1024 and 2048 cores changes: The performance of ScaLAPACK's GENEIG routines drops dramatically. Compared to DC, EleMRRR is about 3.3 and 6.3 times faster; With respect to MRRR instead, EleMRRR is 2.7 and 4.7 times faster. The same holds for the STDEIG, where EleMRRR is about 2.9 and 6.2 times faster than DC and 1.9 and 3.7 times faster than MRRR.

A study on the Juropa supercomputer suggests that the run time of applications can be greatly affected by settings of the underlying MPI implementation [41]. In particular, the switch from static – the default setting – to dynamic memory allocation for MPI connections may result in improved performance. In regards to the performance degradation appearing in Fig. 4.1 *(left)*, such a switch positively impacts some of the GENEIG's stages, including `PDSTEDC`; on the other hand it gravely worsens other stages, including `PDSTEGR`. Overall, ScaLAPACK's DC would only marginally improves, while MRRR's performance would greatly deteriorates. As a consequence, we employ the default MPI settings in all later experiments on Juropa.

In Fig. 4.2 we take a closer look at the six different stages of EleMRRR. The left panel tells us that roughly one third of EleMRRR's execution time – corresponding to Stages 4, 5 and 6 – is proportional to the fraction of computed eigenpairs. Computing a small fraction of eigenpairs would therefore require roughly about two thirds of computing the complete decomposition. On the right-hand panel of Fig. 4.2, we report on the parallel efficiency for all six stages separately. This information is of particular interest in conjunction with the figure on the right, showing the fraction of time spent in each of the stages, to determine if and when one of the routines becomes a bottleneck because of bad scaling.

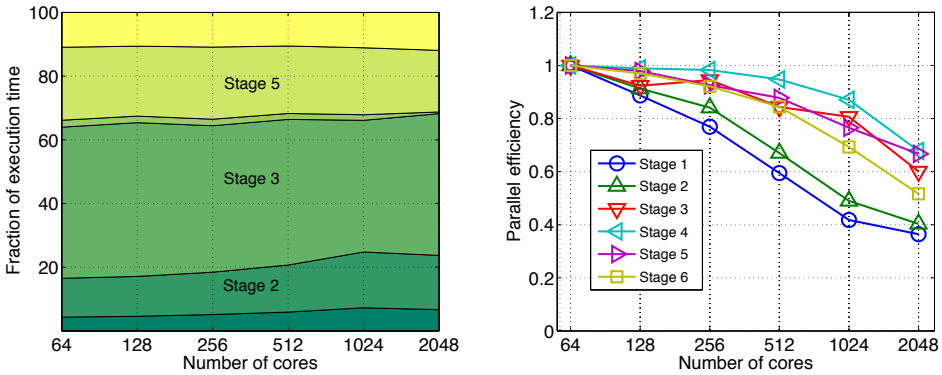

FIG. 4.2. *Strong scalability for the computation of all the eigenpairs for a GENEIG in the form* $Ax = \lambda Bx$. *Left: Parallel efficiency for all six stages of the computation. Achieving a high parallel efficiency is especially important for the stages in which most part of the computation is spent. Right: Fraction of time spent in all six part of the computation. The most time consuming stages are the reduction to tridiagonal form, the reduction to standard form, and the backtransformation to STDEIG, while the tridiagonal eigensolver is negligible. The time spent in the last three stages is proportional to the number of eigenpairs computed.*

The tridiagonal eigensolver – Stage 4 – obtains the highest parallel efficiency. On the other hand, it contributes for less than 2.2% to the overall run time and is therefore negligible in all experiments.

Up to 1024 cores, ScaLAPACK's MRRR shows a similar behavior: the tridiagonal stage makes up for less than 6% of the execution time. With 2048 cores instead, the percentage increases up to 21%. The story is different for DC, as the fraction spent in the tridiagonal stage increases from about 4.5% with 64 cores to 41% with 2048 cores. This analysis suggests that the tridiagonal stage, unless as scalable as the other stages, will eventually account for a significant portion of the execution time.

4.1.2. Weak Scaling. Fig. 4.3 illustrates EleMRRR's timings for the computation of all the eigenpairs of $Ax = \lambda Bx$. This time the matrix size (from 14,142 to 80,000) increases together with the number of cores (from 64 to 2048). The right graph contains the obtained *parallel efficiency*, defined as $e_p = (t_{ref} \cdot p_{ref} \cdot n^3)/(t_p \cdot p \cdot n_{ref}^3)$, where all the quantities are like in the previous section and n_{ref} denotes the smallest matrix size in the experiment.

In the experiments using 512 cores and less, EleMRRR slightly outperforms ScaLAPACK. The right graph indicates that EleMRRR scales well to large problem sizes and high number of processes, with parallel efficiency close to one. Thanks to its better scalability, for the biggest problem EleMRRR is 2.1 and 2.5 times faster than ScaLAPACK's MRRR and DC, respectively.

The execution time is broken down into states in Fig. 4.4 *(left)*. Four comments follow: (a) The time spent in PMRRR – Stage 4 – is in the range of 2.5% to 0.7% and it is is completely negligible, especially for large problem sizes.[12] (b) The timings corresponding to a STDEIG – Stages 3 to 5 – cover between 72% and 71% the GENEIG execution time. (c) The part of the solver whose execution time is roughly proportional to the fraction of desired eigenpairs – Stages 4 to 6 – makes up 37% to 32% of the execution for both a STDEIG and a GENEIG. (d) No one stage in

[12]The timings of only Stage 4 are detailed in Fig. 2.3.

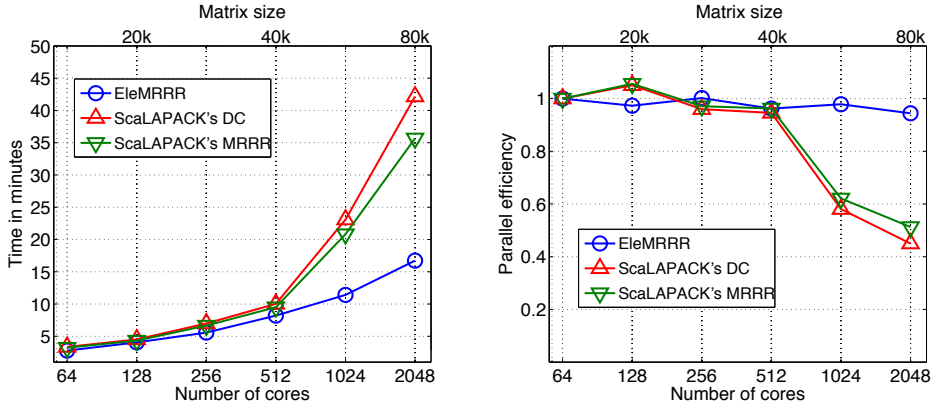

FIG. 4.3. *Weak scalability for the computation of all the eigenpairs of GENEIGs in the form* $Ax = \lambda Bx$. *Matrices* A *and* B *are of varying size. Left: Total execution time. Right: Parallel efficiency; normalized to the execution using 64 cores. All three routines are fast and efficient up to 512 cores. For higher numbers of nodes the efficiency of the ScaLAPACK routines drops, while EleMRRR obtains almost perfect results.*

EleMRRR is becoming a bottleneck, as all of them scale equally well.

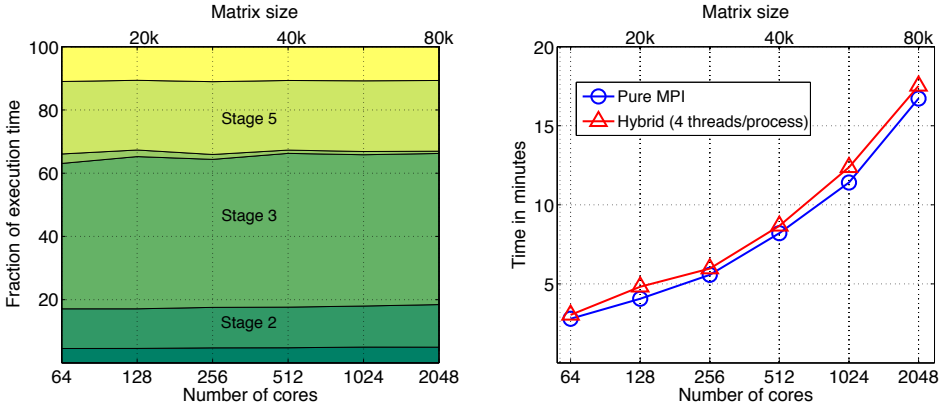

FIG. 4.4. *EleMRRR's weak scalability for the computation of all the eigenpairs of GENEIGs in the form* $Ax = \lambda Bx$. *Left: Fraction of the execution time spent in the six stages, from bottom to top. Right: Comparison between a pure MPI execution and a hybrid execution using one process per socket with four threads per process.*

Fig. 4.4 *(right)* shows the execution of EleMRRR using one process per socket with four threads per process. The resulting execution time is roughly the same as for the pure MPI execution, highlighting the potential of Elemental's hybrid mode. Similar experiments with a higher degree of multi-threading would positively affect PMRRR, but not the reduction the tridiagonal form.

4.2. Jugene. In this section, we verify the prior results obtained on Juropa for a different architecture – namely, the *BlueGene/P* installation *Jugene*. Jugene consists of 73,728 nodes each equipped with a quad core PowerPC 450 processor running at 850 MHz. Each node has 2 GB of memory.

All routines were compiled using the *IBM XL* compilers (ver. 9.0) in combination with the vendor tuned *IBM* MPI library. As on Juropa, we used a square processor grid $P_r = P_c$ whenever possible and $P_r = 2P_c$ otherwise. Similarly, ScaLAPACK (ver. 1.8) in conjunction with the vendor-tuned BLAS included in the ESSL library (ver. 4.3) was used throughout. In contrast to the Juropa experiments, we concentrate on the weak scalability of the *symmetric-definite* generalized eigenproblem. Therefore ScaLAPACK's DC timings correspond to the following sequence of routines: `PDPOTRF`–`PDSYNGST`–`PDSYNTRD`–`PDSTEDC`–`PDORMTR`–`PDTRSM`. Accordingly, ScaLAPACK's MRRR corresponds to the same sequence of routines with `PDSTEDC` replaced by `PDSTEGR`. In both cases, a block size of 64 was found be nearly optimal and used in all experiments. As already explained in Section 4.1, we avoided the use of the routines `PDSYGST` and `PDSYTRD` for the reduction to standard and tridiagonal form, respectively.

For the EleMRRR timings we used Elemental (ver. 0.66), which integrates PMRRR (ver. 0.6). A block size of 96 was identified as nearly optimal and used for all experiments.[13]

4.2.1. Weak Scaling. On the left side, in Fig. 4.5 we present the EleMRRR timings computing all eigenpairs of $Ax = \lambda Bx$. While the size of test matrices range from 15,000 to 84,856, the number of nodes increases from 32 to 1,024 (128 to 4096 cores). On the right side, the execution time is broken down into the six stages of the GENEIG.

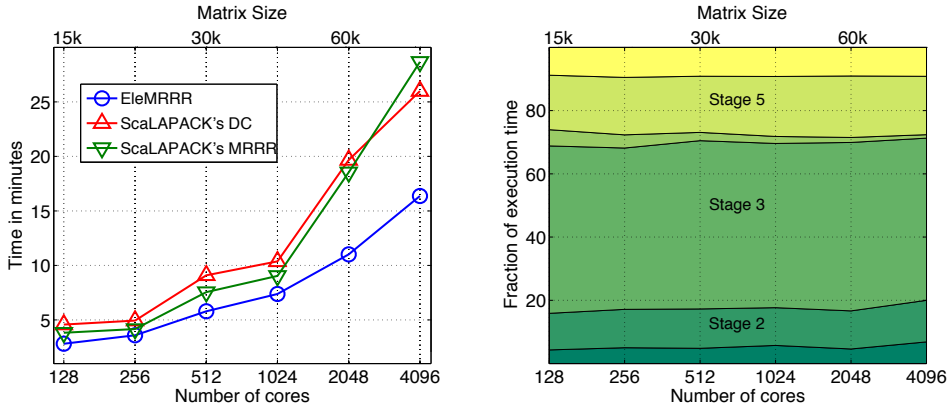

FIG. 4.5. *Weak scalability for the computation of all the eigenpairs of GENEIGs in the form* $Ax = \lambda Bx$. *Left: Total execution time. EleMRRR is the fastest solver accross the board. Right: Fraction of the execution time spent in the six stages, from bottom to top.*

Both graphs show a similar behavior we observed in the experiments on Juropa. In all experiments EleMRRR outperforms both the ScaLAPACK's solvers.

When comparing the EleMRRR results of Fig. 4.5 with ScaLAPACK's solver we observe the following: (a) Although Stage 4 of MRRR is usually not very time consuming, for the largest problem it is not negligible as it accounts for 27% of the total execution time. Similarly, in all experiments DC's tridiagonal solver accounts for about 20% of the execution time.

[13]The block size for matrix vector products, which did not have a significant influence on the performance, were fixed to 64 in all cases.

Four comments regarding the behavior of the solvers follow. (a) While Stage 4 of ScaLAPACK's MRRR and DC take up to 27% and 20%, respectively, PMRRR accounts for less than 5% of the total execution time. It is therefore negligible in all the experiments. (b) The timings corresponding to STDEIG – Stages 3 to 5 – cover between 75% and 71% of GENEIG's execution time. (c) The part roughly proportional to the fraction of desired eigenpairs – Stages 4 to 6 – makes up between 32% and 28% of both the GENEIG and STDEIG. (d) All stages scale equally well, such that no stage becomes the computational bottleneck.

5. Conclusions. To enable scientists to tackle large-scale dense eigenvalue problems, solvers must be fast and scalable: They have to make effective use todays' parallel computer systems that offer multiple types of parallelism.

Our study of dense large-scale generalized and standard eigenproblems was motivated by performance problems of commonly used routines in the ScaLAPACK library. In this paper we identify such problems and provide clear guidelines on how to circumvent them: By invoking suitable routines with the right settings, the users can assemble solvers faster than those included in ScaLAPACK.

The main contribution of the paper lays in the introduction of Elemental's dense eigensolvers and our tridiagonal eigensolver PMRRR. Together, they provide a set of routines – labeled EleMRRR – for large-scale eigenproblems. These solvers are part of the publicly available Elemental library and all make use of PMRRR. PMRRR is a parallel version of the MRRR algorithm for computing all or a subset of eigenpairs of tridiagonal matrices. It fully supports pure message-passing, pure multithreaded, and hybrid executions. Our experiments provide evidence that PMRRR is fast and the most scalable tridiagonal solver available.

In a thorough performance study on two state-of-the-art supercomputers, we compared EleMRRR with the solvers built within the ScaLAPACK framework according to our guidelines. For a modest amount of parallelism, the ScaLAPACK solvers obtain results comparable to EleMRRR, provided the fastest routines with best settings are invoked. In general, EleMRRR attains the best performance and obtains the best scalability of all solvers.

Acknowledgments. The paper would not have been possible without the work of Jack Poulson (UT Austin, Texas, USA). We are also grateful for the comments of Robert van de Geijn (UT Austin, Texas, USA) and Edoardo Di Napoli (Jülich Supercomputing Center, Germany). Futhermore, the authors would like to thank Inge Gutheil and Stefan Blügel (Research Center Jülich, Germany). Finally, sincere thanks are extended to the Jülich Supercomputing Center for granting access to Juropa and Jugene, allowing us to complete all the experiments.

REFERENCES

[1] B. Adlerborn, K. Dackland, and B. Kagström, *Parallel and Blocked Algorithms for Reduction of a Regular Matrix Pair to Hessenberg-Triangular and Generalized Schur Forms*, In Applied Parallel Computing, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2367 (2006), pp. 757–767.

[2] 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 Users' Guide*, SIAM, Philadelphia, PA, third ed., 1999.

[3] T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P.R. Willems, *Parallel Solution of Partial Symmetric Eigenvalue Problems from Electronic Structure Calculations*, Parallel Comput., (2011).

[4] J. Bennighof and R. Lehoucq, *An Automated Multilevel Substructuring Method for Eigenspace Computation in Linear Elastodynamics*, SIAM J. Sci. Comput., 25 (2004), pp. 2084–2106.

[5] P. Bientinesi, I. Dhillon, and R. van de Geijn, *A Parallel Eigensolver for Dense Symmetric Matrices Based on Multiple Relatively Robust Representations*, SIAM J. Sci. Comput., 27 (2005), pp. 43–66.

[6] C. Bischof, B. Lang, and X. Sun, *A Framework for Symmetric Band Reduction*, ACM Trans. Math. Software, 26 (2000), pp. 581–601.

[7] L. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK User's Guide*, SIAM, Philadelphia, PA, USA, 1997.

[8] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK: a Portable Linear Algebra Library for Distributed Memory Computers – Design Issues and Performance*, Computer Physics Communications, 97 (1996), pp. 1 – 15. High-Performance Computing in Science.

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

[10] G.-L. Dai, Z.-P. Liu, W.-N. Wang, J. Lu, and K.-N. Fan, *Oxidative Dehydrogenation of Ethane over V_2O_5 (001): A Periodic Density Functional Theory Study*, J. Phys. Chem. C, 112 (2008), pp. 3719–3725.

[11] J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*, SIAM, Philadelphia, PA, USA, 2000.

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

[13] I. Dhillon, *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*, PhD thesis, EECS Department, University of California, Berkeley, 1997.

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

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

[16] I. Dhillon, B. Parlett, and C. Vömel, *The Design and Implementation of the MRRR Algorithm*, ACM Trans. Math. Softw., 32 (2006), pp. 533–560.

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

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

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

[20] J. Gustafson, *Reevaluating Amdahl's Law*, Comm. ACM, 31 (1988), pp. 532–533.

[21] J. Hafner, *Materials Simulations using VASP–a Quantum Perspective to Materials Science*, Computer Physics Communications, 177 (2007), pp. 6 – 13. Proceedings of the Conference on Computational Physics 2006 - CCP 2006, Conference on Computational Physics 2006.

[22] B. Hendrickson, E. Jessup, and C. Smith, *Toward an Efficient Parallel Eigensolver for Dense Symmetric Matrices*, SIAM J. Sci. Comput., 20 (1999), pp. 1132–1154.

[23] T. Inaba and F. Sato, *Development of Parallel Density Functional Program using Distributed Matrix to Calculate All-Electron Canonical Wavefunction of Large Molecules*, J. of Comp. Chemistry, 28 (2007), pp. 984–995.

[24] I. Ipsen, *Computing An Eigenvector With Inverse Iteration*, SIAM Review, 39 (1997), pp. 254–291.

[25] P. Kent, *Computational Challenges of Large-Scale Long-Time First-Principles Molecular Dynamics*, J. Phys.: Conf. Ser., 125 (2008).
[26] M. Kim, *An Efficient Eigensolution Method and its Implementation for Large Structural Systems*, PhD thesis, Aerospace Engineering and Engineering Mechanics Department, The University of Texas at Austin, 2004.
[27] V. Kublanovskaya, *On some Algorithms for the Solution of the Complete Eigenvalue Problem*, Zh. Vych. Mat., 1 (1961), pp. 555–572.
[28] J. Lopéz-Blanco, J. Garzón, and P. Chacón, *iMod: Multipurpose Normal Mode Analysis in Internal Coordinates*, Bioinformatics, (2011).
[29] O. Marques, B. Parlett, and C. Vömel, *Computations of Eigenpair Subsets with the MRRR Algorithm*, Numerical Linear Algebra with Applications, 13 (2006), pp. 643–653.
[30] O. Marques, E. Riedy, and C. Vömel, *Benefits of IEEE-754 Features in Modern Symmetric Tridiagonal Eigensolvers*, SIAM J. Sci. Comput., 28 (2006), pp. 1613–1633.
[31] R. Martin and J. Wilkinson, *Reduction of the Symmetric Eigenproblem $Ax = \lambda Bx$; and Related Problems to Standard Form*, Numerische Mathematik, 11 (1968), pp. 99–110. 10.1007/BF02165306.
[32] C. Moler and G. Stewart, *An Algorithm for Generalized Matrix Eigenvalue Problems*, SIAM J. on Numerical Analysis, 10 (1973), pp. 241–256.
[33] B. Parlett and I. Dhillon, *Fernando's Solution to Wilkinson's Problem: an Application of Double Factorization*, Linear Algebra Appl., 267 (1996), pp. 247–279.
[34] B. Parlett and I. Dhillon, *Relatively Robust Representations of Symmetric Tridiagonals*, Linear Algebra Appl., 309 (2000), pp. 121 – 151.
[35] C. Persson and C. Ambrosch-Draxl, *A Full-Band FPLAPW+k·p-method for Solving the Kohn-Sham Equation*, Computer Physics Communications, 177 (2007), pp. 280 – 287.
[36] M. Petschow and P. Bientinesi, *MR^3-SMP: A Symmetric Tridiagonal Eigensolver for Multi-Core Architectures*, Parallel Computing, (2011). Submitted.
[37] J. Poulson, B. Marker, J. Hammond, N. Romero, and R. van de Geijn, *Elemental: A New Framework for Distributed Memory Dense Matrix Computations*, ACM Trans. Math. Software, (2011). Submitted.
[38] J. Poulson, R. van de Geijn, and J. Bennighof, *Parallel Algorithms for Reducing the Generalized Hermitian-Definite Eigenvalue Problem*, ACM Trans. Math. Software, (2011). Submitted.
[39] Y. Saad, *Numerical Methods for Large Eigenvalue Problems*, SIAM, Philadelphia, PA, USA, 2nd ed., 2011.
[40] M. Sears, K. Stanley, and G. Henry, *Application of a High Performance Parallel Eigensolver to Electronic Structure Calculations*, in Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM), Supercomputing '98, Washington, DC, USA, 1998, IEEE Computer Society, pp. 1–1.
[41] G. Shainer, T. Liu, B. Mayer, J. Dennis, B. Tweddell, N. Eicker, T. Lippert, A. Koehler, J. Hauke, and H. Falter, *HOMME and POPperf High Performance Applications: Optimizations for Scale*, White paper.
[42] K. Stanley, *Execution Time of Symmetric Eigensolvers*, PhD thesis, EECS Department, University of California, Berkeley, Dec 1997.
[43] L. Storchi, L. Belpassi, F. Tarantelli, A. Sgamellotti, and H. Quiney, *An Efficient Parallel All-Electron Four-Component Dirac–Kohn–Sham Program Using a Distributed Matrix Approach*, Journal of Chemical Theory and Computation, 6 (2010), pp. 384–394.
[44] F. Tisseur and J. Dongarra, *Parallelizing the Divide and Conquer Algorithm for the Symmetric Tridiagonal Eigenvalue Problem on Distributed Memory Architectures*, SIAM J. Sci. Comput, 20 (1998), pp. 2223–2236.
[45] S. Tomic, A. Sunderland, and I. Bush, *Parallel Multi-Band k·p Code for Electronic Structure of Zinc Blend Semiconductor Quantum Dots*, J. Mater. Chem., 16 (2006), pp. 1963–1972.
[46] R. van de Geijn, *Using PLAPACK: Parallel Linear Algebra Package*, The MIT Press, 1997.
[47] C. Vömel, *ScaLAPACK's MRRR algorithm*, ACM Trans. Math. Softw., 37 (2010), pp. 1:1–1:35.
[48] J. Wilkinson, *The Calculation of the Eigenvectors of Codiagonal Matrices*, Comp. J., 1 (1958), pp. 90–96.
[49] P. Willems, *On MR^3-type Algorithms for the Tridiagonal Symmetric Eigenproblem and Bidiagonal SVD*, PhD thesis, University of Wuppertal, 2010.