Prospectus for the Next LAPACK and ScaLAPACK Libraries

James Demmel[1], Jack Dongarra[23],
Beresford Parlett[1], William Kahan[1], Ming Gu[1], David Bindel[1],
Yozo Hida[1], Xiaoye Li[1], Osni Marques[1], E. Jason Riedy[1], Christof Voemel[1],
Julien Langou[2], Piotr Luszczek[2], Jakub Kurzak[2],
Alfredo Buttari[2], Julie Langou[2], Stanimire Tomov[2]

[1] University of California, Berkeley CA 94720, USA,
[2] University of Tennessee, Knoxville TN 37996, USA,
[3] Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA,

1 Introduction

Dense linear algebra (DLA) forms the core of many scientific computing applications. Consequently, there is continuous interest and demand for the development of increasingly *better* algorithms in the field. Here 'better' has a broad meaning, and includes improved reliability, accuracy, robustness, ease of use, and most importantly new or improved algorithms that would more efficiently use the available computational resources to speed up the computation. The rapidly evolving high end computing systems and the close dependence of DLA algorithms on the computational environment is what makes the field particularly dynamic.

A typical example of the importance and impact of this dependence is the development of LAPACK [1] (and later ScaLAPACK [2]) as a successor to the well known and formerly widely used LINPACK [3] and EISPACK [3] libraries. Both LINPACK and EISPACK were based, and their efficiency depended, on optimized Level 1 BLAS [4]. Hardware development trends though, and in particular an increasing Processor-to-Memory speed gap of approximately 50% per year, started to increasingly show the inefficiency of Level 1 BLAS *vs* Level 2 and 3 BLAS, which prompted efforts to reorganize DLA algorithms to use block matrix operations in their innermost loops. This formed LAPACK's design philosophy. Later ScaLAPACK extended the LAPACK library to run scalably on distributed memory parallel computers.

There are several current trends and associated challenges that influence the development of DLA software libraries. The main purpose of this work is to identify these trends, address the new challenges, and consequently outline a prospectus for new releases of the LAPACK and ScaLAPACK libraries.

2 Motivation

LAPACK and ScaLAPACK are widely used software libraries for numerical linear algebra. LAPACK provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The ScaLAPACK (Scalable LAPACK) library includes a subset of LAPACK routines redesigned for distributed memory MIMD parallel computers. There have been over sixty-eight million web hits at www.netlib.org (for the associated libraries LAPACK, ScaLAPACK, CLAPACK and LAPACK95). LAPACK and ScaLAPACK are used to solve leading edge science problems and they have been adopted by many vendors and software providers as the basis for their own libraries, including AMD, Apple (under Mac OS X), Cray, Fujitsu, HP, IBM, Intel, NEC, SGI, several Linux distributions (such as Debian), NAG, IMSL, the MathWorks (producers of MATLAB), InteractiveSupercomputing.com, and PGI. Future improvements in these libraries will therefore have a large impact on the user communities and their ability to advance scientific and technological work.

The ScaLAPACK and LAPACK development is mostly driven by

- algorithm research
- the result of the user/vendor survey
- the demands and opportunities of new architectures and programming languages
- the enthusiastic participation of the research community in developing and offering improved versions of existing Sca/LAPACK codes [5].

The user base is both large and diverse, ranging from users solving the largest and most challenging problems on leading edge architectures to the much larger class of users solving smaller problems of greater variety.

The rest of this paper is organized as follows.

Sec. 3 discusses challenges in making current algorithms run efficiently, scalably, and reliably on future architectures.

Sec. 4 discusses two kinds of improved algorithms: faster ones and more accurate ones. Since it is hard to improve both simultaneously, we choose to include a new faster algorithm if it is about as accurate as previous algorithms, and we include a new more accurate algorithm if it is at least about as fast as the previous algorithms.

Sec. 5 describes new linear algebra functionality that will be included in the next Sca/LAPACK release.

Sec. 6 describes our proposed software structure for Sca/LAPACK.

Sec. 7 describes a few initial performance results.

3 Challenges of Future Architectures

Parallel computing is becoming ubiqitous at all scales of computation: It is no longer just exemplified by the TOP 500 list of the fastest computers in the world, where over 400 have 513 or more processors, and over 100 have 1025 or more processors. In a few years typical laptops are predicted to have 64 cores per multicore processor chip, and up to 256 hardware threads per chip. So unless all algorithms (not just numerical linear algebra!) can exploit this parallelism, they will not only cease to speed up, but in fact slow down compared to machine peak.

Furthermore, the gap between processor speed and memory speed continues to grow exponentially: processor speeds are improving at 59% per year, main memory bandwidth at only 23%, and main memory latency at a mere 5.5% [6]. This means that an algorithm that is efficient today, because it does enough floating point operations per memory reference to mask slow memory speed, may not be efficient in the near future. The same story holds for parallelism, with communication network bandwidth improving at just 26%, and network latency unimproved since the Cray T3E in 1996 until recently.

The large scale target architectures of importance for LAPACK and ScaLAPACK include platforms like the Cray X1 at Oak Ridge National Laboratory, the Cray XT3 (Red Storm) at Sandia National Laboratory, the IBM Blue Gene/L system at Lawrence Livermore National Laboratory, a large Opteron cluster at Lawrence Berkeley National Laboratory (LBNL), a large Itanium-2 cluster at Pacific Northwest National Laboratory, the IBM SP3 at LBNL, and near term procurements underway at various DOE and NSF sites.

Longer term the High Productivity Computing Systems (HPCS) program [7] is supporting the construction of petascale computers. At the current time two vendors are designing hardware and software platforms that should scale to a petaflop by 2010: Cray's Cascade system (with the Chapel programming language), and IBM's PERCS system (with X10). Sun is also building a language called Fortress for petascale platforms. Other interesting platforms include IBM's Blue Planet [8], and vector extensions to IBM's Power systems called ViVA and ViVA-2.

As these examples illustrate, LAPACK and ScaLAPACK will have to run efficiently and correctly on a much wider array of platforms than in the past. It will be a challenge to map LAPACK's and ScaLAPACK's current software hierarchy of BLAS/BLACS/PBLAS/LAPACK/ScaLAPACK efficiently to all these platforms. For example, on a platform with multiple levels of parallelism (multicores, SMPs, distributed memory) would it be better to treat each SMP node as a ScaLAPACK process, calling BLAS which are themselves parallel, or should each processor within the SMP be mapped to a process, or something else? At a minimum the many tuning parameters (currently hard-coded within ILAENV) will have to be tuned for each platform; see sec. 6.

A more radical departure from current practice would be to make our algorithms asynchronous. Currently our algorithms are block synchronous, with phases of computation followed by communication with (implicit) barriers. But on networks than can overlap communication and computation, or on multithreaded shared memory machines, block synchrony can leave a significant fraction of the platform idle at any time. For example, the LINPACK benchmark version of LU decomposition exploits such asynchrony and can runs 2x faster than its block synchronous ScaLAPACK counterpart. See section 6.3.

Future platform are also likely to be heterogeneous, in performance and possibly floating point semantics. One example of heterogeneous performance is a cluster purchased over time, with both old, slow processors and new, fast processors. Varying user load can also cause performance heterogeneity. But there is even performance heterogeneity within a single processor. The best known examples are the x86/x86-64 family of processors with SSE and 3DNow! instruction sets, PowerPC family chips with Altivec extensions such as IBM Cell, scalar versus vector units on the Cray X1, X1E and XD1 platforms, and platforms with both GPUs and CPUs. In the case of Cell, single precision runs at 10x the speed of double precision. Speed differences like this motivate us to consider using rather different algorithms than currently in use, see sec.4.

Heterogeneities can also arise in floating point semantics. This is obvious in case of clusters of different platforms, but also within platforms. To illustrate one challenge imagine that one processor runs fastest when handling denormalized numbers according to the IEEE 754 floating point standard [9, 10], and another runs fastest when flushing them to zero. On such a platform sending a message containing a denormalized number from one processor to another can change its value (to zero) or even lead to a trap. Either way, correctness is a challenge, and not just for linear algebra.

4 Better Algorithms.

Three categories of routines are going to be addressed in Sca/LAPACK: (1) improved algorithms for functions in LAPACK, which also need to be put in ScaLAPACK (discussed here) (2) functions now in LAPACK but not ScaLAPACK (discussed in section 5), and (3) functions in neither LAPACK nor ScaLAPACK (also discussed in section 5).

There are a number of faster and/or more accurate algorithms that are going to be incorporated in Sca/LAPACK. The following is a list of each set of future improvements.

4.1 Algorithmic improvements for the solution of linear systems

1. The recent developments of extended precision arithmetic [11–13, 4] in the framework of the new BLAS standard allow the use of higher precision iterative refinement to improve computed solutions. Recently, it has been shown how to modify the classical algorithm of Wilkinson [14, 15] to compute not just an error bound measured by the infinity (or max) norm, but also a component-wise, relative error bound, i.e. a bound on the number of correct digits in each component. Both error bounds can be computed for a tiny $O(n^2)$ extra cost after the initial $O(n^3)$ factorization [16].
2. As mentioned in sec. 3, there can be a large speed difference between different floating point units on the same processor, with single precision running 10x faster than double precision on an IBM Cell, and 2x faster on the SSE unit than the x86 unit on some Intel platforms. We can exploit iterative

refinement to run faster in these situations. Suppose we want to solve $Ax = b$ where all the data types are double. The idea is to round A to single, perform its LU factorization in single, and then do iterative refinement with the computed solution $\hat{x}$ and residual $r = A\hat{x} - b$ in double, stopping when the residual is as small as it would be had the LU decomposition been done in double. This mean that the $O(n^3)$ work of LU decomposition will run at the higher speed of single, and only the extra $O(n^2)$ of refinement will run in double. This approach can get a factor of 2x speedup on a laptop with x86 and SSE units, and promises more on Cell. Of course it only converges if A's condition number is less than about $1/\sqrt{\epsilon}$, where ϵ is machine precision, but in this common case it is worthwhile. For algorithms beyond solving $Ax = b$ and least squares, such as eigenvalue problems, current iterative refinement algorithms increase the $O(n^3)$ work by a small constant factor. But if there is a factor of 10x to be gained by working in single, then they are still likely to be worthwhile.

3. Gustavson, Kågström and others have recently proposed a new set of *recursive data structures* for dense matrices [17–19]. These data structures represent a matrix as a collection of small rectangular blocks (chosen to fit inside the L1 cache), which are stored using one of several "space filling curve" orderings. The idea is that the data structure and associated recursive matrix algorithms are *cache oblivious* [20], that is they optimize cache locality without any explicit blocking such as conventionally done in LAPACK and ScaLAPACK, or any of the tuning parameters (beyond the L1 cache size).
   The reported benefits of these data structures and associated algorithms to which they apply are usually slightly higher peak performance on large matrices and a faster increase towards peak performance as the dimension grows. Sometimes, slightly modified, tuned BLAS are used for operations on matrices assumed to be in L1 cache. The biggest payoff by far is for factoring symmetric matrices stored in packed format, where the current LAPACK routines are limited to the performance of Level 2 BLAS, which do $O(1)$ flops per memory reference. Whereas the recursive algorithms can use the faster Level 3 BLAS, which do $O(n)$ flops per memory reference and can be optimized to hide slower memory bandwidth and latencies.
   The drawback of these algorithms is their use of a completely different and rather complicated data structure, which only a few expert users could be expected to use. That leaves the possibility of copying the input matrices in conventional column-major (or row-major) format into the recursive data structure. Furthermore, they are only of benefit for "one-sided factorizations" (LU, LDL^T, Cholesky, QR), but none of the "two-sided factorizations" needed for the eigenvalue decomposition (EVD) or SVD (there is a possibility they might be useful when no eigenvectors or singular vectors are desired).
   The factorization of symmetric packed matrices will be incorporated into LAPACK using the recursive data structures, copying the usual data structure in-place to the recursive data structure. The copying costs $O(n^2)$ contrast to the overall $O(n^3)$ operation count, so the asymptotic speeds should be the

same. Even if the recursive data structures will be used for other parts of LAPACK, the same column-major interface data structures will be kept for the purpose of ease of use.

4. Ashcraft, Grimes and Lewis [21] proposed a variation of Bunch-Kaufman factorization for solving symmetric indefinite systems $Ax = b$ by factoring $A = LDL^T$ with different pivoting. The current Bunch-Kaufman factorization is backward stable for the solution of $Ax = b$ but can produce unbounded L factors. Better pivoting provides better accuracy for applications requiring bounded L factors, like optimization and the construction of preconditioners [15, 22].
5. A Cholesky factorization with diagonal pivoting [23, 15] that avoids a breakdown if the matrix is nearly indefinite/rank-deficient is valuable for optimization problems (and has been requested by users) and is also useful for the high accuracy solution of the symmetric positive definite EVD, see below. For both this pivoting strategy and the one proposed above by Ashcraft/Grimes/Lewis, published results indicate that, on uniprocessors (LAPACK), the extra search required (compared to Bunch-Kaufman) has a small impact on performance. This may not be the case for distributed memory (ScaLAPACK) in which the extra searching and pivoting may involve nonnegligible communication costs.
6. Progress has been made in the development of new algorithms for computing or estimating the condition number of tridiagonal [24, 25] or triangular matrices [26]. These algorithms play an important role in obtaining error bounds in matrix factorizations, and the most promising algorithms should be evaluated and incorporated in the future release.

4.2 Algorithmic improvements for the solution of eigenvalue problems

Algorithmic improvements to the current LAPACK eigensolvers concern both accuracy and performance.

1. Braman, Byers, and Mathias proposed in their SIAM Linear Algebra Prize winning work [27, 28] an up to 10x faster Hessenberg QR-algorithm for the nonsymmetric EVD. This is the bottleneck of the overall nonsymmetric EVD, for which significant speedups should be expected. Byers recently spent a sabbatical with James Demmel where he did much of the software engineering required to convert his prototype into LAPACK format. An extension of this work with similar benefits will be extended to the QZ algorithm for Hessenberg-triangular pencils, with collaboration from Mehrmann. Similar techniques will be exploited to accelerate the routines for (block) companion matrices; see Sec. 5.2.
2. An early version of an algorithm based on Multiple Relatively Robust Representations (MRRR) [29–32] for the tridiagonal symmetric eigenvalue problem (STEGR) was incorporated into LAPACK version 3. This algorithm promised to replace the prior $O(n^3)$ QR algorithm (STEQR) or $\approx O(n^{2.3})$

divide & conquer (STEDC) algorithm with an $O(n^2)$ algorithm. It should have cost $O(nk)$ operations to compute the nk entries of k n-dimensional eigenvectors, the minimum possible work, in a highly parallel way. In fact, the algorithm in LAPACK v.3 did not cover all possible eigenvalue distributions and resorted to a slower and less accurate algorithm based on classical inverse iteration for "difficult" (highly clustered) eigenvalue distributions. The inventors of MRRR, Parlett and Dhillon, have continued to work on improving this algorithm. Very recently, Parlett and Vömel have proposed a solution for the last hurdle [33] and now pass the Sca/LAPACK tests for the most extreme examples of highly multiple eigenvalues. (These arose in some tests from very many large "glued Wilkinson matrices" constructed so that large numbers of mathematically distinct eigenvalues agreed to very high accuracy, much more than double precision. The proposed solution involves randomization, making small random perturbations to an intermediate representation of the matrix to force all eigenvalues to disagree in at least 1 or 2 bits.) Given the solution to this last hurdle, this algorithm may be propagated to all the variants of the symmetric EVD (eg., banded, generalized, packed, etc.) in LAPACK. A parallelized version of this algorithm will be available for the corresponding ScaLAPACK symmetric EVD routines. Currently ScaLAPACK only has parallel versions of the oldest, least efficient (or least accurate) LAPACK routines. This final MRRR algorithm requires some care at load balancing because the Multiple Representations used in MRRR represent subsets of the spectrum based on how clustered they are, which may or may not correspond to a good load balance. Initial work in this area is very promising [34].

3. The MRRR algorithm should in principle also be applied to the SVD, replacing the current $O(n^3)$ or $\approx O(n^{2.3})$ bidiagonal SVD algorithms with an $O(n^2)$ algorithm. Associated theory and a preliminary prototype implementation have been developed [35], but some potential obstacles to guaranteed stability remain [36].

4. There are three phases in the EVD (or SVD) of a dense or band matrix: (1) reduction to tridiagonal (or bidiagonal) form, (2) the subsequent tridiagonal EVD (or bidiagonal SVD), and (3) backtransforming the eigenvectors (or singular vectors) of the tridiagonal (or bidiagonal) to correspond to the input matrix. If many (or all) eigenvectors (or singular vectors) are desired, the bottleneck had been phase 2. But now the MRRR algorithm promises to make phase 2 cost just $O(n^2)$ in contrast to the $O(n^3)$ costs of phases 1 and 3. In particular, Howell and Fulton [37] recently devised a new variant of reduction to bidiagonal form for the SVD that has the potential to eliminate half the memory references by reordering the floating point operations (flops). Howell and Fulton fortunately discovered this algorithm during the deliberations of the recent BLAS standardization committee, because its implementation required new BLAS routines (routines GEMVT and GEMVER [4]) which were then added to the standard. These routines are called "Level 2.5 BLAS" because they do many more than $O(1)$ but fewer than $O(n)$ flops

per memory reference. Preliminary tests indicate speedups of up to nearly 2x.

5. For the SVD, when only left or only right singular vectors are desired, there are other variations on phase 1 to consider that reduce both floating point operations and memory references [38, 39]. Initial results indicate reduced operation counts by a ratio of up to .75, but at the possible cost of numerical stability for some singular vectors.
6. When few or no vectors are desired, the bottleneck shifts entirely to phase 1. Bischof and Lang [40] have proposed a Successive Band Reduction (SBR) algorithm that will asymptotically (for large dimension n) change most of the Level 2 BLAS operations in phase 1 to Level 3 BLAS operations (see the attached letter from Lang). They report speedups of almost 2.4x. This approach is not suitable when a large number of vectors are desired because the cost of phase 3 is much larger per vector. In other words, depending on how many vectors are desired, either the SBR approach will be used or the one-step reduction (the Howell/Fulton variant for the SVD and the current LAPACK code for the symmetric EVD). And if only left or only right singular vectors are desired, the algorithms described in bullet 5 might be used. This introduces a machine-dependent tuning parameter to choose the right algorithm; see tuning of this and other parameters in Sec. 6.2. It may also be possible to use Gustavson's recursive data structures to accelerate SBR.
7. Drmač and Veselić have made significant progress on the performance of the one-sided Jacobi algorithm for computing singular values with high relative accuracy [41, 42]. In contrast to the algorithms described above, their's can compute most or all of the significant digits in tiny singular values when these digits are determined accurately by the input data and when the above algorithms return only roundoff noise. The early version of this algorithm introduced by James Demmel in [43, 41] was quite slower than the conventional QR-iteration-based algorithms and much slower than the MRRR algorithms discussed above. But recent results reported by Drmač at [44] show that a combination of clever optimizations have finally led to an accurate algorithm that is *faster* than the original QR-iteration-based algorithm. Innovations include preprocessing by QR factorizations with pivoting, block application of Jacobi rotations, and early termination. Two immediate applications include the (full matrix) SVD and the symmetric positive-definite EVD, by first reducing to the SVD using the Cholesky-with-pivoting algorithm discussed earlier.
8. Analogous, high accuracy algorithms for the symmetric indefinite EVD have also been designed. One approach by Slapničar [45, 46] uses a J-symmetric Jacobi algorithm with hyperbolic rotations, and another one by Dopico/ Molera/ Moro [47] does an SVD, which "forgets" the signs of the eigenvalues and then reconstructs the signs. The latter can directly benefit by the Drmač/Veselič algorithm above.

5 Added Functionality

5.1 Putting more of LAPACK into ScaLAPACK.

Table 1 compares the available data types in the latest releases of LAPACK and ScaLAPACK. After the data type description, the prefixes used in the respective libraries are listed. A blank entry indicates that the corresponding type is not supported. The most important omissions in ScaLAPACK are as follows: (1) There is no support for packed storage of symmetric (SP,PP) or Hermitian (HP,PP) matrices, nor the triangular packed matrices (TP) resulting from their factorizations (using $\approx n^2/2$ instead of n^2 storage); these have been requested by users. The interesting question is what data structure to support. One possibility is recursive storage as discussed in Sec. 5.2 [48, 17–19]. Alternatively the packed storage may be partially expanded into a 2D array in order to apply Level 3 BLAS (GEMM) efficiently. Some preliminary ScaLAPACK prototypes support packed storage for the Cholesky factorization and the symmetric eigenvalue problem [49]. (2) ScaLAPACK only offers limited support of band matrix storage and does not specifically take advantage of symmetry or triangular form (SB,HB,TB). (3) ScaLAPACK does not support data types for the standard (HS) or generalized (HG, TG) nonsymmetric EVDs; see further below.

Table 2 compares the available functions in LAPACK and ScaLAPACK. The relevant user interfaces ('drivers')are listed by subject and acronyms are used for the software in the respective libraries. Table 2 also shows that,in the ScaLAPACK library, the implementation of some driver routines and their specialized computational routines are currently missing.

For inclusion in ScaLAPACK:

1. The solution of symmetric linear systems (SYSV), combined with the use of symmetric packed storage (SPSV), will be a significant improvement with respect to both memory and computational complexity over the currently available *LU* factorization. It has been requested by users and is expected to be used widely. In addition to solving systems it is used to compute the inertia (number of positive, zero and negative eigenvalues) of symmetric matrices.
2. EVD and SVD routines of all kinds (standard for one matrix and generalized for two matrices) are missing from ScaLAPACK. For SYEV, ScaLAPACK has `p_syev` (QR algorithm), `p_syevd` (divide and conquer), `p_syevx` (bisection and inverse iteration); a prototype code is available for MRRR (`p_syevr`). For SVD, ScaLAPACK has `p_gesvd` (QR algorithm). And for NEV, ScaLAPACK has `p_gehrd` (reduction to Hessenberg form), `p_lahqr` (reduction of a Hessenberg matrix to Schur form). This two routines enable users to get eigenvalues of a nonsymmetric matrix but not (easily) the eigenvectors. The MRRR algorithm is expected to be exploited for the SVD and symmetric EVD as are new algorithms of Braman/Byers/Mathias for the nonsymmetric EVD (see Sec. 4), work is under way to get the QZ algorithm for the nonsymmetric generalized eigenvalue problem.

	LAPACK	SCALAPACK
general band	GB	GB, DB
general (i.e., unsymmetric, in some cases rectangular)	GE	GE
general matrices, generalized problem	GG	GG
general tridiagonal	GT	DT
(complex) Hermitian band	HB	
(complex) Hermitian	HE	HE
upper Hessenberg matrix, generalized problem	HG	
(complex) Hermitian, packed storage	HP	
upper Hessenberg	HS	LAHQR only
(real) orthogonal, packed storage	OP	
(real) orthogonal	OR	OR
positive definite band	PB	PB
general positive definite	PO	PO
positive definite, packed storage	PP	
positive definite tridiagonal	PT	PT
(real) symmetric band	SB	
symmetric, packed storage	SP	
(real) symmetric tridiagonal	ST	ST
symmetric	SY	SY
triangular band	TB	
generalized problem, triangular	TG	
triangular, packed storage	TP	
triangular (or in some cases quasi-triangular)	TR	TR
trapezoidal	TZ	TZ
(complex) unitary	UN	UN
(complex) unitary, packed storage	UP	

Table 1. Data types supported in LAPACK and ScaLAPACK. A blank entry indicates that the corresponding format is not supported in ScaLAPACK.

3. LAPACK provides software for the linearly constrained (generalized) least squares problem, and users in the optimization community will benefit from a parallel version. In addition, algorithms for rank deficient, standard least squares problems based on the SVD are missing from ScaLAPACK; it may be that a completely different algorithm based on the MRRR algorithm (see Sec. 4) may be more suitable for parallelism instead of the divide & conquer (D&C) algorithm that is fastest for LAPACK.
4. Expert drivers that provide error bounds, or other more detailed structural information about eigenvalue problems, should be provided.

5.2 Extending current functionality.

This subsection outlines possible extensions of the functionalities available in LAPACK and ScaLAPACK. These extensions are mostly motivated by users but also by research progress.

	LAPACK	SCALAPACK
Linear Equations	GESV (LU)	PxGESV
	POSV (Cholesky)	PxPOSV
	SYSV (LDL^T)	missing
Least Squares (LS)	GELS (QR)	PxGELS
	GELSY (QR w/pivoting)	missing
	GELSS (SVD w/QR)	missing
	GELSD (SVD w/D&C)	missing
Generalized LS	GGLSE (GRQ)	missing
	GGGLM (GQR)	missing
Symmetric EVD	SYEV (QR)	PxSYEV
	SYEVD (D&C)	PxSYEVD
	SYEVR (RRR)	missing
Nonsymmetric EVD	GEES (HQR)	missing driver
	GEEV (HQR + vectors)	missing driver
SVD	GESVD (QR)	PxGESVD (missing complex C/Z)
	GESDD (D&C)	missing
Generalized Symmetric EVD	SYGV (inverse iteration)	PxSYGVX
	SYGVD (D&C)	missing
Generalized Nonsymmetric EVD	GGES (HQZ)	missing
	GGEV (HQZ + vectors)	missing
Generalized SVD	GGSVD (Jacobi)	missing

Table 2. LAPACK codes and the corresponding parallel version in ScaLAPACK. The underlying LAPACK algorithm is shown in parentheses. "Missing" means both drivers and computational routines are missing. "Missing driver" means that the underlying computational routines are present.

1. Several updating facilities are planned to be included in a new release. While updating matrix factorizations like Cholesky, LDL^T, LU, QR [50] have a well established theory and unblocked (i.e. non cache optimized) implementations exist, e.g. in LINPACK [3], the efficient update of the SVD is a current research topic [51]. Furthermore, divide & conquer based techniques are promising for a general framework of updating eigendecompositions of submatrices.
2. Semi-separable matrices are generalizations of the inverses of banded matrices, with the property that any rectangular submatrix lying strictly above or strictly below the diagonal has a rank bounded by a small constant. Recent research has focused on methods exploiting semiseparability, or being a sum of a banded matrix and a semiseparable matrix, for better efficiency [52, 53]. The development of such algorithms is being considered in a future release. Most exciting are the recent observations of Gu, Bini and others [54] that a companion matrix is banded plus semiseparable, and that this structure is preserved under QR iteration to find its eigenvalues. This observation let us accelerate the standard method used in MATLAB and other libraries for finding roots of polynomials from $O(n^3)$ to $O(n^2)$. An initial rough prototype

of this code becomes faster than the highly tuned LAPACK eigensolver for n between 100 and 200 and then becomes arbitrarily faster for larger n. While the current algorithm has been numerically stable on all examples tested so far, more work needs to be done to guarantee stability in all cases. The same technique should apply to finding eigenvalues of block companion matrices, i.e. matrix polynomials, yielding speedups proportional to the degree of the matrix polynomial.

3. Eigenvalue problems for matrix polynomials [55] are common in science and engineering. The most common case is the quadratic eigenvalue problem $(\lambda^2 M + \lambda D + K)x = 0$, where typically M is a mass matrix, D a damping matrix, K a stiffness matrix, λ a resonant frequency, and x a mode shape. The classical solution is to linearize this eigenproblem, asking instead for the eigenvalues of a system of twice the size: $\lambda \cdot \begin{bmatrix} 0 & I \\ M & D \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} I & 0 \\ 0 & K \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = 0$ where $y_2 = x$ and $y_1 = \lambda x$. But there are a number of ways to linearize, and some are better at preserving symmetries in the solution of the original problem or saving more time than others. There has been a great deal of recent work on picking the right linearization and subsequent algorithm for its EVD to preserve desired structures. In particular, for the general problem $\sum_{i=0}^{k} \lambda^i \cdot A_i \cdot x = 0$, the requested cases are symmetric ($A_i = A_i^T$, arising in mechanical vibrations without gyroscopic terms), its even ($A_i = (-1)^i A_i^T$) and odd ($A_i = (-1)^{i+1} A_i^T$) variations (used with gyroscopic terms), and palindromic ($A_i = A_{k-i}^T$, arising in discrete time periodic and continuous time control). Recent references include [56–65].

4. Matrix functions (square root, exponential, sign function) play an important role in the solution of differential equations in both science and engineering, and have been requested by users. Recent research progress has led to the development of several new algorithms [66–75] that could be included in a future release.

5. The eigenvalue and singular value decomposition of products and quotients of matrices play an important role in control theory. Such functionalities, incorporated from the software library SLICOT [76] are being considered, using the improved underlying EVD algorithms. Efficient solvers for Sylvester and Lyapunov equations that are also currently in SLICOT could be incorporated.

6. Multiple user requests concern the development of out-of-core versions of matrix factorizations. ScaLAPACK prototypes [49] are under development that implement out-of-core data management for the LU, QR, and Cholesky factorizations [77, 78]. Users have asked for two kinds of parallel I/O: to a single file from a sequential LAPACK program (possible with sequential I/O in the reference implementation), and to a single file from MPI-based parallel I/O in ScaLAPACK.

6 Software

6.1 Improving Ease of Use

"Ease of use" can be classified as follows: ease of programming (which includes the possibility to easily convert from serial to parallel, form LAPACK to ScaLAPACK and the possiblity to use high level interfaces), ease of obtaining predictable results in dynamic environments (for debugging and performance), and ease of installation (including performance tuning). Each will be discussed in turn.

There are tradeoffs involved in each of these subgoals. In particular, ultimate ease of programming, exemplified by typing $x = A\backslash b$ in order to solve $Ax = b$ (paying no attention to the data type, data structure, memory management or algorithm choice) requires an infrastructure and user interface best left to the builders of systems like MATLAB and may come at a significant performance and reliability penalty. In particular, many users now exercise, and want to continue to exercise, detailed control over data types, data structures, memory management and algorithm choice, to attain both peak performance and reliability (eg., not running out of memory unexpectedly). But some users also would like Sca/LAPACK to handle work space allocation automatically, make it possible to call Sca/LAPACK on a greater variety of user-defined data structures, and pick the best algorithm when there is a choice.

To accomodate these "ease of programming" requests as well as requests to make the Sca/LAPACK code accessible from other languages than Fortran, the following steps are considered:

1. Produce new F95 modules for the LAPACK drivers, for work-space allocation and algorithm selection.
2. Produce new F95 modules for the ScaLAPACK drivers, which convert, if necessary, the user input format (eg., a simple block row layout across processors) to the optimal one for ScaLAPACK (which may be a 2D block cyclic layout with block sizes that depend on the matrix size, algorithm and architecture). Allocate memory as needed.
3. Produce LAPACK and ScaLAPACK wrappers in other languages. Based on current user surveys, these languages will tentatively be C, C++, Python and MATLAB.

See section 6.2 for details on the software engineering approach to these tasks.

Ease of conversion from serial code (LAPACK) to parallel code (ScaLAPACK) is done by making the interfaces (at least at the driver level) as similar as possible. This includes expanding ScaLAPACK's functionality to include as much of LAPACK as possible (see section 5).

Another ease-of-programming request is improved documentation. The Sca/LAPACK websites are continously developed to enable ongoing user feedback and support and the websites employ tools like bugzilla to track reported bugs.

Obtaining predictable results in a dynamic environment is important for debugging (to get the same answer when the code is rerun), for reproducibility, auditability (for scientific or legal purposes), and for performance (so that runtimes do not vary widely and unpredictably).

First consider getting the same answer when the problem is rerun. To establish reasonable expectations, consider 3 cases: (1) Rerunning on different computers with different compilers. Reproducibility here is more than one can expect. (2) Rerunning on the same platform (or cluster) but with different data layouts or blocking parameters. Reproducibility here is also more than the user can expect, because roundoff errors will differ as, say, matrix products are computed in different orders. (3) Just typing "a.out" again with the same inputs. Reproducibility here should be the user goal, but it is not guaranteed because asynchronous communication can result in, for example, dot products of distributed vectors being computed in different orders with different roundoff errors. Using an existing switch in the BLACS to insist on the same order of communication (with an unavoidable performance penalty) will address this particular problem, but investigations are underway to know whether this is the only source of nonreproducibility (asynchrony in tuned SMP BLAS is another possible source, even for "sequential" LAPACK). As more fully asynchronous algorithms are explored to get higher performance (see section 4), this problem becomes more interesting. Reproducibility will obviously come with a performance penalty but is important for debugging of codes that call Sca/LAPACK and for situations where auditability is critical.

Now consider obtaining predictable performance, in the face of running on a cluster where the processor subset that actually performs the computation may be chosen dynamically, and have a dynamically varying load of other jobs. This difficult problem is discussed further in section 6.2.

Ease of installation, which may include platform-specific performance tuning, depends on the multiple modes of access of the Sca/LAPACK libraries: (1) Some users may use vendor-supplied libraries prebuilt (and pretuned) on their platforms, so their installation needs are already well addressed. (2) Some users may use netlib to download individual routines and the subroutines they call (but not the BLAS, which must be supplied separately). These users have decided to perform the installations on their own (perhaps for educational purposes). For these users, it can be made possible to select an architecture from a list, so that the downloaded code has parameter values that are quite likely to optimize the user's performance. (3) Other users may download the entire package from netlib and install and tune it on their machines. The use of autoconf and automatic performance tuning should be expected.

Different users will want different levels of installation effort since complete testing and performance tuning can take a long time. For performance tuning, a database of pretuned parameters for various computing platforms will be built, and if the user indicates that he is happy with the pretuned parameters, performance tuning can be sidedstepped. As discussed in section 6.2, there is a spectrum of tuning effort possible, depending on what fraction of peak performance one seeks and how long one is willing to take to tune. Similarly, testing the installation for correctness can be done at at least 4 levels (and differently for each part of the library): (1) No test cases need be run at all if the user does not want to. (2) A few small test cases can be run to detect a flawed installation but not failures on difficult numerical cases. (3) The current test set can be run, which is designed for a fairly thorough numerical testing, and in fact not infrequently indicates error bounds exceeding tight thresholds on some problems and platforms. Some of these test cases are known failure modes for certain algorithms. For example, inverse iteration is expected to sometimes fail to get orthogonal eigenvectors on sufficiently clustered eigenvalues, but it depends on the vagaries of roundoff. (4) More extensive "torture test" sets are used internally for development, which the most demanding users (or competing algorithm developers) should use for testing. If only for the sake of future algorithm developers, these extensive test sets should be easily available.

6.2 Improved Software Engineering

The following is a description of the Sca/LAPACK software engineering (SWE) approach. The main goals are to keep the substantial code base maintainable, testable and evolvable into the future as architectures and languages change. Maintaining compatibility with other software efforts and encouraging 3rd party contributions to the efforts of the Sca/LAPACK team are also goals [5].

These goals involve tradeoffs. One could explore starting "from scratch", using higher level ways to express the algorithms from which specific implementations could be generated. This approach yields high flexibility allowing the generation of code that is optimized for future computing platforms with different layers of parallelism, different memory hierarchies, different ratios of computation rate to bandwidth to latency, different programming languages and compilers, etc. Indeed, one can think of the problem as implementing the following *meta-program*:

```
(1) for all linear algebra problems
    (linear systems, eigenproblems, ...)
(2)   for all matrix types
      (general, symmetric, banded, ...)
(3)     for all data types
        (real, complex, single, double, higher precision)
(4)       for all machine architectures
          and communication topologies
(5)         for all programming interfaces

(6)           provide the best algorithm(s) available in terms of
              performance and accuracy (``algorithms'' is plural
              because sometimes no single one is always best)
```

The potential scope can appear quite large, requiring a judicious mixture of prioritization and automation. Indeed, there is prior work in automation [79], but so far this work has addressed only part of the range of algorithmic techniques

Sca/LAPACK needs (eg., not eigenproblems), it may not easily extend to more asynchronous algorithms and still needs to be coupled to automatic performance tuning techniques. Still, some phases of the meta-program are at least partly automatable now, namely steps (3) through (5) (see below).

Note that line (5) of the meta-program is "programming interfaces" not "programming languages," because the question of the best implementation language is separate from providing ways to call it (from multiple languages). Currently Sca/LAPACK is written in F77. Over the years, the Sca/LAPACK team and others have built on this to provide interfaces or versions in other languages: LAPACK95 [80] and LAPACK3E [81] for F95 (LAPACK3E providing a straightforward wrapper, and LAPACK95 using F95 arrays to simplify the interfaces at some memory and performance costs), CLAPACK in C [82] (translated, mostly automatically, using f2c [83]), LAPACK++ [84], TNT [85] in C++, and JLAPACK in Java [86] (translated using f2j).

First is the SWE development plan and then the SWE research plan.

1. the core of Sca/LAPACK will be maintained in Fortran, adopting those features of F95 that most improve ease-of-use and ease-of-development, but do not prevent the most demanding users from attaining the highest performance and reliable control over the run-time environment. Keeping Fortran is justified for cost and continuity reasons, as well as the fact that the most effective optimizing compilers still work best on Fortran (even when they share "back ends" with the C compiler, because of the added difficulty of discerning the absence of aliasing in C) [87].
   The F95 features adopted include recursion (to support new matrix data structures and associated algorithms discussed in section 4), modules (to support production of versions for different precisions, beyond single and double), and environmental enquiries (to replace xLAMCH), but not automatic workspace allocation (see the next bullet).
2. F95 versions of the Sca/LAPACK drivers will be provided (which will usually be wrappers) to improve ease-of-use, possibly at some performance and reliability costs. For example, automatically allocating workspace using assumed-size arrays (as in the more heavily used LAPACK95, as opposed to LAPACK3E), and performing algorithm selection when appropriate (based on performance models described below) may be done.
3. ScaLAPACK drivers will be provided that take a variety of parallel matrix layouts and automatically identify and convert to the optimal layout for the problem to be solved. Many users have requested accomodation of simpler or more general input formats, which may be quite different from the more complicated performance-optimized 2D block-cycle (and possibly recursive) layouts used internally. Using the performance models described below, these drivers will determine the optimal layout for the input problem, estimate the cost of solving "in-place" versus converting to the optimal layout, solving, and converting back, and choose the fastest solution. Separate layout conversion routines will be provided to help the user identify and convert to better layouts.

4. Wrappers for the Sca/LAPACK drivers in other languages will be provided, with interfaces that are "natural" for those languages, chosen based on importance and demand from the user survey. Currently this list includes C, Python and MATLAB.
   In particular, no native implementations in these languages will be provided, depending instead on interoperability of these languages with the F95 subset used in (1).
5. The new Sca/LAPACK routines will be converted to use the latest BLAS standard [88, 4, 12], which also provides new high precision functionality necessary for new routines required in section 4 [12, 16], systematically ensure thread-safety, and deprecate superseded routines.
6. Appropriate tools will be used (eg., autoconf, bugzilla, svn, automatic overnight build and test, etc.) to streamline installation and development and encourage third party contributions. Appropriate source control and module techniques will be used to minimize the number of versions of each code, reducing the number of versions in the cross product {real,complex} $\times$ {single, double, quad, ...} to one or two.
7. Performance tuning will be done systematically. Initial experiments are showing up to 10x speedups using different communication schemes that can be applied in the BLACS [89, 90]. In addition to tuning the BLAS [91, 92] and BLACS, there are over 1300 calls to the ILAENV routine in LAPACK, many of which return tuning parameters that have never been systematically tuned. Some of these parameters are block sizes for blocked algorithms, some are problem size thresholds for choosing between different algorithms, and yet others are numerical convergence thresholds. ScaLAPACK has yet more parameters associated with parallel data layouts and communication algorithms.
   As described in section 6.1, there are different levels of effort possible for performance tuning, and it may also be done at different times. For example, as part of the development process, a database of pretuned parameters and performance models will be built for various computing platforms, which will be good enough for many users [93]. Still, a tool is needed that at user-install time systematically searches a very large parameter space of tuning parameters to pick the best values, where users can "dial" the search effort from quick to exhaustive and then choose different levels of search effort for different routines depending on which are more important. Sophisticated data modeling techniques may be used to minimize search time [94].

Beyond these development activities, the following research tasks are being performed, which should influence and improve the development.

1. As discussed in section 3 emerging architectures offer parallelism at many different levels, from multicore chips to vector processors to SMPs to distributed memory machines to clusters, many of which will appear simultaneously in one computing system. Alongside these layers is the current software hierarchy in ScaLAPACK: BLAS, BLACS, PBLAS, LAPACK and ScaLAPACK. An investigation needs to be done to decide how to best map these

software layers to the hardware layers. For example, should a single BLAS call use all the parallelism available in an SMP and its multicore chips, or should the BLAS call just use the multicore chips with individual ScaLAPACK processes mapping to different SMP nodes? The number of possible combinations is large and growing. These mappings have to be explored to identify the most effective ones, using performance modeling to both measure progress (what fraction of peak is reached?) and limit the search space.

2. There is extensive research in other high performance programming languages funded by DOE, NSA and the DARPA HPCS program. UPC [95], Titanium [96], CAF [97], Fortress [98], X10 [99] and Cascade [100] are all languages under active development, and are being designed for high productivity SWE on high peformance machines. It is natural to ask if these are appropriate programming languages for ScaLAPACK, especially for the more asynchronous algorithms that may be fastest and for more fine-grained architectures like Blue Gene. Prototype selected ScaLAPACK codes will be provided in some of these languages to assess their utility.
3. Further automating the production of the Sca/LAPACK software is a worthy goal. One can envision expressing each algorithm once in a sufficiently high level language and then having a compiler (or source-to-source translator) automatically produce versions for any architecture or programming interface. Work in this direction includes [79]. The hope is that by limiting the scope to dense linear algebra, the translation problem will be so simplified that writing the translator pays off in being able to create the many needed versions of the code. But there are a number of open research problems that need to be solved for such an approach to work. First, it has been demonstrated for one-sided factorizations (eg LU, Cholesky and QR) but not on more complex algorithms like some two-sided ones needed by eigenproblems, including the successive-band-reduction (SBR) schemes discussed in section 4. Secondly, much if not most of Sca/LAPACK involves iterative algorithms for eigenproblems with different styles of parallelism, and it is not clear how to extend to these cases. Third, it is not clear how to best express the more asynchronous algorithms that can achieve the highest performance; this appears to involve either more sophisticated compiler analysis of more synchronous code (which this approach hoped to avoid) or a different way of expressing dependencies. In particular, one may want to use one of the programming languages mentioned above. Finally, it is not clear how to best exploit the multiple levels of parallelism discussed above, i.e. which should be handled by the high-level algorithm, the programming language, the compiler, the various library levels (BLAS, PBLAS) and so on. These are worthy research goals to solve before using this technique for development.
4. Performance tuning may be very time consuming if implemented in a brute force manner, by running and timing many algorithms for all parameter values in a large search space. A number of users have already expressed concern about installation time and difficulty, when this additional tuning may occur. Based on earlier experience [94], it is possible to use statistical

models to both limit the search space and more accurately determine the optimal parameters at run time.

5. In a dynamically changing cluster environment, a call to Sca/LAPACK might be run locally and sequentially, or remotely on one or more processors chosen at call-time. The use of performance models will be available with dynamically updated parameters (based on system load) to choose the right subset of the cluster to use, including the cost of moving the problem to a different subset and moving the answer back. This will help achieve performance predictability.

6.3 Multicore and Multithreading

With the number of cores on multicore chips expected to reach tens or potentially hundreds in a few years, efficient implementations of numerical libraries using shared memory programming models is of high interest. The current message passing paradigm used in ScaLAPACK and elsewhere introduces unnecessary memory overhead and memory copy operations, which degrade performance, along with the making it harder to schedule operations that could be done in parallel. Limiting the use of shared memory to fork-join parallelism (perhaps with OpenMP) or to its use within the BLAS does not address all these issues.

On the other hand, a number of goals can be achieved much more easily in shared memory than on distributed memory systems. The most striking simplification is no need for data partitioning, which can be replaced by work partitioning. Still, cache locality has to be preserved and, in this aspect, the locality of the two dimensional block cyclic (or perhaps recursive) layout cannot be eliminated.

There are several established programming models for shared memory systems appropriate for different levels of parallelism in the applications with the the client/server, work-crew, and pipeline being the most established ones. Since matrix factorizations are rich in data dependencies, the pipeline seems to be the most applicable model. Another advantage of pipelining is its match to hardware for streaming data processing, like the IBM Cell Broadband Engine. To achieve efficiency we must avoid pipeline stalls (also called bubbles) when data dependencies block execution. The next paragraph illustrates this approach for LU factorization.

LU and other matrix factorization have left-looking and right-looking formulations [101]. It has even been observed that transition between the two can be done by automatic code transformations [102], although more powerful methods than simple dependency analysis is necessary. It is known that lookahead can be used to improve performance, by performing panel factorizations in parallel with the update to the trailing matrix from the previous step of the algorithm [103]. The lookahead can be of arbitrary depth; this fact is exploiting by the LINPACK benchmark [104].

We observe that the right-looking and the left-looking formulations are two extremes of a spectrum of possible execution paths, with the lookahead providing a smooth transition between them. We regard the right-looking formulation as having zero lookahead, and the left-looking formulation as having maximum lookahead; see figure 1.

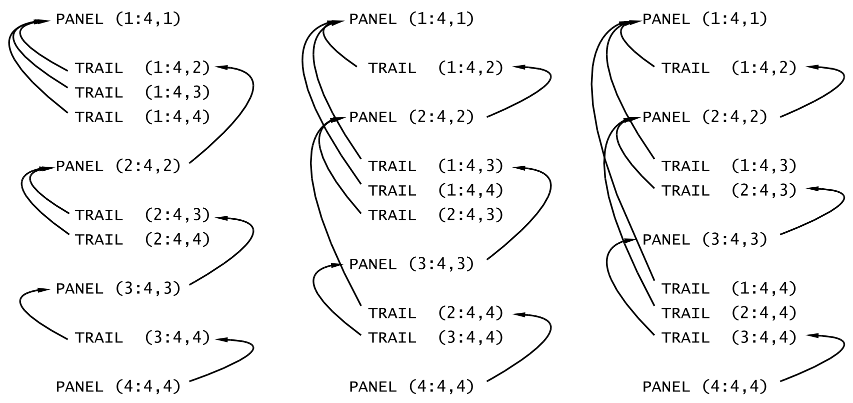

Fig. 1. Different possible factorization schemes: right-looking with no lookahead, right-looking with a lookahead of 1, left-looking (right-looking with maximum lookahead). Arrows show data dependencies.

The deeper the lookahead, the faster panels can be factorized and the more matrix multiplications are accumulated at the end of a factorization. Shallow lookaheads introduce pipeline stalls, or bubbles, at the end of a factorization. On the other hand, deep lookaheads introduce bubbles at the beginning of a factorization. Any fixed depth lookahead may stall the pipeline at both the beginning and at the end of execution.

Recent experiments show that pipeline stalls can be greatly reduced if unlimited lookahead is allowed and the lookahead panel factorizations are dynamically scheduled in such a way that their issues do not stall the pipeline. Dynamic work scheduling can easily and elegantly be implemented on shared memory, whereas it is a much more complex undertaking in a distributed memory arrangements. It is also worth observing that distributed memory implementations do not favor deep lookaheads due to storage overhead, which is not a problem in shared memory environments.

7 Performance

We give a few recent performance results for ScaLAPACK driver routines on recent architectures. We discuss strong scalability, i.e. we keep the problem size constant while increasing the number of processors.

Figure 2 gives the time to solution for a factoring a linear system of equations of size $n = 8,000$ on a cluster of dual processor 64 bit AMD Opterons interconnected with a Gigabit ethernet. As the number of processors increases from 1 to 64, the time decreases from 110 sec (3.1 GFlops) to 9 sec (37.0 GFlops[4].)

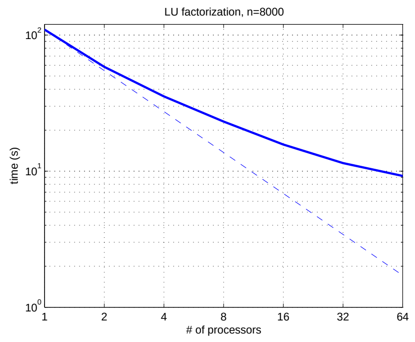

Fig. 2. Scalability of the LU factorization for ScaLAPACK (`pdgetrf`) for a matrix of size $n = 8,000$.

Figure 3 gives the time to solution to find the eigenvalue and eigenvectors of two symmetric matrices, one of size $n = 4,000$, the other of size $12,000$. The number of processors grows from 1 to 16 in the $n = 4000$ case and from 4 to 64 in the $n = 12000$ case. The machine used a cluster of dual processor 64 bit Intel Xeon EMTs interconnected with a Myrinet MX interconnect. The matrices are generated randomly using the same generator as in the Linpack Benchmark, and so have random eigenvalue distributions with no tight clusters.

[4] Thanks to Emmanuel Jeannot for sharing the result.

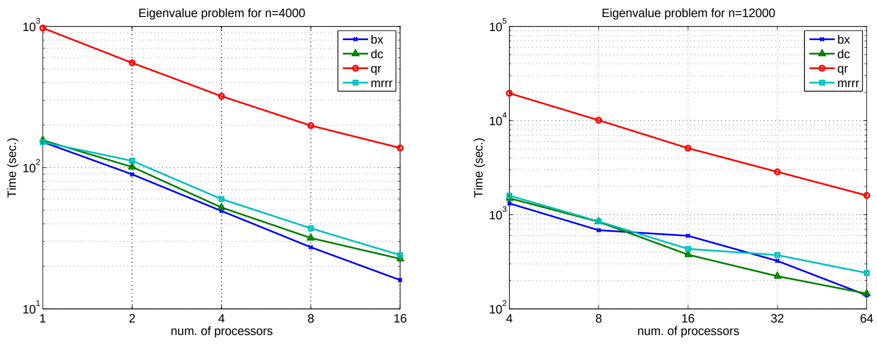

Fig. 3. Scalability of the Symmetric Eigenvalue Solver routines in ScaLAPACK (`pdsyev`) with a matrix of size 4,000 (*left*) and one of size 12,000 (*right*). Four different methods for the tridiagonal eigensolve have been tested: the BX (`pdsyevx`), QR (`pdsyev`), DC (`pdsyevd`) and MRRR (`pdsyevr`).

References

1. Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., Sorensen, D.: LAPACK Users' Guide, Release 1.0. SIAM, Philadelphia (1992) 235 pages.
2. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK Users' Guide. SIAM, Philadelphia (1997)
3. Dongarra, J., Bunch, J., Moler, C., Stewart, G.W.: LINPACK User's Guide. SIAM, Philadelphia, PA (1979)
4. Blackford, L.S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., Whaley, R.C.: An updated set of Basic Linear Algebra Subroutines (BLAS). ACM Trans. Math. Soft. **28**(2) (2002)
5. : (LAPACK Contributor Webpage) http://www.netlib.org/lapack-dev/contributions.html.
6. Graham, S., Snir, M., C. Patterson, e.: Getting up to Speed: The Future of Supercomputing. National Research Council (2005)
7. : High productivity computing systems (hpcs). (www.highproductivity.org)
8. Simon, H., Kramer, W., Saphir, W., Shalf, J., Bailey, D., Oliker, L., Banda, M., McCurdy, C.W., Hules, J., Canning, A., Day, M., Colella, P., Serafini, D., Wehner, M., Nugent, P.: Science-driven system architecture: A new process for leadership class computing. Journal of the Earth Simulator **2** (2005) 2–10
9. ANSI/IEEE New York: IEEE Standard for Binary Floating Point Arithmetic. Std 754-1985 edn. (1985)
10. : IEEE Standard for Binary Floating Point Arithmetic Revision. `grouper.ieee.org/groups/754` (2002)
11. Li, X.S., Demmel, J.W., Bailey, D.H., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kang, S.Y., Kapur, A., Martin, M.C., Thompson, B.J., Tung, T., Yoo, D.J.: Design, implementation and testing of extended and mixed precision BLAS. Trans. Math. Soft. **28**(2) (2002) 152–205

12. Bailey, D., Demmel, J., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kang, S., Kapur, A., Li, X., Martin, M., Thompson, B., Tung, T., Yoo, D.: Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Soft. **28**(2) (2002) 152–205
13. Blackford, L.S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., Whaley, R.C., Maany, Z., Krough, F., Corliss, G., Hu, C., Keafott, B., Walster, W., Wolff v. Gudenberg, J.: Basic Linear Algebra Subprograms Techical (BLAST) Forum Standard. Intern. J. High Performance Comput. **15**(3-4) (2001)
14. Wilkinson, J.H.: Rounding Errors in Algebraic Processes. Prentice Hall, Englewood Cliffs (1963)
15. Higham, N.J.: Accuracy and Stability of Numerical Algorithms. 2nd edn. SIAM, Philadelphia, PA (2002)
16. Demmel, J., Hida, Y., Kahan, W., Li, X.S., Mokherjee, S., Riedy, E.J.: Error bounds from extra precise iterative refinement (2005)
17. Gunnels, J.A., Gustavson, F.G.: Representing a symmetric or triangular matrix as two rectangular matrices. Poster presentation. SIAM Conference on Parallel Processing, San Francisco (2004)
18. Gustavson, F.: Recursion leads to automatic variable blocking for dense linear-algebra algorithms. IBM J. of Res. and Dev. **41**(6) (1997) www.research.ibm.com/journal/rd/416/gustavson.html.
19. Elmroth, E., Gustavson, F., Jonsson, I., Kågström, B.: Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Review **46**(1) (2004) 3–45
20. et al, L.A.: Cache-oblivious and cache-aware algorithms. Schloss Dagstuhl International Conference, www.dagstuhl.de/04301 (2004)
21. Ashcraft, C., Grimes, R.G., Lewis, J.G.: Accurate symmetric indefinite linear equation solvers. SIAM J. Mat. Anal. Appl. **20(2)** (1998) 513–561
22. Cheng, S.H., Higham, N.: A modified Cholesky algorithm based on a symmetric indefinite factorization. SIAM J. Mat. Anal. Appl. **19**(4) (1998) 1097–1110
23. Higham, N.J.: Analysis of the Cholesky decomposition of a semi-definite matrix. In Cox, M.G., Hammarling, S., eds.: Reliable Numerical Computation. Clarendon Press, Oxford (1990) 161–186
24. Dhillon, I.S.: Reliable computation of the condition number of a tridiagonal matrix in O(n) time. SIAM J. Mat. Anal. Appl. **19(3)** (1998) 776–796
25. Hargreaves, G.I.: Computing the condition number of tridiagonal and diagonal-plus-semiseparable matrices in linear time. Technical Report submitted, Department of Mathematics, University of Manchester, Manchester, England (2004)
26. Duff, I.S., Vömel, C.: Incremental Norm Estimation for Dense and Sparse Matrices. BIT **42(2)** (2002) 300–322
27. Braman, K., Byers, R., Mathias, R.: The multishift QR algorithm. Part I: Maintaining well-focused shifts and Level 3 performance. SIAM J. Mat. Anal. Appl. **23(4)** (2001) 929–947
28. Braman, K., Byers, R., Mathias, R.: The multishift QR algorithm. Part II: Aggressive early deflation. SIAM J. Mat. Anal. Appl. **23(4)** (2001) 948–973
29. Parlett, B.N., Dhillon, I.S.: Fernando's solution to Wilkinson's problem: an application of double factorization. Lin. Alg. Appl. **267** (1997) 247–279
30. Parlett, B.N., Dhillon, I.S.: Relatively robust representations of symmetric tridiagonals. Lin. Alg. Appl. **309**(1-3) (2000) 121–151
31. Parlett, B.N., Dhillon, I.S.: Orthogonal eigenvectors and relative gaps. SIAM J. Mat. Anal. Appl. **25(3)** (2004) 858–899

32. Parlett, B.N., Dhillon, I.S.: Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices (2004)
33. Parlett, B.N., Vömel, C.: Tight clusters of glued matrices and the shortcomings of computing orthogonal eigenvectors by multiple relatively robust representations. University of California, Berkeley (2004) In preparation.
34. Bientinisi, P., Dhillon, I.S., van de Geijn, R.: A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations. Technial Report TR-03-26, Computer Science Dept., University of Texas (2003)
35. Grosser, B.: Ein paralleler und hochgenauer O(n^2) Algorithmus für die bidiagonale Singulärwertzerlegung. PhD thesis, University of Wuppertal, Wuppertal, Germany (2001)
36. Willems, P. personal communication (2006)
37. Fulton, C., Howell, G., Demmel, J., Hammarling, S.: Cache-efficient bidiagonalization using BLAS 2.5 operators. 28 pages, in progress (2004)
38. Barlow, J., Bosner, N., Drmač, Z.: A new stable bidiagonal reduction algorithm. `www.cse.psu.edu/~barlow/fastbidiag3.ps` (2004)
39. Ralha, R.: One-sided reduction to bidiagonal form. Lin. Alg. Appl. **358** (2003) 219–238
40. Bischof, C.H., Lang, B., Sun, X.: A framework for symmetric band reduction. Trans. Math. Soft. **26**(4) (2000) 581–601
41. Demmel, J., Gu, M., Eisenstat, S., Slapničar, I., Veselić, K., Drmač, Z.: Computing the singular value decomposition with high relative accuracy. Lin. Alg. Appl. **299**(1–3) (1999) 21–80
42. Drmač, Z., Veselić, K.: New fast and accurate Jacobi SVD algorithm. Technical report, Dept. of Mathematics, University of Zagreb (2004)
43. Demmel, J., Veselić, K.: Jacobi's method is more accurate than QR. SIAM J. Mat. Anal. Appl. **13**(4) (1992) 1204–1246
44. : Fifth International Workshop on Accurate Solution of Eigenvalue Problems. www.fernuni-hagen.de/MATHPHYS/iwasep5 (2004)
45. Slapničar, I., Truhar, N.: Relative perturbation theory for hyperbolic singular value problem. Lin. Alg. Appl. **358** (2002) 367–386
46. Slapničar, I.: Highly accurate symmetric eigenvalue decomposition and hyperbolic SVD. Lin. Alg. Appl. **358** (2002) 387–424
47. Dopico, F.M., Molera, J.M., Moro, J.: An orthogonal high relative accuracy algorithm for the symmetric eigenproblem. SIAM. J. Mat. Anal. Appl. **25**(2) (2003) 301–351
48. Andersen, B.S., Wazniewski, J., G., G.F.: A recursive formulation of Cholesky factorization of a matrix in packed storage. Trans. Math. Soft. **27**(2) (2001) 214–244
49. Blackford, L.S., Choi, J., Cleary, A., Demmel, J., Dhillon, I., Dongarra, J.J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D.W., Whaley, R.C.: Scalapack prototype software. Netlib, Oak Ridge National Laboratory (1997)
50. Golub, G., Van Loan, C.: Matrix Computations. 3rd edn. Johns Hopkins University Press, Baltimore, MD (1996)
51. Gu, M., Eisenstat, S.C.: (A stable and fast algorithm for updating the singular value decomposition) in preparation.
52. Chandrasekaran, S., Gu, M.: Fast and stable algorithms for banded plus semiseparable systems of linear equations. SIAM J. Mat. Anal. Appl. **25(2)** (2003) 373–384

53. Vandebril, R., Van Barel, M., Mastronardi, M.: An implicit QR algorithm for semiseparable matrices to compute the eigendecomposition of symmetric matrices. Report TW 367, Department of Computer Science, K.U.Leuven, Leuven, Belgium (2003)
54. Bini, D., Eidelman, Y., Gemignani, L., Gohberg, I.: Fast QR algorithms for Hessenberg matrices which are rank-1 perturbations of unitary matrices. Dept. of Mathematics report 1587, University of Pisa, Italy (2005) `www.dm.unipi.it/~gemignani/papers/begg.ps`.
55. Gohberg, I., Lancaster, P., Rodman, L.: Matrix Polynomials. Academic Press, New York (1982)
56. Apel, T., Mehrmann, V., Watkins, D.: Structured eigenvalue methods for the computation of corner singularities in 3D anisotropic elastic structures. Preprint 01–25, SFB393, Sonderforschungsbereich 393, Fakultät für Mathematik, TU Chemnitz (2001)
57. Apel, T., Mehrmann, V., Watkins, D.: Structured eigenvalue methods for the computation of corner singularities in 3D anisotropic elastic structures. Comp. Meth. App. Mech. Eng. **191** (2002) 4459–4473
58. Apel, T., Mehrmann, V., Watkins, D.: Numerical solution of large scale structured polynomial eigenvalue problems. In: Foundations of Computational Mathematics, Springer Verlag (2003)
59. Meerbergen, K.: Locking and restarting quadratic eigenvalue solvers. Report RAL-TR-1999-011, CLRC, Rutherford Appleton Laboratory, Dept. of Comp. and Inf., Atlas Centre, Oxon OX11 0QX, GB (1999)
60. Mehrmann, V., Watkins, D.: Structure-preserving methods for computing eigenpairs of large sparse skew-Hamiltoninan/Hamiltonian pencils. SIAM J. Sci. Comput. **22** (2001) 1905–1925
61. Mehrmann, V., Watkins, D.: Polynomial eigenvalue problems with hamiltonian structure. Elec. Trans. Num. Anal. **13** (2002) 106–113
62. Tisseur, F.: Backward error analysis of polynomial eigenvalue problems. Lin. Alg. App. **309** (2000) 339–361
63. Tisseur, F., Meerbergen, K.: A survey of the quadratic eigenvalue problem. SIAM Review **43** (2001) 234–286
64. Triebsch, F.: Eigenwertalgorithmen für symmetrische λ-Matrizen. PhD thesis, Fakultät für Mathematik, Technische Universität Chemnitz (1995)
65. Benner, P., Byers, R., Mehrmann, V., Xu, H.: Numerical computation of deflating subspaces of embedded Hamiltonian pencils. Technical Report SFB393/99-15, Fakultät für Mathematik, TU Chemnitz, 09107 Chemnitz, FRG (1999)
66. Davies, P., Higham, N.J.: A Schur-Parlett algorithm for computing matrix functions. SIAM J. Mat. Anal. Appl. **25(2)** (2003) 464–485
67. Higham, N.J., Smith, M.I.: Computing the matrix cosine. Numerical Algorithms **34** (2003) 13–26
68. Smith, M.I.: A Schur algorithm for computing matrix pth roots. SIAM J. Mat. Anal. Appl. **24(4)** (2003) 971–989
69. Parks, M.: A study of algorithms to compute the matrix exponential. PhD thesis, University of California, Berkeley, California (1994)
70. Byers, R.: Numerical stability and instability in matrix sign function based algorithms. In Byrnes, C., Lindquist, A., eds.: Computational and Combinatorial Methods in Systems Theory, North-Holland (1986) 185–200
71. Byers, R.: Solving the algebraic Riccati equation with the matrix sign function. Lin. Alg. Appl. **85** (1987) 267–279

72. Byers, R., He, C., Mehrmann, V.: The matrix sign function method and the computation of invariant subspaces. preprint (1994)
73. Kenney, C., Laub, A.: Rational iteration methods for the matrix sign function. SIAM J. Mat. Anal. Appl. **21** (1991) 487–494
74. Kenney, C., Laub, A.: On scaling Newton's method for polar decomposition and the matrix sign function. SIAM J. Mat. Anal. Appl. **13**(3) (1992) 688–706
75. Bai, Z., Demmel, J., Gu, M.: Inverse free parallel spectral divide and conquer algorithms for nonsymmetric eigenproblems. Num. Math. **76** (1997) 279–308 UC Berkeley CS Division Report UCB//CSD-94-793, Feb 94.
76. Benner, P., Mehrmann, V., Sima, V., Van Huffel, S., Varga, A.: SLICOT - a subroutine library in systems and control theory. Applied and Computational Control, Signals, and Circuits **1** (1999) 499–539
77. Dongarra, J., Hammarling, S., Walker, D.: Key concepts for parallel out-of-core LU factorization. Computer Science Dept. Technical Report CS-96-324, University of Tennessee, Knoxville, TN (1996) www.netlib.org/lapack/lawns/lawn110.ps.
78. Dongarra, J., D'Azevedo, E.: The design and implmentation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines. Computer Science Dept. Technical Report CS-97-347, University of Tennessee, Knoxville, TN (1997) www.netlib.org/lapack/lawns/lawn118.ps.
79. Gunnels, J.A., Gustavson, F.G., Henry, G.M., van de Geijn, R.A.: FLAME: Formal Linear Algebra Methods Environment. ACM Transactions on Mathematical Software **27**(4) (2001) 422–455
80. Barker, V., Blackford, S., Dongarra, J., Du Croz, J., Hammarling, S., Marinova, M., Wasniewski, J., Yalamov, P.: LAPACK95 Users' Guie. SIAM (2001) www.netlib.org/lapack95.
81. Anderson, E.: LAPACK3E. www.netlib.org/lapack3e (2003)
82. : (CLAPACK: LAPACK in C) http://www.netlib.org/clapack/.
83. : (f2c: Fortran-to-C translator) http://www.netlib.org/f2c.
84. Dongarra, J., Pozo, R., Walker, D.: Lapack++: A design overview of ovject-oriented extensions for high performance linear algebra. In: Supercomputing 93, IEEE (1993) math.nist.gov/lapack++.
85. : (TNT: Template Numerical Toolkit) http://math.nist.gov/tnt.
86. : (JLAPACK: LAPACK in Java) http://icl.cs.utk.edu/f2j.
87. Coarfa, C., Dotsenko, Y., Mellor-Crummey, J., Chavarria-Miranda, D., Contonnet, F., El-Ghazawi, T., Mohanti, A., Yao, Y.: An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In: Proc. 10th ACM SIGPLAN Symp. on Principles and Practice and Parallel Programming (PPoPP 2005). (2005) www.hipersoft.rice.edu/caf/publications/index.html.
88. Blackford, S., Corliss, G., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Hu, C., Kahan, W., Kaufman, L., Kearfott, B., Krogh, F., Li, X., Maany, Z., Petitet, A., Pozo, R., Remington, K., Walster, W., Whaley, C., Wolff v. Gudenberg, J., Lumsdaine, A.: Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard. Intern. J. High Performance Comput. **15**(3–4) (2001) 305 pages, also available at `www.netlib.org/blas/blast-forum/`.
89. Nishtala, R., Chakrabarti, K., Patel, N., Sanghavi, K., Demmel, J., Yelick, K., Brewer, E.: Automatic tuning of collective communications in MPI. poster at SIAM Conf. on Parallel Proc., San Francisco, `www.cs.berkeley.edu/~rajeshn/poster_draft_6.ppt` (2004)

90. Vadhiyar, S.S., Fagg, G.E., Dongarra, J.: Towards an accurate model for collective communications. Intern. J. High Perf. Comp. Appl., special issue on Performance Tuning **18**(1) (2004) 159–167
91. Whaley, R.C., Petitet, A., Dongarra, J.: Automated empirical optimization of software and the ATLAS project. Parallel Computing **27**(1–2) (2001) 3–25
92. Whaley, R.C., Dongarra, J.: (The ATLAS WWW home page) `http://www.netlib.org/atlas/`.
93. : (OSKI: Optimized Sparse Kernel Interface) http://bebop.cs.berkeley.edu/oski/.
94. Vuduc, R., Demmel, J., Bilmes, J.: Statistical models for automatic performance tuning. In: Intern. Conf. Comput. Science. (2001)
95. Cantonnet, F., Yao, Y., Zahran, M., El-Ghazawi, T.: Productivity analysis of the UPC language. In: IPDPS 2004 PMEO workshop. (2004) www.gwu.edu/ upc/publications/productivity.pdf.
96. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performnace Java dialect. Concurrency: Practice and Experience **10** (1998) 825–836
97. Numrich, R., Reid, J.: Co-array Fortran for parallel programming. Fortran Forum, 17 (1998)
98. Allen, S.e.a.: The Fortress language specification, version 0.707. (research.sun.com/projects/plrg/fortress0707.pdf)
99. Saraswat, V.: Report on the experimental language X10, v0.41. IBM Research technical report (2005)
100. Callahan, D., Chamberlain, B., Zima, H.: The Cascade high-productivity language. In: 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004), IEEE Computer Society (2004) 52–60 www.gwu.edu/ upc/publications/productivity.pdf.
101. Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.A.: Numerical Linear Algebra for High-Performance Computers. SIAM, Philadelphia, PA (1998)
102. Menon, V., Pingali, K.: Look left, look right, look left again: An application of fractal symbolic analysis to linear algebra code restructuring. Int. J. Parallel Comput. **32**(6) (2004) 501–523
103. Strazdins, P.E.: A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Int. J. Parallel Distrib. Systems Networks **4**(1) (2001) 26–35
104. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK Benchmark: past, present and future. Concurrency Computat.: Pract. Exper. **15** (2003) 803–820