

D 3 Numerical Linear Algebra

I. Gutheil

Zentralinstitut für Angewandte Mathematik

Forschungszentrum Jülich GmbH

Contents

1	Systems of Linear Equations	2
1.1	Direct Linear System Solvers	2
1.2	Iterative Linear System Solvers	2
2	Eigensystems	3
2.1	Direct eigensolvers	3
2.2	Iterative eigensolvers	5
3	Libraries	6
3.1	LAPACK, Linear Algebra PACKage	6
3.2	ScaLAPACK, Scalable Linear Algebra PACKage	6
3.3	PETSc, Portable, Extensible Toolkit for Scientific Computation	9
3.4	ARPACK, PARPACK, Parallel ARnoldi PACKage	10
4	Final Remarks	10

Two of the most important linear algebra problems in scientific applications are the solution of linear equation systems and the solution of eigenvalue problems. Depending on the problem size and other properties of the system matrix several methods for the solution are available. We shall present some methods for the solution of both problems together with their implementations as numerical libraries for modern supercomputers.

1 Systems of Linear Equations

For the solution of linear equation systems

$$A\vec{x} = \vec{b}, \quad A \in \mathbb{R}^{nn}, \quad \vec{b}, \vec{x} \in \mathbb{R}^n \quad (1)$$

there are two different approaches, direct solvers and iterative solvers.

1.1 Direct Linear System Solvers

In direct solvers the system matrix A is decomposed into two triangular matrices

$$A = LU, \quad A, L, U \in \mathbb{R}^{nn}, \quad L \text{ lower and } U \text{ upper triangular matrix} \quad (2)$$

which gives

$$A\vec{x} = LU\vec{x} = L(U\vec{x}) =: L\vec{y} = \vec{b}. \quad (3)$$

This method is well known as LU decomposition method or Gaussian elimination. The resulting triangular equation systems $L\vec{y} = \vec{b}$ and $U\vec{x} = \vec{y}$ are then solved by forward and backward substitution.

If the system matrix A is symmetric positive definite, the upper triangular factor is the transpose of the lower triangular factor and the decomposition is called Cholesky decomposition.

$$A\vec{x} = LL^T\vec{x} = L(L^T\vec{x}) =: L\vec{y} = \vec{b}. \quad (4)$$

It takes only half as many operations as LU decomposition.

Direct linear system solvers theoretically deliver the exact solution. The decomposition of a matrix is an $O(n^3)$ computation whereas the forward- and backward-substitution are $O(n^2)$ computations. A major advantage of direct solvers is the fact that the decomposition – the most time consuming part of the computation – has to be done only once if the equation has to be solved with several right hand sides and the same system matrix during an algorithm.

The main disadvantage of direct solvers is the occurrence of so-called fill-ins during LU decomposition which lead to full lower and upper triangular matrices L and U even if A is sparse. A variant of the Cholesky and LU decomposition algorithm, the so-called multi-frontal method can preserve some sparsity during the decomposition phase [1][2].

1.2 Iterative Linear System Solvers

For the solution of partial differential equations iterative solvers are sometimes preferred to direct solvers.

The basic iterative solvers are the (total-step) and the (single-step) iteration [3]. They are both based on an additive matrix decomposition

$$A = D - L - U, \quad A, D, L, U \in \mathbb{R}^{nn}, \quad (5)$$

D diagonal matrix, L lower and U upper triangular matrix with 0 diagonal.
It follows that

$$\begin{aligned} A\vec{x} = \vec{b} &\Leftrightarrow (D - L - U)\vec{x} = \vec{b} \Leftrightarrow D\vec{x} = \vec{b} + (L + U)\vec{x} \\ &\Leftrightarrow \vec{x} = D^{-1}\vec{b} + D^{-1}(L + U)\vec{x}. \end{aligned} \quad (6)$$

If D is nonsingular the iteration rule

$$\vec{x}^{(k+1)} = D^{-1}\vec{b} + D^{-1}(L + U)\vec{x}^{(k)}, \quad k = 0, 1, \dots \quad (7)$$

with some starting vector $\vec{x}^{(0)}$ is the Jacobi iteration. The name total-step iteration comes from the fact that in each iteration only values from the last iteration are used. In the Gauß-Seidel iteration values from the last and from the current iteration are used where available.

In all iterative methods the solution is found by a fixed point iteration with $O(n^2)$ operations per iteration if the matrix is full. (The operations come from the matrix-vector multiplication $(L + U)\vec{x}$).

The sparsity of a matrix which normally is not preserved using direct solvers can be exploited by iterative methods because the number of operations for the matrix-vector product is proportional to the number of non-zero elements of the matrix.

The simple iterative methods are easy to understand but not suited for very large systems.

For symmetric positive definite matrices there are also several variants of the conjugate gradient algorithm [4].

Another method suited for very large problem sizes, the multi-grid method, will be presented by B. Steffen in chapter D.5 [5].

2 Eigensystems

For the solution of the real symmetric or complex hermitian eigenvalue problem

$$A\vec{x} = \lambda\vec{x}, \quad A \in \mathbb{R}^{nn}, \quad \vec{x} \in \mathbb{R}^n, \quad \lambda \in \mathbb{R} \quad (8)$$

there are several methods most of which use a three-stage algorithm where the matrix A first is reduced to tridiagonal form, then the eigenvalue problem for the tridiagonal matrix is solved and finally eigenvectors of the tridiagonal matrix are back transformed to the eigenvectors of the original matrix [6]. These methods are called direct solvers whereas solvers which do iterations with the original matrix to get the eigenvalues and eigenvectors are called iterative solvers.

Like the direct linear system solvers, the reduction based solvers do not preserve general sparsity of the system matrix. They require the storage for at least a full upper or lower triangular matrix in the symmetric case. Thus they are best suited for full matrices.

There are also reduction based solvers for the non-symmetric eigenvalue problem. They will not be presented here.

2.1 Direct eigensolvers

The so-called direct solvers all use the fact that eigenvalues are preserved under similarity transformations and that the eigenvalues of a symmetric tridiagonal matrix are more easily computed

than those of a full symmetric matrix. Most implementations in libraries use Householder transformations to get

$$A = QTQ^T \text{ with } QQ^T = Id \text{ (Identity matrix), } T \text{ tridiagonal matrix.} \quad (9)$$

The matrix Q in that context is the product of Householder matrices which are used to zero out the elements underneath the first sub-diagonal of the matrix (and above the first super-diagonal). The tridiagonal eigenvalue problem is then solved by an iterative method. Here we have several choices depending on the needs.

1. If all eigenvalues and eigenvectors are needed and the eigenvectors have to be orthogonal to working precision the preferred method is the [7].
2. If only a few eigenvalues and perhaps the corresponding eigenvectors are needed the preferred method is bisection to compute the eigenvalues and to compute the corresponding eigenvectors of the tridiagonal matrix. Eigenvectors belonging to clustered eigenvalues have to be re-orthogonalized by modified Gram-Schmidt orthogonalization. If there are only a few small clusters of eigenvalues and the rest is well separated this method is even faster than the QR-algorithm if all eigenvalues and eigenvectors are needed. In that case the algorithm can easily be parallelized by doing the inverse iteration for the eigenvectors belonging to different eigenvalues on different processors.
If large clusters of eigenvalues occur the modified Gram-Schmidt orthogonalization can sum up to $O(n^3)$ operations and thus reduce the performance significantly. Especially on parallel computers this can lead to complete performance break-down if the re-orthogonalization for one cluster has to be done on a single processor.
3. If all eigenvalues and eigenvectors are needed and the orthogonality of the eigenvectors is sufficient even if it is not quite up to working precision, then a divide-and-conquer method [8] to compute the eigenvalues and eigenvectors of a tridiagonal matrix is much faster than using the QR-algorithm. The main disadvantage of this method is that it requires more memory than the QR-algorithm.
4. To avoid the trouble of re-orthogonalization with bisection and inverse iteration there is a new method using a relatively robust representation of the tridiagonal matrix [9]. This method is up to now only implemented in the library for the full eigensystem. If only a part of the eigenspectrum is wanted the library routine switches to the traditional algorithm with bisection, inverse iteration, and re-orthogonalization.

Somewhere between direct and iterative methods is Jacobi's method [10] which is based on the idea of zeroing out off-diagonal elements of the matrix A using rotations. The zeroes introduced by one rotation do not persist the next rotation, but the sum of squares of all off-diagonal elements is reduced in each step until the matrix has almost-diagonal form. Then the eigenvalues can be read from the diagonal entries and the eigenvectors from the columns of the orthogonal transformation matrix which is the product of all the rotations applied to A . There are several ideas how to choose the next off-diagonal element to be zeroed out, one is to take all entries of the strictly lower triangle of the matrix row-by-row.

Usually Jacobi's method requires a lot more operations than direct solvers, but for strongly diagonally dominant matrices (where the off-diagonal entries are already much smaller than the diagonal entries) it can be faster than reduction-based methods. Additionally it is easy to parallelize and in some cases it was observed that it can deliver more accurate eigensystems than the other methods.

2.2 Iterative eigensolvers

Iterative methods for eigenvalue problems are applied if only a few eigenvalues and eigenvectors of often very large and sparse matrices are wanted. They are derived from the power method. The background of this method is that for each symmetric matrix $A \in \mathbb{R}^{nn}$ the vector space \mathbb{R}^n has a basis of eigenvectors of A , i.e. each vector $\vec{v} \in \mathbb{R}^n$ can be expressed as

$$\vec{v} = c_1 \vec{x}_1 + c_2 \vec{x}_2 + \cdots + c_n \vec{x}_n \quad (10)$$

with $A\vec{x}_i = \lambda_i \vec{x}_i$. If we multiply equation (10) by A we get

$$\begin{aligned} A\vec{v} &= c_1 A\vec{x}_1 + c_2 A\vec{x}_2 + \cdots + c_n A\vec{x}_n \\ &= c_1 \lambda_1 \vec{x}_1 + c_2 \lambda_2 \vec{x}_2 + \cdots + c_n \lambda_n \vec{x}_n . \end{aligned} \quad (11)$$

If we multiply it with A several times we obtain:

$$\begin{aligned} A^k \vec{v} &= c_1 A^k \vec{x}_1 + c_2 A^k \vec{x}_2 + \cdots + c_n A^k \vec{x}_n \\ &= c_1 \lambda_1^k \vec{x}_1 + c_2 \lambda_2^k \vec{x}_2 + \cdots + c_n \lambda_n^k \vec{x}_n \\ &= \lambda_1^k \left(c_1 \vec{x}_1 + c_2 \left(\frac{\lambda_2}{\lambda_1} \right)^k \vec{x}_2 + \cdots + c_n \left(\frac{\lambda_n}{\lambda_1} \right)^k \vec{x}_n \right) . \end{aligned} \quad (12)$$

It follows that if A has a dominant eigenvalue $|\lambda_1| \gg |\lambda_i|$, $i = 2, \dots, n$ then $\left(\frac{\lambda_i}{\lambda_1}\right) \rightarrow 0$ for $i > 1$ and therefore

$$\lim_{k \rightarrow \infty} \frac{A^k \vec{v}}{\lambda_1^k} = c_1 \vec{x}_1 . \quad (13)$$

For any eigenvalue/eigenvector pair λ_i, \vec{x}_i we have

$$\frac{\vec{x}_i^T A \vec{x}_i}{\vec{x}_i^T \vec{x}_i} = \frac{\vec{x}_i^T \lambda_i \vec{x}_i}{\vec{x}_i^T \vec{x}_i} = \lambda_i . \quad (14)$$

This is called the Rayleigh quotient. Equation (13) together with equation (14) leads to the power iteration with a starting vector \vec{v}_0

$$\vec{v}_{i+1} = \frac{A \vec{v}_i}{\|A \vec{v}_i\|} \quad (15)$$

$$\lambda_{i+1} = \frac{\vec{v}_{i+1}^T A \vec{v}_{i+1}}{\vec{v}_{i+1}^T \vec{v}_{i+1}} \quad (16)$$

until $\|A \vec{v}_i - \lambda_i \vec{v}_i\| < tol |\lambda_i|$ for some tolerance value tol . It can be seen that only matrix-vector products and scalar products are needed, thus if the matrix A is sparse, the number of operations needed in each step is proportional to the number of non-zero elements of A .

The more sophisticated iterative methods consider the operation of the $n \times n$ -matrix A on a small k -dimensional subspace of \mathbb{R}^n or \mathbb{C}^n [11]. They find eigenvalues and eigenvectors of the projection of A onto that space and construct eigenpairs of the whole matrix from those of a smaller $k \times k$ -matrix operating on the subspace. The Lanczos method [12], and the Arnoldi method [13] take the k th Krylow subspace ($\text{span}\{\vec{v}_0, A\vec{v}_0, A^2\vec{v}_0, \dots, A^{k-1}\vec{v}_0\}$). There are other subspace iteration methods like the Davidson method [14], and the Jacobi-Davidson method [15]. In all cases only matrix-vector operations with the matrix A are necessary, eigenvalue computations and other $O(k^3)$ operations are only performed in the lower dimensional subspace.

3 Libraries

To get good performance on modern computers it is recommended to use well optimized mathematical libraries. Many computer vendors supply their computers with basic sequential libraries, e.g. the ESSL [16] on IBM. An important basis for good performance on modern computers where the CPU speed often is almost one magnitude faster than the access to main memory is the “data re-use factor” r [6]. It is the number of operations performed divided by the number of data that are moved between the main memory and the small fast caches. The data re-use factor can best be explained with the example of the different levels of BLAS (Basic Linear Algebra Subprograms) [17][18][19].

The BLAS 1 routines do vector-vector operations such as dot-product where $2n - 1$ operations are done and $2n$ data are transferred from memory leading to $r \approx 1$.

BLAS 2 routines perform matrix-vector operations such as matrix-vector multiplication with roughly $2n^2$ operations and n^2 accesses to memory leading to $r \approx 2$.

BLAS 3 routines which do matrix-matrix operations like matrix-matrix multiplication are the only routines where r scales with n : roughly $2n^3$ operations are performed with $4n^2$ data movements, thus $r \approx n/2$. With that data re-use factor the CPU can be used to almost peak performance. Well-optimized BLAS 3 routines thus are the basis for good performance on modern computers. For many computers there are vendor-optimized BLAS routines. Alternatively there are the Automatically Tuned Linear Algebra Software (Atlas [20]) BLAS routines and for some types of processors there are also BLAS routines optimized by K. Goto [21] available.

3.1 LAPACK, Linear Algebra PACKage

The basic numerical library for linear algebra computations is LAPACK [22]. It is the successor of LINPACK and EISPACK and uses the highest level BLAS routines by applying blocked algorithms. Like the BLAS routines many or all of the LAPACK routines are often vendor optimized, but even if there is no vendor optimized version of LAPACK the public domain version in combination with an optimized version of the BLAS will lead to rather good performance on most computers. The BLAS and LAPACK routines have a Fortran 77 interface which means that all variables are called by reference and two-dimensional arrays are assumed to be stored in column major order.

LAPACK contains routines for the solution of linear equation systems, linear least squares problems, eigenvalue and singular value problems. The routines are written for full, banded, and packed matrices, not for general sparse matrices.

Only direct solvers for linear equation systems and eigenvalue problems are part of LAPACK. For the eigenvalue problem all versions mentioned above can be found in LAPACK.

3.2 ScaLAPACK, Scalable Linear Algebra PACKage

The largest and most flexible public domain library with basic numerical operations for distributed memory parallel systems up to now is . Within the ScaLAPACK project many LAPACK routines were ported to distributed memory computers using message passing. ScaLAPACK, like LAPACK, is a Fortran 77 library.

In parallel libraries for distributed memory multiprocessors one major issue is the distribution

of data to the processors. There are many ways to distribute data, especially matrices, to processors. In the ScaLAPACK Users' Guide [23] several ways to distribute dense matrices are presented and discussed.

For performance and load balancing reasons ScaLAPACK has chosen a two-dimensional block-cyclic distribution for full matrices (see ScaLAPACK Users' Guide). First the matrix is divided into blocks of size $MB \times NB$. These blocks are then uniformly distributed across the $NP \times NQ$ processor grid in a cyclic manner. As a result, every process owns a collection of blocks, which are contiguously stored in a two-dimensional "column major" array.

This local storage convention allows ScaLAPACK software to efficiently use local memory by calling BLAS 3 routines on sub-matrices that may be larger than a single $MB \times NB$ block. Figure 1 shows the distribution of a 9×9 -matrix subdivided into blocks of size 3×2 distributed across a 2×2 -processor grid.

	0		1		0		1		0
0	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}
	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}
	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	a_{39}
1	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}	a_{58}	a_{59}
	a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}	a_{68}	a_{69}
0	a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}	a_{78}	a_{79}
	a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}	a_{87}	a_{88}	a_{89}
	a_{91}	a_{92}	a_{93}	a_{94}	a_{95}	a_{96}	a_{97}	a_{98}	a_{99}

Fig. 1: Block-cyclic 2D distribution of a 9×9 -matrix subdivided into 3×2 -blocks to a 2×2 -processor grid. The numbers outside the matrix indicate processor row and column indices respectively.

The communication in ScaLAPACK is based on the BLACS (Basic Linear Algebra Communication Subroutines) [24]. Public domain versions of the BLACS based on MPI and PVM are available. For IBM p690 clusters there exists also a version of the BLACS in the Parallel Engineering and Scientific Library PESSL [25] using LAPI which can be faster than the public domain version based on MPI.

The basic routines of ScaLAPACK are the PBLAS (Parallel Basic Linear Algebra Subroutines). They contain parallel versions of the BLAS which are parallelized using BLACS for communication and sequential BLAS for computation. Thus PBLAS deliver very good performance on most parallel computers. The PBLAS are internally written in C and allocate additional workspace proportional to the block size. If there is a memory limitation like on BlueGene/L the block sizes have to be chosen small enough to avoid program crashes due to lack of memory. Based on BLACS and PBLAS ScaLAPACK contains direct parallel solvers for dense linear systems (LU and Cholesky decomposition) and linear systems with banded system matrix as well as parallel routines for the solution of linear least squares problems and for singular value decomposition. Routines for the computation of all or some of the eigenvalues and eigenvectors of dense real symmetric matrices and dense complex hermitian matrices and for the generalized symmetric definite eigenvalue problem are also included in ScaLAPACK. All versions of the

direct eigensolvers except the one based on the relatively robust representation are parallelized in ScaLAPACK.

The distribution of data for ScaLAPACK usage has to be done by the user. This is sometimes troublesome. To help users ScaLAPACK also contains additional libraries to treat distributed matrices and vectors. One of them is the TOOLS library, which offers useful routines for example to find out which part of the global matrix a local process has in its memory or to find out the global index of a matrix element corresponding to its local index and vice versa. Unfortunately these routines are documented only in the source code of the routines and not in the Users' Guide.

Another library is the REDIST library which is documented in the ScaLAPACK Users' Guide. It contains routines to copy any block-cyclicly distributed (sub)matrix to any other block-cyclicly distributed (sub)matrix.

ScaLAPACK as a parallel successor of LAPACK attempts to leave the calling sequence of the subroutines unchanged as much as possible in comparison to the corresponding sequential subroutine from LAPACK. The user should have to change only a few parameters in the calling sequence to use ScaLAPACK routines instead of LAPACK routines.

Therefore ScaLAPACK uses so-called descriptors, which are integer arrays containing all necessary information about the distribution of a matrix. This descriptor appears in the calling sequence of the parallel routine instead of the leading dimension of the matrix in the sequential one.

For example the sequential BLAS 3 routine for the computation of

$$C = \alpha AB + \beta C, \quad A \in \mathbb{R}^{M \times K}, \quad B \in \mathbb{R}^{K \times N}, \quad C \in \mathbb{R}^{M \times N},$$

overwriting the original C with the result, has the following calling sequence:

```
...
CALL DGEMM(TRANSA, TRANSB, M, N, K, alpha, A(1,1), LDA, &
           B(1,1), LDB, beta, C(1,1), LDC)
...
```

whereas the ScaLAPACK routine PDGEMM is called

```
...
! Call of PDGEMM with descriptors and the global
! starting indices of the whole matrix
CALL PDGEMM(TRANSA, TRANSB, M, N, K, alpha, A, 1, 1, DESCA, &
           B, 1, 1, DESCB, beta, C, 1, 1, DESCC)
...
```

The main problem is that the user has to take care of the data distribution. He has to choose the processor grid by initializing MP , the number of processor rows, and NP , the number of processor columns and to determine the block size by choosing MB and NB , the number of rows and the number of columns per block, respectively. For many routines, especially for the eigenvalue solvers and the Cholesky decomposition, $MB = NB$ is necessary.

It is completely left to the user to put the correct local part of the matrix to the right places and to put the correct data to the descriptor. The Users' Guide and the comments at the beginning of all routines are sufficient to use ScaLAPACK correctly but for someone not familiar with parallel programming it can be rather difficult and time-consuming to learn how to use it.

The main steps the user has to perform for creating and filling a matrix A with functions of the global indices of its elements are (it is assumed that $MB=NB$ and $N=M=K$):

```

...
! Create the MP * NP processor grid
CALL BLACS_GRIDINIT(ICTXT,'Row-major',MP,NP)
! Find my processor coordinates MYROW and MYCOL
! NPROW should return same value as MP,
! NPCOL should return same value as NP
CALL BLACS_GRIDINFO(ICTXT,NPROW,NPCOL,MYROW,MYCOL)
! Compute local dimensions with routine NUMROC from TOOLS
! N is dimension of the matrix
! NB is block size
MYNUMROWS = NUMROC(N,NB,MYROW,0,NPROW)
MYNUMCOLS = NUMROC(N,NB,MYCOL,0,NPCOL)
! Local leading dimension of A,
! Number of local rows of A
MXLLDA = MYNUMROWS
! Allocate only the local part of A
ALLOCATE(A(MXLLDA,MYNUMCOLS))
! Fill the descriptors, P0 and Q0 are processor coordinates
! of the processor holding global element A(1,1)
CALL DESCINIT(DESCA,N,N,NB,NB,P0,Q0,ICTXT,MXLLDA,INFO)
! Fill the local part of the matrix with data
do j = 1, MYNUMCOLS, NB ! Fill the local column blocks
  do jj=1,min(NB,MYNUMCOLS-j+1) ! All columns of one block
    jloc = j-1 + jj ! local column index
    jglob = (j-1)*NPCOL + MYCOL*NB + jj ! global column index
    do i = 1, MYNUMROWS, NB ! local row blocks in this column
      do ii=1,min(NB,MYNUMROWS-i+1) ! rows in this row block
        iloc = i-1 + ii ! local row index
        iglob = (i-1)*NPROW + MYROW*NB + ii ! global row index
        A(iloc,jloc) = function of global indices iglob, jglob
      enddo
    enddo
  enddo
enddo
...

```

The four nested loops show how local and global indices can be computed from block sizes, the number of rows and columns in the processor grid and the processor coordinates.

3.3 PETSc, Portable, Extensible Toolkit for Scientific Computation

PETSc [26] is not a classical subroutine library but a suite of data structures and routines that provide the building blocks for the solution of large-scale application codes on parallel (or serial) computers. It uses the MPI standard for all message-passing communication.

PETSc includes a suite of iterative linear and nonlinear equation solvers which can easily be

used in application codes written in C and C++. Additionally it provides mechanisms needed in parallel codes like simple parallel matrix and vector assembly routines. Matrices and vectors are PETSc objects and are created and filled by PETSc commands. Thus the user is shielded from many details of the message passing.

Before using PETSc it is necessary to read the User Manual to become familiar with it. There are also examples which can be used as a starting point.

3.4 ARPACK, PARPACK, Parallel ARnoldi PACKage

ARPACK/PARPACK [27][28] are implementations of the implicitly restarted Arnoldi method. The main feature is the so called reverse-communication interface. This means that there is no need to express a matrix-vector product in a fixed way. Even if the matrix is not explicitly available ARPACK can be called. Only the action of the matrix on a vector has to be expressed. The eigenvector computation is an iterative process where an ARPACK routine is called in turn with a matrix-vector multiplication routine provided by the user until the process converges.

ARPACK depends on some LAPACK routines and on the BLAS. It is recommended to use optimized BLAS together with ARPACK. An optimized version of LAPACK can only be used if it is clear that it has the correct version and the correct calling sequences. In all other cases it is recommended to use the LAPACK routines delivered with ARPACK in combination with optimized BLAS routines.

PARPACK is a parallelized version of ARPACK for distributed memory multiprocessors. It can use either MPI or BLACS for communication. It also uses the reverse communication interface. This means that there is no need for a fixed matrix distribution like with ScaLAPACK. The user only has to provide the program with a matrix-vector-product routine which delivers a distributed vector.

4 Final Remarks

On today's computers it is crucial to use tuned library routines for basic computations wherever possible to get a satisfactory part of the theoretical peak performance. On sequential computers the slow memory access is the bottleneck.

On parallel computers for load balancing, minimization of communication and memory usage the distribution of data to the processors is a critical task. Communication costs are high compared to computation costs thus one should have a detailed plan about the program and the libraries to be used before starting with programming. The data layout should ideally be the same for all parts of the computation, but redistribution of data can still be better than not using well-optimized library routines.

For dense linear equation systems or eigenvalue problems with dimensions of $n \leq 50000$ direct solvers like those from ScaLAPACK can be applied on parallel computers, if the dimension comes to an order of 1000000 or more the matrices should be sparse and thus iterative solvers have to be applied to preserve sparsity.

References

- [1] I.S. Duff, J.K. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations*
ACM Trans. Math. Software 9, pp. 302-325 (1997)
- [2] W.H. Liu, *The multifrontal method for sparse matrix solution: theory and practice*
SIAM Review, **34**, 1, (1992)
- [3] L. Hagemann and D. Young, *Applied Iterative Methods*
(New York, Academic Press 1981)
- [4] O. Axelsson, *Iterative Solution Methods*
(Cambridge University Press New York 1994)
- [5] B. Steffen, *Multiscale and Multigrid Procedures*
This volume
- [6] B. Lang, *Direct Solvers for Symmetric Eigenvalue Problems*
Modern Methods and Algorithms of Quantum Chemistry, Proceedings, Jülich, Germany,
pp. 231-259 (2000)
- [7] J.G.F. Francis, *The QR transformation: A unitary analogue to the LR transformation, part I and II*
Computer J. 4, pp. 265-272 and 332-345 (1961/62)
- [8] J.J.M. Cuppen, *A divide and conquer method for the symmetric tridiagonal eigenproblem*
Numerische Mathematik 36, pp. 177-195 (1981)
- [9] B.N. Parlett and I.S. Dhillon, *Relatively robust representations of symmetric tridiagonals*
Linear Algebra and its Applications 309, pp. 121-151 (2000)
- [10] B.N. Parlett, *The Symmetric Eigenvalue Problem*
(SIAM, Philadelphia, PA 1998) (Updated reprint of the 1980 Prentice-Hall edition)
- [11] B. Steffen, *Subspace Methods for Sparse Eigenvalue Problems*
Modern Methods and Algorithms of Quantum Chemistry, Proceedings, Jülich, Germany,
pp. 307-314 (2000)
- [12] J.K. Cullum and R.A. Willoughby, *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*
Volume I: Theory, (Birkhäuser, Boston, Basel, Stuttgart 1985)
- [13] R.B. Morgan, *On Restarting the Arnoldi Method for Large Nonsymmetric Eigenvalue Problems*
Mathematics of Computation **65**, 215, pp. 1213-1230 (1996)
- [14] E.R. Davidson, *The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices*
Journal of Computational Physics 17, pp. 87-94 (1975)

- [15] G.L.G. Sleijpen and H.A. Van der Vorst, *A Jacobi-Davidson Iterative Method for Linear Eigenvalue Problems*
SIAM Journal on Matrix Analysis and Applications, **17**, 2, pp. 401-425 (1996)
electronically: <http://www.siam.org/journals/sirev/42-2/36308.html> (2000)
- [16] *Engineering and Scientific Subroutine Library for AIX Version 4.2*
[http://publib.boulder.ibm.com/infocenter/clresctr/topic/com.ibm.cluster.essl.doc/](http://publib.boulder.ibm.com/infocenter/clresctr/topic/com.ibm.cluster.essl.doc/esslbooks.html#essl_42)
[/esslbooks.html#essl_42](http://publib.boulder.ibm.com/infocenter/clresctr/topic/com.ibm.cluster.essl.doc/esslbooks.html#essl_42)
- [17] C.L. Lawson, R.J. Hanson, D.R. Kinkaid, and F.T. Krogh, *Basic linear algebra subprograms for FORTRAN usage*
ACM Transactions on Mathematical Software **5**, 3, pp. 308-323, 1979
- [18] J.J. Dongarra, J. Du Croz, and R.J. Hanson, *An extended set of FORTRAN basic linear algebra subprograms*
ACM Transactions on Mathematical Software **14**, 1, pp. 1-17, 1988
- [19] J.J. Dongarra, J. Du Croz, and I. Duff, *A set of level 3 basic linear algebra subprograms*
ACM Transactions on Mathematical Software **16**, 1, pp. 1-17, 1990
- [20] *ATLAS - Automatically Tuned Linear Algebra Software*
<http://www.netlib.org/atlas>
- [21] *Texas Advanced Computing Center*
<http://www.tacc.utexas.edu/resources/software/>
- [22] E. Anderson, Z. Bai, C. Bischof et al., *LAPACK Users' Guide, Second Edition*
(SIAM, Philadelphia 1995)
- [23] L.S. Blackford, J. Choi, A. Cleary et al., *ScaLAPACK Users' Guide*
(SIAM, Philadelphia 1997)
- [24] J.J. Dongarra and R.C. Whaley, *A User's Guide to the BLACS v1.1*
(LAPACK Working Note 94, 1997),
<http://www.netlib.org/lapack/lawns/lawn94.ps>
- [25] *Parallel Engineering and Scientific Subroutine Library for AIX Version 3.2*
[http://publib.boulder.ibm.com/infocenter/clresctr/topic/com.ibm.cluster.essl.doc/](http://publib.boulder.ibm.com/infocenter/clresctr/topic/com.ibm.cluster.essl.doc/esslbooks.html#pessl_aix32)
[/esslbooks.html#pessl_aix32](http://publib.boulder.ibm.com/infocenter/clresctr/topic/com.ibm.cluster.essl.doc/esslbooks.html#pessl_aix32)
- [26] S. Balay, W. Gropp, L.C. McInnes, and B. Smith, *PETSc – Portable, Extensible Toolkit for Scientific Computation*
<http://www-unix.mcs.anl.gov/petsc/petsc-2/>
- [27] R.B. Lehouc, D.C. Sorensen, C. Yang, *ARPACK Users' Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*
<http://www.caam.rice.edu/software/ARPACK/>
- [28] K.J. Maschhoff and D.C. Sorensen, *A Portable Implementation of ARPACK for Distributed Memory Parallel Architectures*
http://www.caam.rice.edu/~kristyn/parpack_home.html