Chapter 13

Parallel Linear Algebra Software

Victor Eijkhout, Julien Langou, and Jack Dongarra

In this chapter we discuss numerical software for linear algebra problems on parallel computers. We focus on some of the most common numerical operations: linear system solving and eigenvalue computations.

Numerical operations such as linear system solving and eigenvalue calculations can be applied to two different kinds of matrix: dense and sparse. In dense systems, essentially every matrix element is nonzero. In sparse systems, a sufficiently large number of matrix elements is zero that a specialized storage scheme is warranted; for an introduction to sparse storage, see [3]. Because the two classes are so different, usually different numerical softwares apply to them.

We discuss ScaLAPACK and PLAPACK as the choices for dense linear system solving (see Section 13.1). For solving sparse linear systems, there exist two classes of algorithms: direct methods and iterative methods. We will discuss SuperLU as an example of a direct solver (see Section 13.2.1) and PETSc as an example of iterative solvers (see Section 13.2.2). For eigenvalue computation, the distinction between sparse and dense matrices does not play so large a role as it does in systems solving; for eigenvalues the main distinction is whether one wants all the possible eigenvalues and attendant eigenvectors, or just a subset, typically the few largest or smallest. ScaLAPACK and PLAPACK are packages that start with a dense matrix to calculate all or potentially part of the spectrum (see Section 13.1), while ARPACK (see Section 13.2.3) is preferable when only part of the spectrum is wanted; since it uses reverse communication, PARPACK can handle matrices in sparse format.

In addition to numerical software operations, we discuss the issue of load balancing. We focus on two software packages, ParMetis and Chaco, which can be used in the above-mentioned sparse packages.

We conclude this chapter with a list of software for linear algebra that is freely available on the Web [16].

13.1 Dense Linear Algebra Software

In the most general way to solve a dense linear system in a parallel distributed environment, the matrix is stored as a distributed array, and the system is solved by Gaussian elimination. This is the basic algorithm in ScaLAPACK and PLAPACK, the two packages we discuss for solving a linear system with a distributed dense coefficient matrix. (Sparse systems are discussed in Section 13.2.) On a single processor, the algorithm for dense linear system solving is fairly obvious, although a good deal of optimization is needed for high performance (see Section 13.3). In a distributed context, achieving high performance—especially performance that scales up with increasing processor numbers—requires radical rethinking the basic data structures. Both ScaLAPACK and PLAPACK use the same very specific data distribution named "2D block-cyclic distribution". This distribution is in fact appropriate for any operation on dense distributed systems and thus represents the key to ScaLAPACK and PLAPACK.

In this section, we explain the 2D block-cyclic distribution and focus on how to specify it in ScaLAPACK in order to solve a linear system or an eigen problem. We then briefly compare ScaLAPACK and PLAPACK calling style.

13.1.1 ScaLAPACK

ScaLAPACK is a parallel version of LAPACK, both in function and in software design. Like the earlier package, ScaLAPACK targets linear system solution and eigenvalue calculation for dense and banded matrices. In a way, ScaLAPACK is the culmination of a line of linear algebra packages that started with LINPACK and EISPACK. The coding of those packages was fairly straightforward, using at most Basic Linear Algebra Subprograms (BLAS) Level-1 operations as an abstraction level. LAPACK [1, 15] attains high efficiency on a single processor (or a small number of shared-memory processors) through the introduction of blocked algorithms and the use of BLAS Level-3 operations. ScaLAPACK uses these blocked algorithms in a parallel context to attain scalably high performance on parallel computers.

The seemingly contradictory demands of portability and efficiency are realized in ScaLAPACK through confining the relevant parts of the code to two subroutine libraries: the BLAS for the computational kernels and the BLACS (Basic Linear Algebra Communication Subprograms) for communication kernels. While the BLACS come with ScaLAPACK, the user is to supply the BLAS library; see Section 13.3.

ScaLAPACK is intended to be called from a Fortran environment, as are the examples in this section. The distribution has no C prototypes, but interfacing to a C program is simple, observing the usual name conversion conventions.

13.1.2 ScaLAPACK Parallel Initialization

ScaLAPACK relies for its communications on the BLACS which offers an abstraction layer over MPI. Its main feature is the ability to communicate submatrices, rather than arrays, and of both rectangular and trapezoidal shape. The latter is of obvious value in factorization algorithms. We will not go into the details of the

BLACS here; instead, we focus on the aspects that come into play in the program initialization phase.

Suppose you have divided your parallel machine into an approximately square grid of `nprows` by `npcols` processors. The following two calls set up a BLACS processor grid – its identifier is `ictxt` – and return the current processor number (by row and column) in it:

```
call sl_init(ictxt,nprows,npcols)
call blacs_gridinfo(ictxt,nprows,npcols,myprow,mypcol)
```

Correspondingly, at the end of your code you need to release the grid by

```
call blacs_gridexit(ictxt)
```

A context is to ScaLAPACK what a communicator is in MPI.

ScaLAPACK Data Format

Creating a matrix in ScaLAPACK is, unfortunately, not simple. The difficulty lies in the fact that for scalable high performance on factorization algorithms, the storage mode 2D block-cyclic storage is to be used. The blocking is what enables the use of BLAS Level-3 routines; the cyclic storage is needed for scalable parallelism.

Specifically, the block-cyclic storage implies that a global (i, j) coordinate in the matrix gets mapped to a triplet of (p, l, x) for both the i and the j directions, where p is the processor number, l the block, and x the offset inside the block.

The mapping of the data on the processors is dependent on four parameters: `nprows` and `npcols` for the processor grid (see 13.1.2); and, `bs_i` and `bs_j` for the size of the blocks inside the matrix. In Figure 13.1, we give an example of how the data is distributed among the processors.

The block sizes (`bs_i` and `bs_j`) has to be decided by the user; 64 is usually a safe bet. For generality, let us assume that block sizes `bs_i` and `bs_j` have been chosen, we explain thereafter a general way to map the data onto the processors. First we determine how much storage is needed for the local part of the matrix:

```
mlocal = numroc(mglobal,bs_i,myprow,0,nprows)
nlocal = numroc(nglobal,bs_j,mypcol,0,npcols)
```

where `numroc` is a library function. (The m and n sizes of the matrix need not be equal, since ScaLAPACK also has routines for QR factorization and such.)

Filling in a matrix requires the conversion from (i, j) coordinates to (p, l, x) coordinates. It is best to use conversion functions

```
p_of_i(i,bs,p) = mod(int((i-1)/bs),p)
l_of_i(i,bs,p) = int((i-1)/(p*bs))
x_of_i(i,bs,p) = mod(i-1,bs)+1
```

that take i or j as input, as well as the block size and the number of processors in that direction. The global matrix element (i, j) is then mapped to

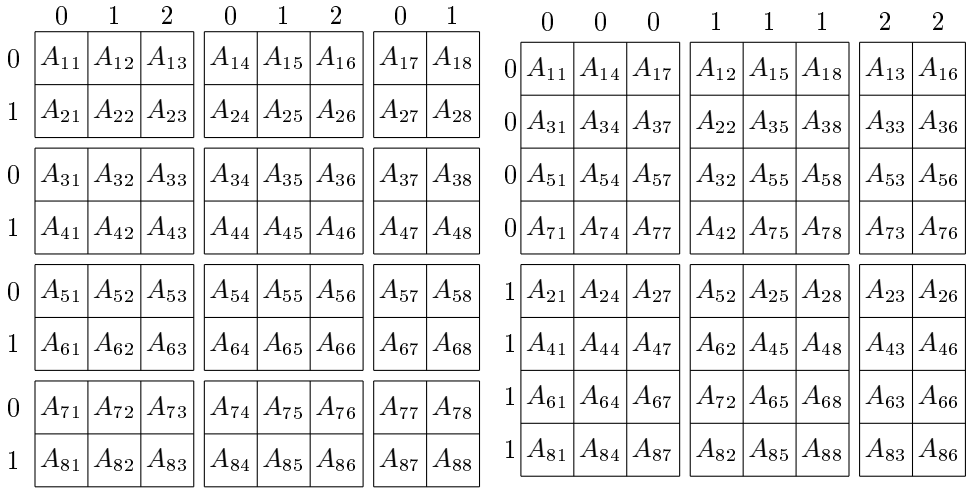

Figure 13.1. *2D block-cyclic distribution of a matrix of order n with parameters (`nprows`= $n/8$,`npcols`= $n/8$,`bs_i`= 2,`bs_j`= 3). On the left, the original data layout, the matrix is partitioned with blocks of size $n/8$; on the right, the data is mapped on a 2×3 processor grid.*

```
pi = p_of_i(i,bs_i,nprows)
li = l_of_i(i,bs_i,nprows)
xi = x_of_i(i,bs_i,nprows)

pj = p_of_i(j,bs_j,npcols)
lj = l_of_i(j,bs_j,npcols)
xj = x_of_i(j,bs_j,npcols)

mat(li*bs_i+xi,lj*bs_j+xj) = mat_global(i,j)
```

if the current processor is (p_i, p_j).

13.1.3 Calling ScaLAPACK Routines

ScaLAPACK routines adhere to the LAPACK naming scheme: `PXYYZZZ`, where `P` indicates parallel; `X` is the "precision" (meaning single or double, real or complex); `YY` is the shape and properties (with `GE` for general rectangular, `TR` for triangular, `PO` for symmetric positive definite ...); and `ZZZ` denotes the function.

For most functions there is a "simple driver" (for instance, `SV` for system solving), which makes the routine name in our example `PDGESV` for double precision, as well as an "expert driver," which has `X` attached to the name, `PDGESVX` in this example. The expert driver usually has more input options and usually returns more diagnostic information.

In the call to a ScaLAPACK routine, information about the matrix has to be

passed by way of a descriptor:

```
      integer desca(9)
      call descinit(desca,
     >     mglobal,nglobal, bs_i,bs_j, 0,0,ictxt,lda,ierr)
      call pdgesv(nglobal,1, mat_local,1,1, desca,ipiv,
     >     rhs_local,1,1, descb, ierr)
```

where `lda` is the allocated first dimension of `a` (`lda>mlocal`).

ScaLAPACK linear solver routines support dense and banded matrices. The drivers for solving a linear system are `PxyySV`, where `yy=GE,GB,PO,PB,PT` for general, general banded, SPD, SPD banded and SPD tridiagonal, respectively. The reader is referred to the *ScaLAPACK Users' Guide* [4] for more details. The input matrix A of the system is on output overwritten with the LU factorization, and the right-hand side B is overwritten with the solution. Temporary storage is needed only for the (integer) pivot locations for nonsymmetric matrices.

For the symmetric (Hermitian) eigenvalue problem there are the simple and expert driver routines PxSYEV (PxHEEV); the nonsymmetric real problem is tackled in two steps: reduction to upper Hessenberg form by `PDGEHRD`, followed by reduction of the Hessenberg matrix to Schur form by `PDLAHQR`; the non-Hermitian complex problem is not supported. ScaLAPACK has routines for the generalized eigenvalue problem only in the symmetric (Hermitian) definite case: `PxSYGST` (with `x=S,D`), and `PxHEGST` (with `x=C,Z`).

13.1.4 PLAPACK

PLAPACK [18] is a package with very similar functionality to ScaLAPACK, but with a different calling style. It also relies on optimized BLAS routines, and is therefore able to achieve a high performance. Whereas ScaLAPACK uses a calling style that is very 'Fortran' in nature, to stay close to its LAPACK roots, PLAPACK uses a more object oriented style. Its interface is similar in philosophy to that of PETSc (see Section 13.2.2).

As an illustration of this object-oriented handling of matrices and vectors, here are matrix-vector multiply and triangular system solve calls:

```
PLA_Gemv( PLA_NO_TRANS, one, A, x, zero, b );
PLA_Trsv( PLA_LOWER_TRIANGULAR, PLA_NO_TRANSPOSE, PLA_UNIT_DIAG, A, b );
```

The distribution of the matrix over the processors is induced a "distribution template" declared by the user, and passed to the matrix creation call:

```
PLA_Matrix_create(  datatype, size, size,
                    templ, PLA_ALIGN_FIRST, PLA_ALIGN_FIRST, &A );
```

PLAPACK wins over ScaLAPACK in user-friendliness in filling in the matrix: like in PETSc, matrix elements can be specified anywhere, and instead of writing them directly into the data structure, they are passed by a `PLA_API_axpy_matrix_to_global` call. On the other hand, PLAPACK seems to lack ScaLAPACK's sophistication of simple and expert drivers, less attention seems to be paid to issues of numerical stability, and the coverage of PLAPACK is not as large as the one of ScaLAPACK.

13.2 Sparse Linear Algebra Software

Software for sparse matrices exploits the sparsity to reduce the storage requirements and the number of floating-point operations performed. Much more than with dense systems, there exists a large variety of software. Two aspects leads to a particularly complex software environment. First, whereas the best method for dense problem is generally easy to decide, the best method for solving a given sparse problem is highly dependent of the matrix itself, its graph and its numerical properties. This means that for any given task we can have several packages. Secondly, software packages are often stronger in some particular aspect of the problem, and an optimal solution is often a combination of packages. This results in a categorization of software along the lines of the various subtasks. Section 13.4 classifies software both ways.

For linear system solving, the most foolproof algorithm is still Gaussian elimination. This is the principle behind SuperLU (Section 13.2.1). In certain applications, especially physics-based ones, the matrix has favorable properties that allow so-called iterative solution methods, which can be much more efficient than Gaussian elimination. The PETSc package is built around such iterative methods and also provides the appropriate preconditioners (see Section 13.2.2). In Section 13.2.3, we address the eigen computation problem and focus on PARPACK. In Section 13.2.4, we try to give the reader a notion of the vertical depth of the software environment for sparse problem by mentioning the load balancing issue.

The way data is handled by sparse software differs from one package to another, and this can be an important consideration in the choice of the software. There are three main ways of handling sparse matrices: (a) the software does not 'have' the matrix but accesses the operator via the matrix-vector operation; this can be realized via reverse communication (e.g. PARPACK) or by having the user provide the matrix-vector product routine; (b) the user provides a pointer to its own data structure to the software (e.g. SuperLU); (c) the user fills the data structure of the software (e.g. PETSc). Pro and against of these three different characteristics are discussed at the end of each part.

13.2.1 SuperLU

SuperLU [8, 19] is one of the foremost direct solvers for sparse linear system. It is available in single-processor, multithreaded, and parallel versions.

Sparse direct solver aim at finding a reordering of the matrix such that the fill-in during the factorization is as small as possible (see as well ?? Chapter:9:Duff-Ng). One of the particular feature of SuperLU is to find cliques in the matrix graph to obtain a high computational efficiency. Eliminating these cliques reduces the cost of the graph algorithms used; and since cliques lead to dense submatrices, it enables the use of higher-level BLAS routines.

The sequential and threaded versions of SuperLU use partial pivoting for numerical stability. Partial pivoting is avoided in the parallel version, however, because it would lead to large numbers of small messages. Instead, "static pivoting" is used, with repair of zero pivots during run time. To compensate for these numerically suboptimal strategies, the solution process uses iterative refinement in a hope to

increase the accuracy of the solution.

Like ScaLAPACK, SuperLU has both simple drivers and expert drivers; the latter give the user opportunity for further steering, return more detailed information, and are more sophisticated in terms of numerical precision.

SuperLU is written in C. The standard installation comes with its own collection of BLAS routines; one can edit the makefile to ensure that an optimized version of the BLAS library is used.

13.2.2 PETSc

PETSc is a library geared towards the solution of partial differential equations. It features tools for manipulating sparse matrix data structures, a sizable repertoire of iterative linear system solvers and preconditioners, as well as nonlinear solvers and time-stepping methods. Although it is written in C, there are Fortran and F90 interfaces.

PETSc distinguishes itself from other libraries in a few aspects. First of all, it is usable as a toolbox: there are many low-level routines that can be used to implement new methods. In particular, there are tools for parallel computation (the so-called `VecScatter` objects) that offer an abstraction layer over straight MPI communication.

Secondly, PETSc's approach to parallelism is very flexible. Many routines operate on local matrices as easily as on distributed ones. Impressively, during the construction of a matrix any processor can specify the value of any matrix element. This, for instance, facilitates writing parallel FEM codes, since, along processor boundaries, elements belonging to different processors will contribute to the value of the same matrix element.

A difference between PETSc and other packages (e.g. SuperLU see section 13.2.1) that is often counted as a disadvantage is that its data structures are internal and not explicitly documented. It is hard for a user to provide its own matrix, the user needs to build up a PETSc matrix by passing matrix elements through function calls.

```
MatCreate(comm,...,&A);
for (i=... )
  for (j=... )
    MatSetValue(A,...,i,j,value,...);
```

This implies that the user faces the choice between maintaining duplicate matrices (one in the native user format and one in PETSc format) with the resulting storage overhead, or using PETSc throughout the code.

Once PETSc data objects have been built, they are used in an object-oriented manner, where the contents and the exact nature of the object are no longer visible:

```
MatMult(A,X,Y);
```

Likewise, parameters to the various objects are kept internal:

```
PCSetType(pc,PCJACOBI);
```

Of particular relevance in the current context is that after the initial creation of an object, its parallel status is largely irrelevant.

13.2.3 PARPACK

Often, in eigenvalue computations, not all eigenvalues or eigenvectors are needed. In such cases one is typically interested in the largest or smallest eigenvalues of the spectrum, or eigenvalues clustered around a certain value.

While ScaLAPACK (section 13.1.1) has routines that can compute a full spectrum, ARPACK focuses on this practically important case of the computation of a small number of eigenvalues and corresponding eigenvectors. It is based on the Arnoldi method[1].

The Arnoldi method is unsuitable for finding eigenvalues in the interior of the spectrum, so such eigenvalues are found by 'shift-invert': given some σ close to the eigenvalues being sought, one solves the eigenvalue equation $(A-\sigma)^{-1}x = \mu x$, since eigenvalues of A close to σ will become the largest eigenvalues of $(A-\sigma)^{-1}$.

Reverse communication program structure

The Arnoldi method has the attractive property of accessing the matrix only through the matrix-vector product operation. However, finding eigenvalues, other than the largest, requires solving linear systems with the given matrix or one derived from it.

Since the Arnoldi method can be formulated in terms of the matrix-vector product operation, this means that ARPACK strictly speaking never needs access to individual matrix elements. To take advantage of this fact, ARPACK uses a technique called 'reverse communication', which dispenses with the need for the user to pass the matrix to the library routines. Thus, ARPACK can work with any user data structure, or even with matrices that are only operatively defined (e.g. FMM).

With reverse communication, whenever a matrix operation is needed, control is passed back to the user program, with a return parameter indicating what operation is being requested. The user then satisfies this request, with input and output in arrays that are passed to the library, and calls the library routine again, indicating that the operation has been performed.

Thus, the structure of a routine using ARPACK will be along these lines:

```
      ido = 0
10    continue
      call dsaupd( ido, .... )
      if (ido.eq.-1 .or. ido.eq.1) then
C        perform matrix vector product
         goto 10
      end if
```

[1] In fact, the pure Arnoldi method would have prohibitive memory demands; what is used here is the 'implicitly restarted Arnoldi method' [11].

For the case of shift-invert or the generalized eigenvalue problem there will be more clauses to the conditional, but the structure stays the same.

ARPACK can be used in a number of different modes, covering the regular and generalized eigenvalue problem, symmetry of the matrix A (and possibly M), and various parts of the spectrum to be computed. Rather than explaining these modes, we will refer the reader to the excellent example drivers provided in the ARPACK distribution.

Practical aspects of using ARPACK

ARPACK is written in Fortran. No C prototypes are given, but the package is easily interfaced to a C application, observing the usual linker naming conventions for Fortran and C routines. The parallel version of ARPACK, PARPACK, can be based on either MPI or the BLACS, the communication layer of ScaLAPACK; see section 13.1.1. ARPACK uses LAPACK, and unfortunately, relies on an older version than the current. While this version is included in the distribution, it means that it can not easily be replaced by a vendor-optimized version.

The flip side of the data independence obtained by reverse communication is that it is incumbent on the user to provide a matrix vector product, which especially in the parallel case is not trivial. Also, in the shift-invert case it is up to the user to provide a linear system solver.

13.2.4 Load Balancing

Many applications can be distributed in more than one way over a parallel architecture. Even if one distribution is the natural result of one component of the computation, for instance setup of a grid and generation of the matrix, a subsequent component, for instance an eigenvalue calculation, may be so labour-intensive that the cost of a full data redistribution may be outweighed by resulting gains in parallel efficiency.

In this section we discuss two packages for graph partitioning: ParMetis and Chaco. These packages aim at finding a partitioning of a graph that assigns roughly equally sized subgraphs to processors, thereby balancing the work load, while minimizing the size of the separators and the consequent communication cost. The reader will also be interested in Chapter **??** (chap:7:boman.devine.karypis).

13.2.5 ParMetis

ParMetis [10, 17] is a parallel package for mesh or graph partitioning for parallel load balancing. It is based on a three-step coarsening / partitioning / uncoarsening algorithm that the authors claim is faster than multiway spectral bisection. It can be used in several modes, for instance repartitioning graphs from adaptively refined meshes, or partitioning graphs from multi-physics simulations.

The input format of ParMetis, in its serial form, is a variant on compressed matrix storage: the adjacency of each element is stored consecutively (excluding the diagonal, but for each pair u, v storing both (u, v) and (v, u)), with a pointer

array indicating where each element's data starts and ends. Both vertex and edge weights can be specified optionally. The parallel version of the graph input format takes blocks of consecutive nodes and allocates these to subsequent processors. An array that is identical on each processor then indicates which range of variables each processor owns. The distributed format uses global numbering of the nodes.

The output of ParMetis is a mapping of node numbers to processors. No actual redistribution is performed.

13.2.6 Chaco

The Chaco [14] package comprises several algorithms for graph partitioning, including inertial, spectral, Kernighan-Lin, and multi-level algorithms. It can be used in two modes:

stand-alone In this mode, input and output are done through files.

library Chaco can also be linked to C or Fortran codes, and all data is passed through arrays.

Unlike ParMetis, Chaco runs only sequentially.

Zoltan [5] is a package for dynamic load balancing that builds on top of Chaco. Thanks to an object-oriented design, it is data structure neutral, so it can be interfaced using existing user data structures.

13.3 Support Libraries

The packages in this chapter rely on two very common support libraries: MPI and BLAS. On most parallel distributed platform, there is at least one MPI library installed.

The Basic Linear Algebra Subprograms [7], or BLAS for short, a fairly simple linear algebra kernels that you could easily code yourself in a few lines. You could also download the source and compile it [13]. However, this is unlikely to give good performance, no matter the level of sophistication of your compiler. The recommended way is to use vendor libraries which are available on a number of platforms, for instance in the `ESSL` library on IBM machines and the `mkl` on Intel. On platforms without such vendor libraries (or sometimes even if they are present) we recommend that you install the ATLAS [12] package, which gives a library tuned to your specific machine. In a nutshell, ATLAS has a search algorithm that generates many implementations of each kernel, saving the one with the highest performance. This will far outperform anything you can write by hand.

13.4 Freely Available Software for Linear Algebra on the Web

Tables 13.1–13.5 present a list of freely available software for the solution of linear algebra problems. The interest is in software for high-performance computers that is

available in "open source" form on the web for solving problems in numerical linear algebra, specifically dense, sparse direct and iterative systems and sparse iterative eigenvalue problems.

Additional pointers to software can be found at: www.nhse.org/rib/repositories/nhse/catalog/#Numerical_Programs_and_Routines A survey of Iterative Linear System Solver Packages can be found at: www.netlib.org/utk/papers/iterative-survey.

Package	Support	Type		Language			Mode		Dense	Sparse
		Real	Complex	f77	c	c++	Seq	Dist		
ATLAS	yes	X	X	X	X		X		X	
BLAS	yes	X	X	X	X		X		X	
FLAME	yes	X	X	X	X		X		X	
LINALG	?									
MTL	yes	X				X	X		X	
NETMAT	yes	X				X	X		X	
NIST S-BLAS	yes	X	X	X	X		X			X
PSBLAS	yes	X	X	X	X		X	M		X
SparseLib++	yes	X	X		X	X	X			X
uBLAS	yes	X	X		X	X	X		X	

Table 13.1. *Support routines for numerical linear algebra.*

Package	Support	Type		Language			Mode	
		Real	Complex	f77	c	c++	Seq	Dist
LAPACK	yes	X	X	X	X		X	
LAPACK95	yes	X	X	95			X	
NAPACK	yes	X		X				
PLAPACK	yes	X	X	X	X			M
PRISM	yes	X		X			X	M
ScaLAPACK	yes	X	X	X	X			M/P

Table 13.2. *Available software for dense matrix.*

Reading List

Linear systems The literature on linear system solving, like the research in this topic, is mostly split along the lines of direct versus iterative solvers. An introduction that covers both (as well as eigenvalue methods) is the book by Dongarra et al. [6]. A very practical book about linear system solving by iterative methods is the 'Templates' book [3], which in addition to the mathematical details contains sections on sparse data storage and other practical matters. More in depth and less software oriented is the book by Saad [9].

Eigensystems Along the lines of the Templates book for linear systems is a similar book for eigenvalues problems [2].

Package	Support	Type		Language			Mode			
		Real	Complex	f77	c	c++	Seq	Dist	SPD	Gen
DSCPACK	yes	X			X		X	M	X	
HSL	yes	X	X	X			X		X	X
MFACT	yes	X			X		X		X	
MUMPS	yes	X	X	X	X		X	M	X	X
PSPASES	yes	X		X	X			M	X	
SPARSE	?	X	X		X		X		X	X
SPOOLES	?	X	X		X		X	M		X
SuperLU	yes	X	X	X	X		X	M		X
TAUCS	yes	X	X		X		X		X	X
UMFPACK	yes	X	X		X		X			X
Y12M	?	X		X			X		X	X

Table 13.3. *Sparse direct solvers.*

Package	Support	Type		Language			Mode			
		Real	Complex	f77	c	c++	Seq	Dist	Sym	Gen
(B/H)LZPACK	yes	X	X	X			X	M/P	X	X
HYPRE	yes	X			X		X	M	X	
QMRPACK	?	X	X	X			X		X	X
LASO	?	X		X			X		X	
P_ARPACK	yes	X	X	X			X	M/P		X
PLANSO	yes	X		X			X	M	X	
SLEPc	yes	X	X		X		X	M	X	X
SPAM	yes	X		90			X		X	
TRLAN	yes	X		X			X	M	X	

Table 13.4. *Sparse eigenvalue solvers.*

Package	Support	Type		Language			Mode		Precond.		Iterative Solvers	
		Real	Comp.	f77	c	c++	Seq	Dist	SPD	Gen	SPD	Gen
AZTEC	no	X			X		X	M	X	X	X	X
BILUM	yes	X		X			X	M		X		X
BlockSolve95	?	X		X	X	X		M	X	X	X	X
BPKIT	yes	X		X	X	X	X	M	X	X		
CERFACS	yes	X	X	X			X	M			X	X
HYPRE	yes	X		X	X		X	M	X	X	X	X
IML++	?	X		X	X	X	X				X	X
ITL	yes	X				X	X				X	X
ITPACK	?	X		X			X				X	X
LASPack	yes	X			X		X				X	X
LSQR	yes	X		X	X		X					X
pARMS	yes	X		X	X		X	M			X	X
PARPRE	yes	X			X			M	X	X		
PETSc	yes	X	X	X	X		X	M	X	X	X	X
P-SparsLIB	yes	X		X				M		X		X
PSBLAS	yes	X	X	f90	X		X	M	X	X	X	X
QMRPACK	?	X	X	X			X				X	X
SLAP	?	X		X						X		
SPAI	yes	X			X		X	M			X	X
SPLIB	?	X		X			X				X	X
SPOOLES	?	X	X		X		X	M	X	X	X	X
SYMMLQ	yes	X	X	X			X				X	X
TAUCS	yes	X	X		X		X		X	X	X	X
Templates	yes	X		X	X		X				X	X
Trilinos	yes	X				X	X	M	X	X	X	X

Table 13.5. *Sparse iterative solvers.*

Bibliography

[1] E. Anderson, Z. Bai, C. H. Bischof, J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[2] Z. Bai, J. Demmel, J. J. Dongarra, A. Ruhe, and H. A. van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems, A Practical Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. A. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1994. http://www.netlib.org/templates/.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. S. Dhillon, J. J. Dongarra, S. Hammerling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[5] E. Boman, K. Devine, R. Heaphy, B. Hendrickson, W. Mitchell, M. St. John, and C. Vaughan, *Zoltan home page*. http://www.cs.sandia.gov/Zoltan, 1999.

[6] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Numerical Linear Algebra for High-Performance Computers*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[7] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. Krogh, *Basic linear algebra subprograms for FORTRAN usage*, ACM Trans. Math. Software, 5 (1979), pp. 308–323.

[8] X. S. Li, *Sparse Gaussian Eliminiation on High Performance Computers*, PhD thesis, University of California at Berkeley, 1996.

[9] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003. originally published

by PWS Publishing Company, Boston, 1996; this edition is available for download from http://www.cs.umn.edu/~saad.

[10] K. Schloegel, G. Karypis, and V. Kumar, *Parallel multilevel algorithms for multi-constraint graph partitioning*, in Proceedings of EuroPar-2000, 2000.

[11] D. C. Sorensen, *Implicit application of polynomial filters in a k-step Arnoldi method*, SIAM J. Matrix Anal., 13 (1992), pp. 357–385.

[12] R. C. Whaley, A. Petitet, and J. J. Dongarra, *Automated empirical optimizations of software and the ATLAS project*, Parallel Computing, 27 (2001), pp. 3–35.

[13] *BLAS home page*. http://www.netlib.org/blas.

[14] *Chaco home page*. http://www.cs.sandia.gov/~bahendr/chaco.html.

[15] *LAPACK home page*. http://www.netlib.org/lapack.

[16] *Freely available software for linear algebra on the web (may 2004)*. http://www.netlib.org/utk/people/JackDongarra/la-sw.html.

[17] *ParMETIS home page*. http://www-users.cs.umn.edu/~karypis/metis/parmetis/index.html.

[18] *PLAPACK home page*. http://www.cs.utexas.edu/users/plapack/.

[19] *SuperLU home page*. http://www.nersc.gov/~xiaoye/SuperLU/.