Chapter 10

Common Issues

10.1 Sparse Matrix Storage Formats

J. Dongarra

The efficiency of most of the iterative methods considered in this book is determined primarily by the performance of the matrix-vector product and therefore on the storage scheme used for the matrix. Often, the storage scheme used arises naturally from the specific application problem.

In this section we will review some of the more popular sparse matrix formats that have been used in numerical software packages such as ITPACK [263], NSPCG [345], and SPARSPAK [191], some of which have more recently been adopted as part of a new software standard by the BLAS Technical Forum (see ETHOME for further information). In §10.2.2, we demonstrate how the matrix-vector product is formulated using two of the sparse matrix formats.

If the coefficient matrix A is sparse, large scale eigenvalue problems can be most efficiently solved if the zero elements of A are neither manipulated nor stored. Sparse storage schemes allocate contiguous storage in memory for the nonzero elements of the matrix, and perhaps a limited number of zeros. This, of course, requires a scheme for knowing where the elements fit into the full matrix.

There are many methods for storing the data (see, for instance, Saad [386] and Eijkhout [156]). Here we will discuss compressed row and column storage, block compressed row storage, diagonal storage, jagged diagonal storage, and skyline storage.

10.1.1 Compressed Row Storage

The compressed row and column (in the next section) storage formats are the most general: they make absolutely no assumptions about the sparsity structure of the matrix, and they don't store any unnecessary elements. On the other hand, they are not very efficient, needing an indirect addressing step for every single scalar operation in a matrix-vector product or preconditioner solve.

The compressed row storage (CRS) format puts the subsequent nonzeros of the matrix rows in contiguous memory locations. Assuming we have a nonsymmetric sparse matrix A, we create three vectors: one for floating point numbers (`val`) and the other

two for integers (`col_ind`, `row_ptr`). The `val` vector stores the values of the nonzero elements of the matrix A as they are traversed in a row-wise fashion. The `col_ind` vector stores the column indexes of the elements in the `val` vector. That is, if $\texttt{val(k)} = a_{i,j}$, then $\texttt{col\_ind(k)} = j$. The `row_ptr` vector stores the locations in the `val` vector that start a row; that is, if $\texttt{val(k)} = a_{i,j}$, then $\texttt{row\_ptr(i)} \le k < \texttt{row\_ptr(i+1)}$. By convention, we define $\texttt{row\_ptr(n+1)} = nnz + 1$, where nnz is the number of nonzeros in the matrix A. The storage savings for this approach is significant. Instead of storing n^2 elements, we need only $2nnz + n + 1$ storage locations.

As an example, consider the nonsymmetric matrix A defined by

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}. \tag{10.1}$$

The CRS format for this matrix is then specified by the arrays {`val`, `col_ind`, `row_ptr`} given below:

`val`	10	-2	3	9	3	7	8	7	$3 \cdots 9$	13	4	2	-1
`col_ind`	1	5	1	2	6	2	3	4	$1 \cdots 5$	6	2	5	6

`row_ptr`	1	3	6	9	13	17	20

If the matrix A is symmetric, we need only store the upper (or lower) triangular portion of the matrix. The tradeoff is a more complicated algorithm with a somewhat different pattern of data access.

10.1.2 Compressed Column Storage

Analogous to CRS, there is compressed column storage (CCS), which is also called the *Harwell-Boeing* sparse matrix format [139]. The CCS format is identical to the CRS format except that the columns of A are stored (traversed) instead of the rows. In other words, the CCS format is the CRS format for A^T.

The CCS format is specified by the 3 arrays {`val`, `row_ind`, `col_ptr`}, where `row_ind` stores the row indices of each nonzero, and `col_ptr` stores the index of the elements in `val` which start a column of A. The CCS format for the matrix A in (10.1) is given by

`val`	10	3	3	9	7	8	4	8	$8 \cdots 9$	2	3	13	-1
`row_ind`	1	2	4	2	3	5	6	3	$4 \cdots 5$	6	2	5	6

`col_ptr`	1	4	8	10	13	17	20

10.1.3 Block Compressed Row Storage

If the sparse matrix A is composed of square dense blocks of nonzeros in some regular pattern, we can modify the CRS (or CCS) format to exploit such block patterns. Block matrices typically arise from the discretization of partial differential equations in which there are several *degrees of freedom* associated with a grid point. We then partition

the matrix in small blocks with a size equal to the number of degrees of freedom and treat each block as a dense matrix, even though it may have some zeros.

If n_b is the dimension of each block and $nnzb$ is the number of nonzero blocks in the $n \times n$ matrix A, then the total storage needed is $nnz = nnzb \times n_b^2$. The block dimension n_d of A is then defined by $n_d = n/n_b$.

Similar to the CRS format, we require three arrays for the BCRS format: a rectangular array for floating point numbers (`val`$(1 : nnzb, 1 : n_b, 1 : n_b)$) which stores the nonzero blocks in (block) row-wise fashion, an integer array (`col_ind`$(1 : nnzb)$) which stores the actual column indices in the original matrix A of the $(1, 1)$ elements of the nonzero blocks, and a pointer array (`row_blk`$(1 : n_d + 1)$) whose entries point to the beginning of each block row in `val(:,:,:)` and `col_ind(:)`. The savings in storage locations and reduction in time spent doing indirect addressing for block compressed row storage (BCRS) over CRS can be significant for matrices with a large n_b.

10.1.4 Compressed Diagonal Storage

If the matrix A is banded with bandwidth that is fairly constant from row to row, then it is worthwhile to take advantage of this structure in the storage scheme by storing subdiagonals of the matrix in consecutive locations. Not only can we eliminate the vector identifying the column and row, but we can pack the nonzero elements in such a way as to make the matrix-vector product more efficient. This storage scheme is particularly useful if the matrix arises from a finite element or finite difference discretization on a tensor product grid.

We say that the matrix $A = (a_{i,j})$ is *banded* if there are nonnegative constants p, q, called the left and right *halfbandwidth*, such that $a_{i,j} \neq 0$ only if $i-p \leq j \leq i+q$. In this case, we can allocate for the matrix A an array `val(1:n,-p:q)`. The declaration with reversed dimensions `(-p:q,n)` corresponds to the LINPACK band format [132], which, unlike compressed diagonal storage (CDS), does not allow for an efficiently vectorizable matrix-vector multiplication if `p + q` is small.

Usually, band formats involve storing some zeros. The CDS format may even contain some array elements that do not correspond to matrix elements at all. Consider the nonsymmetric matrix A defined by

$$A = \begin{bmatrix} 10 & -3 & 0 & 0 & 0 & 0 \\ 3 & 9 & 6 & 0 & 0 & 0 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 0 & 0 & 8 & 7 & 5 & 0 \\ 0 & 0 & 0 & 9 & 9 & 13 \\ 0 & 0 & 0 & 0 & 2 & -1 \end{bmatrix}. \tag{10.2}$$

Using the CDS format, we store this matrix A in an array of dimension `(6,-1:1)` using the mapping

$$\texttt{val(i,j)} = a_{i,i+j}. \tag{10.3}$$

Hence, the rows of the `val(:,:)` array are

`val(:,-1)`	0	3	7	8	9	2
`val(:, 0)`	10	9	8	7	9	-1
`val(:,+1)`	-3	6	7	5	13	0

Notice the two zeros corresponding to nonexisting matrix elements.

10.1.5 Jagged Diagonal Storage

The jagged diagonal storage (JDS) format can be useful for the implementation of iterative methods on parallel and vector processors (see Saad [385]). Like the CDS format, it gives a vector length of essentially the same size as the matrix. It is more space-efficient than CDS at the cost of a gather/scatter operation.

A simplified form of JDS, called ITPACK storage or Purdue storage, can be described as follows. For the following nonsymmetric matrix, all elements are shifted left:

$$\begin{bmatrix} 10 & -3 & 0 & 1 & 0 & 0 \\ 0 & 9 & 6 & 0 & -2 & 0 \\ 3 & 0 & 8 & 7 & 0 & 0 \\ 0 & 6 & 0 & 7 & 5 & 4 \\ 0 & 0 & 0 & 0 & 9 & 13 \\ 0 & 0 & 0 & 0 & 5 & -1 \end{bmatrix} \longrightarrow \begin{bmatrix} 10 & -3 & 1 & \\ 9 & 6 & -2 & \\ 3 & 8 & 7 & \\ 6 & 7 & 5 & 4 \\ 9 & 13 & & \\ 5 & -1 & & \end{bmatrix}$$

after which the columns are stored consecutively. All rows are padded with zeros on the right to give them equal length. Corresponding to the array of matrix elements `val(:,:)`, an array of column indices, `col_ind(:,:)`, is also stored:

`val`$(:,1)$	10	9	3	6	9	5
`val`$(:,2)$	-3	6	8	7	13	-1
`val`$(:,3)$	1	-2	7	5	0	0
`val`$(:,4)$	0	0	0	4	0	0

,

`col_ind`$(:,1)$	1	2	1	2	5	5
`col_ind`$(:,2)$	2	3	3	4	6	6
`col_ind`$(:,3)$	4	5	4	5	0	0
`col_ind`$(:,4)$	0	0	0	6	0	0

.

It is clear that the padding zeros in this structure may be a disadvantage, especially if the bandwidth of the matrix varies strongly. Therefore, in the CRS format, we reorder the rows of the matrix decreasingly according to the number of nonzeros per row. The compressed and permuted diagonals are then stored in a linear array. The new data structure is called *jagged diagonals*.

Specifically, we store the (dense) vector of all the first elements in `val`, `col_ind` from each row, together with an integer vector containing the column indices of the corresponding elements. This is followed by the second jagged diagonal consisting of the elements in the second positions from the left. We continue to construct more and more of these jagged diagonals (whose length decreases).

The number of jagged diagonals is equal to the number of nonzeros in the first row, i.e., the largest number of nonzeros in any row of A. The data structure to represent the $n \times n$ matrix A therefore consists of a permutation array (`perm(1:n)`), which reorders the rows, a floating point array (`jdiag(:)`) containing the jagged diagonals in succession, an integer array (`col_ind(:)`) containing the corresponding column indices, and finally a pointer array (`jd_ptr(:)`) whose elements point to the beginning of each jagged diagonal. The advantages of JDS for matrix multiplications were discussed by Saad in [385].

The JDS format for the above matrix A in using the linear arrays {`perm`, `jdiag`, `col_ind`, `jd_ptr`} is given below (jagged diagonals are separated by semicolons):

`jdiag`	1	3	7	8	10	2;	9	9	$8 \cdots$ -1;	9	6	7	5;	13
`col_ind`	1	1	2	3	1	5;	4	2	$3 \cdots 6$;	5	3	4	5;	6

`perm`	5	2	3	4	1	6	`jd_ptr`	1	7	13	17	.

10.1.6 Skyline Storage

The final storage scheme we consider is for skyline matrices, which are also called variable band or profile matrices (see Duff, Erisman, and Reid [138]). It is mostly of importance in direct solution methods, but it can be used for handling the diagonal blocks in block matrix factorization methods. A major advantage of solving linear systems having skyline coefficient matrices is that when pivoting is not necessary, the skyline structure is preserved during Gaussian elimination. If the matrix is symmetric, we only store its lower triangular part. A straightforward approach in storing the elements of a skyline matrix is to place all the rows (in order) into a floating point array (`val(:)`), and then keep an integer array (`row_ptr(:)`) whose elements point to the beginning of each row. The column indices of the nonzeros stored in `val(:)` are easily derived and are not stored.

For a nonsymmetric skyline matrix such as the one illustrated in Figure 10.1, we store the lower triangular elements in skyline storage (SKS) format and store the upper triangular elements in a column-oriented SKS format (transpose stored in row-wise SKS format). These two separated *substructures* can be linked in a variety of ways. One approach, discussed by Saad in [386], is to store each row of the lower triangular part and each column of the upper triangular part contiguously into the floating point array (`val(:)`). An additional pointer is then needed to determine where the diagonal elements, which separate the lower triangular elements from the upper triangular elements, are located.

```
+  x  x                                                  
x  +  x  x                                               
x  x  +  x           x                                   
   x  x  +  x  x  x  x  x                                
         x  +  x  x  x                                   
      x  x  x  +  x  x                                   
            x  x  +  x           x        x              
            x  x  x  +  x  x     x        x              
            x  x  x  x  +  x  x  x        x              
                        x  +  x  x  x     x              
                  x  x  x  x  +  x  x                    
                        x  x  +  x           x         
                        x  x  x  +  x  x  x  x         
                                    +  x  x  x         
                                    x  x  +  x  x      
                                    x  x  x  +  x  x   
                                                +  x  x
                                    x  x  x  x  +  x   
                                                      +
```

Figure 10.1: Profile of a nonsymmetric skyline or variable-band matrix.

10.2 Matrix-Vector and Matrix-Matrix Multiplications

J. Dongarra, P. Koev, and X. Li

10.2.1 BLAS

For the past 15 years or so, there has been a great deal of activity in the area of algorithms and software for solving linear algebra problems. The linear algebra community has long recognized the need for help in developing algorithms into software libraries, and several years ago, as a community effort, put together a de facto standard for identifying basic operations required in linear algebra algorithms and software. The hope was that the routines making up this standard, known collectively as the Basic Linear Algebra Subprograms (BLAS), would be efficiently implemented on advanced-architecture computers by many manufacturers, making it possible to reap the portability benefits of having them efficiently implemented on a wide range of machines. This goal has been largely realized.

The key insight of our approach to designing linear algebra algorithms for advanced architecture computers is that the frequency with which data are moved between different levels of the memory hierarchy must be minimized in order to attain high performance. Thus, our main algorithmic approach for exploiting both vectorization and parallelism in our implementations is the use of block-partitioned algorithms, particularly in conjunction with highly tuned kernels for performing matrix-vector and matrix-matrix operations (the Level 2 and 3 BLAS). In general, the use of block-partitioned algorithms requires data to be moved as blocks, rather than as vectors or scalars, so that although the total amount of data moved is unchanged, the latency (or startup cost) associated with the movement is greatly reduced because fewer messages are needed to move the data.

A second key idea is that the performance of an algorithm can be tuned by a user by varying the parameters that specify the data layout. On shared memory machines, this is controlled by the block size, while on distributed memory machines it is controlled by the block size and the configuration of the logical process mesh.

The question arises, "How can we achieve sufficient control over these various factors that effect performance to obtain the levels of performance that machines can offer?" The answer is through use of the BLAS.

There are now three levels of BLAS:

Level 1 BLAS [288]: for vector operations, such as $y \leftarrow \alpha x + y$;

Level 2 BLAS [133]: for matrix-vector operations, such as $y \leftarrow \alpha Ax + \beta y$;

Level 3 BLAS [134]: for matrix-matrix operations, such as $C \leftarrow \alpha AB + \beta C$.

Here, A, B, and C are matrices; x and y are vectors; and α and β are scalars.

The Level 1 BLAS are used in packages like LAPACK, for convenience rather than for performance: they perform an insignificant fraction of the computation, and they cannot achieve high efficiency on most modern supercomputers.

The Level 2 BLAS can achieve near-peak performance on many vector processors, such as a single processor of a CRAY X-MP or Y-MP, or a Convex C-2 machine. However, on other vector processors such as a CRAY-2 or an IBM 3090 VF, the performance

of the Level 2 BLAS is limited by the rate of data movement between different levels of memory.

The Level 3 BLAS overcome this limitation. This third level of BLAS performs $O(n^3)$ floating point operations on $O(n^2)$ data, whereas the Level 2 BLAS perform only $O(n^2)$ operations on $O(n^2)$ data. The Level 3 BLAS also allow us to exploit parallelism in a way that is transparent to the software that calls them. While the Level 2 BLAS offer some scope for exploiting parallelism, greater scope is provided by the Level 3 BLAS.

To a great extent, the user community embraced the BLAS, not only for performance reasons, but also because developing software around a core of common routines like the BLAS is good software engineering practice. Highly efficient machine-specific implementations of the BLAS are available for most modern high-performance computers. The BLAS have enabled software to achieve high performance with portable code.

In the spirit of the earlier BLAS meetings and the standardization efforts of the Massage-Passing Interface (MPI) and high-performance Fortran (HPF) forums, a technical forum was established to consider expanding the BLAS in the light of modern software, language, and hardware developments. The BLAS Technical Forum meetings began with a workshop in November 1995 at the University of Tennessee. Meetings were hosted by universities and software and hardware vendors.

Various working groups were established to consider issues such as overall functionality, language interfaces, sparse BLAS, distributed memory dense BLAS, extended and mixed precision BLAS, interval BLAS, and extensions to the existing BLAS. The rules of the forum were adopted from those used for the MPI and HPF forums.

A major aim of the standards defined in this document are to enable linear algebra libraries (both public domain and commercial) to interoperate efficiently, reliably, and easily. We believe that hardware and software vendors, higher level library writers, and application programmers all benefit from the efforts of this forum and are the intended end users of these standards.

Further information about the BLAS and BLAS Technical Forum can be obtained on the book's homepage, ETHOME.

10.2.2 Sparse BLAS

In the spirit of the dense BLAS, work is underway in the BLAS Technical Forum to establish a standard for the sparse BLAS. The sparse BLAS interface addresses computational routines for unstructured sparse matrices. Sparse BLAS also contains the three levels of operations as in the dense case. However, only a small subset of the dense BLAS is specified:

Level 1: sparse dot product, vector update, and gather/scatter;

Level 2: sparse matrix-vector multiply and triangular solve;

Level 3: sparse matrix–dense matrix multiply and triangular solve with multiple right-hand sides.

These are the basic operations used in most of the iterative algorithms for solving sparse linear equations and eigensystems. The Level 2 and 3 sparse BLAS interfaces support nine sparse matrix storage formats. The nine formats are divided into two

categories: the *point entry* formats such as compressed row and the *block entry* formats such as block compressed row. In the block entry formats, the sparsity structure is represented as a series of small dense blocks. Note that the nine formats include some surveyed in §10.1, except JDS and SKS.

The interface design of the sparse BLAS is fundamentally different from the dense BLAS. Since there is no single "best" storage format for any sparse matrix operation, the sparse matrix arguments to the Level 2 and 3 sparse BLAS do not use one particular storage format. Instead, a *generic interface* is defined for these routines, in which a sparse matrix argument is a *handle* (integer) representing the matrix. Before calling a sparse BLAS routine, one needs to call a creation routine to create the sparse matrix handle from one of the nine formats. After calling the sparse BLAS routine, one needs to call a cleanup routine to release the resources associated with the matrix handle. This handle-based approach allows one to use the sparse BLAS independently of the sparse matrix storage. The internal representation is implementation dependent and can be chosen for best performance.

Since matrix-vector products often account for most of the time spent in iterative methods, several research efforts have tried to optimize their performance [438, 439, 240, 239]. In matrix-vector multiplication, each matrix entry participates in only one operation, but each vector entry may participate in more than one operation. A primary goal of optimization is to reduce the amount of movement of the source vector between different levels of memory. The optimization techniques include matrix reordering, cache blocking, and register blocking. A recently developed toolbox, called SPARSITY, embraces all these techniques [240, 239]. Depending on the matrix structure, the authors have observed up to threefold speedup even on uniprocessors. SPARSITY also contains mechanisms to automatically choose the best block sizes based on matrix and machine characteristics.

In many of the iterative methods discussed earlier, both the product of a matrix and that of its transpose times a vector are needed; that is, given an input vector x we want to compute products

$$y = Ax \qquad \text{and} \qquad y = A^T x.$$

In the following two subsections, we will present algorithms for the storage format from §10.1, CRS.

10.2.2.1 CRS Matrix-Vector Product

The matrix-vector product $y = Ax$ using CRS format can be expressed in the usual way:

$$y_i = \sum_j a_{i,j} x_j,$$

since this traverses the rows of the matrix A. For an $n \times n$ matrix A, the matrix-vector multiplication is given by

```
for i = 1, n
    y(i)  = 0
    for j = row_ptr(i), row_ptr(i+1) - 1
        y(i) = y(i) + val(j) * x(col_ind(j))
    end;
end;
```

Since this method only multiplies nonzero matrix entries, the operation count is 2 times the number of nonzero elements in A, which is a significant savings over the dense operation requirement of $2n^2$.

For the transpose product $y = A^T x$ we cannot use the equation

$$y_i = \sum_j (A^T)_{i,j} x_j = \sum_j a_{j,i} x_j,$$

since this implies traversing columns of the matrix, an extremely inefficient operation for matrices stored in CRS format. Hence, we switch indices to

$$\text{for all } j, \text{ do for all } i: \qquad y_i \leftarrow y_i + a_{j,i} x_j.$$

The matrix-vector multiplication involving A^T is then given by

```
for i = 1, n
    y(i) =  0
end;
for j = 1, n
    for i = row_ptr(j), row_ptr(j+1)-1
        y(col_ind(i)) = y(col_ind(i)) + val(i) * x(j)
    end;
end;
```

Both matrix-vector products above have largely the same structure, and both use indirect addressing. Hence, their vectorizability properties are the same on any given computer. However, the first product ($y = Ax$) has a more favorable memory access pattern in that (per iteration of the outer loop) it reads two vectors of data (a row of matrix A and the input vector x) and writes one scalar. The transpose product ($y = A^T x$), on the other hand, reads one element of the input vector and one row of matrix A and both reads and writes the result vector y. Unless the machine on which these methods are implemented has three separate memory paths (e.g., Cray's vector computers), the memory traffic will then limit the performance. This is an important consideration for RISC-based architectures.

10.2.2.2 CDS Matrix-Vector Product

If the $n \times n$ matrix A is stored in CDS format, it is still possible to perform a matrix-vector product $y = Ax$ by either rows or columns, but this does not take advantage of the CDS format. The idea is to make a change in coordinates in the doubly nested loop. Replacing $j \to i + j$ we get

$$y_i \leftarrow y_i + a_{i,j} x_j \quad \Rightarrow \quad y_i \leftarrow y_i + a_{i,i+j} x_{i+j} \ .$$

With the index i in the inner loop we see that the expression $a_{i,i+j}$ accesses the jth diagonal of the matrix (where the main diagonal has number 0).

The algorithm will now have a doubly nested loop with the outer loop enumerating the diagonals `diag=-p,q` with p and q the (nonnegative) numbers of diagonals to the left and right of the main diagonal. The bounds for the inner loop follow from the requirement that

$$1 \le \mathtt{i}, \mathtt{i} + \mathtt{j} \le n.$$

The algorithm becomes

```
for i = 1, n
    y(i) = 0
end;
for diag = -diag_left, diag_right
    for loc = max(1,1-diag), min(n,n-diag)
        y(loc) = y(loc) + val(loc,diag) * x(loc+diag)
    end;
end;
```

The transpose matrix-vector product $y = A^T x$ is a minor variation of the algorithm above. Using the update formula

$$\begin{aligned} y_i &\leftarrow y_i + a_{i+j,i} x_j \\ &= y_i + a_{i+j,i+j-j} x_{i+j} \end{aligned}$$

we obtain

```
for i = 1, n
    y(i) = 0
end;
for diag = -diag_right, diag_left
    for loc = max(1,1-diag), min(n,n-diag)
        y(loc) = y(loc) + val(loc+diag, -diag) * x(loc+diag)
    end;
end;
```

The memory access for the CDS-based matrix-vector product $y = Ax$ (or $y = A^T x$) is three vectors per inner iteration. On the other hand, there is no indirect addressing, and the algorithm is vectorizable with vector lengths of essentially the matrix order n. Because of the regular data access, most machines can perform this algorithm efficiently by keeping three base registers and using simple offset addressing.

10.2.3 Fast Matrix-Vector Multiplication for Structured Matrices

Dense matrices that depend on $O(n)$ parameters arise often in practice. Examples of these matrices include

Vandermonde:

$$V = \begin{bmatrix} 1 & 1 & \dots & 1 \\ x_1 & x_2 & \dots & x_n \\ x_1^2 & x_2^2 & \dots & x_n^2 \\ & & \ddots & \\ x_1^{n-1} & x_2^{n-1} & \dots & x_n^{n-1} \end{bmatrix},$$

Toeplitz:

$$T = \begin{bmatrix} t_0 & t_{-1} & \dots & t_{2-n} & t_{1-n} \\ t_1 & t_0 & t_{-1} & & t_{2-n} \\ \vdots & t_1 & t_0 & \ddots & \vdots \\ t_{n-2} & & \ddots & \ddots & t_{-1} \\ t_{n-1} & t_{n-2} & \dots & t_1 & t_0 \end{bmatrix},$$

Hankel: $H = (H_{ij}) = (c_{i+j})$,

Cauchy: $C = (C_{ij}) = (\frac{1}{x_i - y_j})$, and others [257].

These matrices and their inverses can be multiplied by vectors in $O(n \log^k n)$ time, where $0 \le k \le 2$, instead of $O(n^2)$ or $O(n^3)$ time, depending on the structure. This is particularly useful for using iterative solvers with these matrices. The iterative solvers are also useful when solving systems of the form $(A + B)x = b$, where A and B are both structured but belong to different structured classes, or if A is structured and B is banded or sparse, etc. Iterative methods are also useful for solving block systems when the blocks are structured matrices but belong to different structure classes with some blocks also possibly being sparse. For example, the product

$$\begin{bmatrix} A & B \\ C & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} Ax + By \\ Cx \end{bmatrix},$$

where A is diagonal and B and C are Toeplitz can be formed in $O(n \log n)$ time.

Vandermonde and inverses of Vandermonde matrices can be multiplied by a vector in $O(n \log^2 n)$ time [196, 195]. The same is valid for the Vandermonde-like and inverses of Vandermonde–like matrices, where "-like" matrices are defined as in § 10.3.4. Matrix-vector multiplication for Toeplitz and Toeplitz-like matrices takes $O(n \log n)$ time using the fast Fourier transform (FFT) [198]. The Cauchy matrix-vector product Cz is equivalent to the evaluation of the function

$$\Phi(w) = \sum_{i=1}^{n} \frac{-z_i}{y_i - w}$$

at n different points $x_1, x_2, \dots, x_n$ and can be done in $O(n)$ time by using the fast multipole method [76, 205].

The rest of this section shows how the $O(n \log n)$ matrix-vector multiplication works for Toeplitz matrices [198]. For the other classes of structured matrices we refer the reader to [196] and [195].

A circulant matrix is a special Toeplitz matrix of the form

$$C_n = \begin{bmatrix} a_0 & a_{-1} & \dots & a_{2-n} & a_{1-n} \\ a_1 & a_0 & a_{-1} & & a_{2-n} \\ \vdots & a_1 & a_0 & \ddots & \vdots \\ a_{n-2} & & \ddots & \ddots & a_{-1} \\ a_{n-1} & a_{n-2} & \dots & a_1 & a_0 \end{bmatrix},$$

which has the property that $a_{-k} = a_{n-k}$ for $1 \le k \le n-1$. Circulant matrices are diagonalized by the FFT, i.e.,

$$C_n = F_n^* \cdot \mathrm{diag}(F_n a) \cdot F_n,$$

where $a = [a_0, a_1, \ldots, a_n]$ and F_n is the Fourier matrix $F_n(j,k) = \frac{1}{\sqrt{n}} e^{-(j-1)(k-1)2\pi\sqrt{-1}/n}$. It is a well-known fact that F_n is unitary and a product $F_n x$ can be formed in $O(n \log n)$ time [453]. A product $y = C_n x$ can also be formed in $O(n \log n)$ time by the following four steps:

(1) $f = F_n x$,
(2) $g = F_n a$,
(3) $z^T = [f_1 g_1, f_2 g_2, \ldots, f_n g_n]$,
(4) $y = F_n^* z$.

Now, if T_n is a Toeplitz matrix

$$T_n = \begin{bmatrix} t_0 & t_{-1} & \ldots & t_{2-n} & t_{1-n} \\ t_1 & t_0 & t_{-1} & & t_{2-n} \\ \vdots & t_1 & t_0 & \ddots & \vdots \\ t_{n-2} & & \ddots & \ddots & t_{-1} \\ t_{n-1} & t_{n-2} & \ldots & t_1 & t_0 \end{bmatrix},$$

then the product $T_n x$ can be computed in $O(n \log n)$ time by first embedding T_n into a $2n \times 2n$ circulant matrix C_{2n}, since

$$C_{2n} \cdot \begin{bmatrix} y \\ 0 \end{bmatrix} \equiv \begin{bmatrix} T_n & B_n \\ B_n & T_n \end{bmatrix} \cdot \begin{bmatrix} y \\ 0 \end{bmatrix} = \begin{bmatrix} T_n y \\ B_n y \end{bmatrix}$$

and

$$B_n = \begin{bmatrix} 0 & t_{n-1} & \ldots & t_2 & t_1 \\ t_{1-n} & 0 & t_{n-1} & & t_2 \\ \vdots & t_{1-n} & 0 & \ddots & \vdots \\ t_{-2} & & \ddots & \ddots & t_{n-1} \\ t_{-1} & t_{-2} & \ldots & t_{1-n} & 0 \end{bmatrix}.$$

10.3 A Brief Survey of Direct Linear Solvers

J. Demmel, P. Koev, and X. Li

Many of the most effective methods in this book, in particular those using shift-and-invert, require the solution of linear systems of the form $(A-\sigma I)x = b$ or $(A-\sigma I)^* x = b$, where σ is a *shift*, usually chosen to be an approximate eigenvalue of A. Much of the time in these methods can be spent solving these systems, so it is important to use the best methods available.

There are two basic approaches to solving $(A-\sigma I)x = b$: direct and iterative. Direct methods typically use variations on Gaussian elimination. They are effective even when σ is close to an eigenvalue and $A-\sigma I$ is nearly singular, which is when iterative methods

have most trouble converging. Section 10.4 discusses iterative solvers and when they can be effectively used in some eigenvalue algorithms, such as the Jacobi–Davidson method. This section considers only direct methods.

A direct method for solving $(A - \sigma I)x = b$ will consist of two steps:

1. Compute a *factorization* of $A - \sigma I$. This is the most expensive part.
2. Use the factorization to solve $(A - \sigma I)x = b$ or $(A - \sigma I)^* x = b$.

Step 1 usually costs much more than step 2 (e.g., $O(n^3)$ vs. $O(n^2)$ in the dense case). The factorization can be reused to solve $(A - \sigma I)x = b$ for many different right-hand sides b. The factorization only needs to be recomputed when the shift σ is changed. Fortunately, most iterative methods solve many linear systems with different right-hand sides and the same shift, so step 1 is performed many fewer times than step 2.

The choice of algorithm for steps 1 and 2 depends on both the *mathematical structure* of $A - \sigma I$ and the *storage structure* of A. By mathematical structure we most often mean whether $A - \sigma I$ is Hermitian or non-Hermitian, and definite or indefinite.

Hermitian and definite: In this case we compute the *Cholesky factorization* $A - \sigma I = \pm LL^T$, where L is a lower triangular matrix. The choice of sign depends on whether $A - \sigma I$ is positive definite $(+)$ or negative definite $(-)$. In the sparse case, we may instead compute $A - \sigma I = \pm PLL^*P^*$, where P is a permutation matrix.

Hermitian and indefinite: In this case we compute the *Hermitian indefinite factorization* $A - \sigma I = PLDL^*P^*$, where L is lower triangular, P is a permutation matrix, and D is block diagonal with 1 by 1 or 2 by 2 blocks, or perhaps tridiagonal.

Non-Hermitian: In this case we compute the *triangular factorization* $A - \sigma I = P_1 LUP_2$, where L is lower triangular, U is upper triangular, and P_1 and P_2 are permutations (sometimes one P_i is omitted).

There are many other structures and factorizations as well, such as when A is Toeplitz ($a_{i,j}$ depends only on $i - j$).

By storage structure we mean being dense, banded, sparse (in one of the formats described in §10.1), or structured (such as storing the first row and column of a Toeplitz matrix, which determines the other entries). A sparse non-Hermitian matrix may also be stored in a sparse Hermitian data structure, as in § 10.1.

There are specialized algorithms and software for many combinations of these mathematical and storage structures, and choosing the right algorithm can make orders of magnitude difference in solution time for large matrices. In this section we summarize the best algorithms and software available for these problems. It is often the case that the best algorithm in a research paper is not available as well designed and easily accessible software, and we shall concentrate on available software. We recommend software that is well maintained and in the public domain, or available at low cost, unless none is available.

We consider four cases: methods for dense matrices, methods for band matrices, methods for sparse matrices, and methods for structured matrices, such as Toeplitz matrices.

10.3.1 Direct Solvers for Dense Matrices

There are two reasons to consider an iterative method for dense matrices instead of using a transformation method:

1. If one only wants a few eigenvalues and/or eigenvectors near one or a few shifts σ, it may be cheaper to use shift-and-invert with an iterative scheme instead of a transformation method. This is more likely true with non-Hermitian matrices than Hermitian ones, for which the transformation methods are faster.

2. When a matrix is not very sparse, or not very large, a dense solver may be faster than a sparse solver.

The choice of dense solver depends on the mathematical structure of $A - \sigma I$ as follows:

$A - \sigma I$ is Hermitian definite. In this case Cholesky is the algorithm of choice. It is implemented in LAPACK computational routines `xPOTRF` to compute the factorization and `xPOTRS` to solve using the factorization (both are combined in LAPACK driver routine `xPOSVX`). Versions for packed data storage are also available (substitute `PP` for `PO`). Cholesky is implemented in analogous ScaLAPACK routines `PxPOTRF`, `PxPOTRS`, and `PxPOSVX`. The factorization is in MATLAB as `chol`.

$A - \sigma I$ is Hermitian indefinite. In this case Bunch–Kaufman is the algorithm of choice. It is implemented in real (complex) LAPACK computational routines `xSYTRF(xHETRF)` to compute the factorization and `xSYTRS(xHETRS)` to solve using the factorization (both are combined in LAPACK driver routine `xSYSVX(xHESVX)`). Versions for packed data storage are also available (substitute `SP(HP)` for `SY(HE)`). It is not available in ScaLAPACK or MATLAB.

$A - \sigma I$ is non-Hermitian. In this case Gaussian elimination is the algorithm of choice. It is implemented in LAPACK computational routines `xGETRF` to compute the factorization and `xGETRS` to solve using the factorization (both are combined in LAPACK driver routine `xGESVX`). It is implemented in analogous ScaLAPACK routines `PxGETRF`, `PxGETRS`, and `PxGESVX`. The factorization is in MATLAB as `lu`.

10.3.2 Direct Solvers for Band Matrices

This section is analogous to §10.3.1 for dense matrices:

$A - \sigma I$ is Hermitian definite. In this case Cholesky is the algorithm of choice. It is implemented in LAPACK computational routines `xPBTRF` to compute the factorization and `xPBTRS` to solve using the factorization (both are combined in LAPACK driver routine `xPBSVX`). Cholesky is implemented in analogous ScaLAPACK routines `PxPBTRF`, `PxPBTRS`, and `PxPBSV`.

$A - \sigma I$ is non-Hermitian. In this case Gaussian elimination is the algorithm of choice. It is implemented in LAPACK computational routines `xGBTRF` to compute the factorization and `xGBTRS` to solve using the factorization (both are combined in

LAPACK driver routine `xGBSVX`). It is implemented in analogous ScaLAPACK routines `PxGBTRF`, `PxGBTRS`, and `PxGBSV` with partial pivoting, and `PxDBTRF`, `PxDBTRS`, and `PxDBSV` without pivoting (which is risker but faster than using pivoting).

No routines exploiting Hermitian indefinite structure are available (because there is no simple bound on how much the bandwidth can grow because of pivoting). No band routines are in MATLAB. If $A - \sigma I$ is Hermitian and the bandwidth is narrow enough (such as tridiagonal), direct eigensolver methods from §4.2 should be used.

10.3.3 Direct Solvers for Sparse Matrices

Direct solvers for sparse matrices involve much more complicated algorithms than for dense matrices. The main complication is due to the need for efficiently handling the *fill-in* in the factors L and U. A typical sparse solver consists of four distinct steps as opposed to two in the dense case:

1. An ordering step that reorders the rows and columns such that the factors suffer little fill, or that the matrix has special structure, such as block-triangular form.
2. An analysis step or symbolic factorization that determines the nonzero structures of the factors and creates suitable data structures for the factors.
3. Numerical factorization that computes the L and U factors.
4. A solve step that performs forward and back substitution using the factors.

There is a vast variety of algorithms associated with each step. The review papers by Duff [137] (see also [135, Chapter 6]) and Heath, Ng, and Peyton [219] can serve as excellent references for various algorithms. Usually steps 1 and 2 involve only the graphs of the matrices, and hence only integer operations. Steps 3 and 4 involve floating-point operations. Step 3 is usually the most time-consuming part, whereas step 4 is about an order of magnitude faster. The algorithm used in step 1 is quite independent of that used in step 3. But the algorithm in step 2 is often closely related to that of step 3. In a solver for the simplest systems, i.e., symmetric and positive definite systems, the four steps can be well separated. For the most general unsymmetric systems, the solver may combine steps 2 and 3 (e.g. SuperLU) or even combine steps 1, 2 and 3 (e.g. UMFPACK) so that the numerical values also play a role in determining the elimination order.

In the past 10 years, many new algorithms and much new software have emerged which exploit new architectural features, such as memory hierarchy and parallelism. In Table 10.1, we compose a rather comprehensive list of sparse direct solvers. It is most convenient to organize the software in three categories: the software for serial machines, the software for SMPs, and the software for distributed memory parallel machines.

There is no single algorithm or software that is best for all types of linear systems. Some software is targeted for special matrices such as symmetric and positive definite, some is targeted for the most general cases. This is reflected in column 3 of the table, "Scope." Even for the same scope, the software may decide to use a particular algorithm or implementation technique, which is better for certain applications but not

Table 10.1: Software to solve sparse linear systems using direct methods.

Code	Technique	Scope	Contact	
Serial platforms				
MA27	Multifrontal	Sym	HSL	[140]
MA41	Multifrontal	Sym-pat	HSL	[6]
MA42	Frontal	Unsym	HSL	[144]
MA47	Multifrontal	Sym	HSL	[141]
MA48	Right-looking	Unsym	HSL	[142]
SPARSE	Right-looking	Unsym	Kundert	[281]
SPARSPAK	Left-looking	SPD	George	[191]
SPOOLES	Left-looking	Sym and Sym-pat	Ashcraft	[21]
SuperLLT	Left-looking	SPD	Ng	[339]
SuperLU	Left-looking	Unsym	Li	[126]
UMFPACK	Multifrontal	Unsym	Davis	[103]
Shared memory parallel machines				
Cholesky	Left-looking	SPD	Rothberg	[405]
DMF	Multifrontal	Sym	Lucas	[308]
MA41	Multifrontal	Sym-pat	HSL	[7]
PanelLLT	Left-looking	SPD	Ng	[211]
PARASPAR	Right-looking	Unsym	Zlatev	[471]
PARDISO	Left-right looking	Sym-pat	Schenk	[395]
SPOOLES	Left-looking	Sym and Sym-pat	Ashcraft	[21]
SuperLU_MT	Left-looking	Unsym	Li	[127]
Distributed memory parallel machines				
CAPSS	Multifrontal	SPD	Raghavan	[220]
DMF	Multifrontal	Sym	Lucas	[308]
MUMPS	Multifrontal	Sym and Sym-pat	Amestoy	[8]
PaStiX	Left-right looking*	SPD	CEA	[224]
PSPASES	Multifrontal	SPD	Gupta	[210]
SPOOLES	Left-looking	Sym and Sym-pat	Ashcraft	[21]
SuperLU_DIST	Right-looking	Unsym	Li	[306]
S+	Right-looking†	Unsym	Yang	[182]

* In spite of the title of the paper

† Uses QR storage to statically accommodate any LU fill-in

Abbreviations used in the table: SPD = symmetric and positive definite; Sym = symmetric and may be indefinite; Sym-pat = symmetric nonzero pattern but unsymmetric values; Unsym = unsymmetric; HSL = Harwell Subroutine Library: `http://www.cse.clrc.ac.uk/Activity/HSL`

for others. In column 2, "Technique," we give a high-level algorithmic description. For a review of the distinctions between left-looking, right-looking, and multifrontal and their implications on performance, we refer the reader to the papers by Heath, Ng, and Peyton [219] and Rothberg [370]. Sometimes the best (or only) software is not in the public domain, but available commercially or in research prototypes. This is reflected in column 4, "Contact," which could be the name of a company or the name of the author of the research code.

In the context of shift-and-invert spectral transformation (SI) for eigensystem analysis, we need to factorize $A-\sigma I$, where A is fixed. Therefore, the nonzero structure of $A-\sigma I$ is fixed. Furthermore, for the same shift σ, it is common to solve many systems with the same matrix and different right-hand sides (in which case the solve cost can be comparable to factorization cost). It is reasonable to spend a little more time in steps 1 and 2 to speed up steps 3 and 4. That is, one can try different ordering schemes and estimate the costs of numerical factorization and solutions based on symbolic factorization, and use the best ordering. For instance, in computing the SVD, one has the choice between shift-and-invert on AA^*, A^*A, and $\left[\begin{smallmatrix} 0 & A \\ A^* & 0 \end{smallmatrix}\right]$, all of which can have rather different factorization costs, as discussed in Chapter 6.

Some solvers have the ordering schemes built in, but others do not. It is also possible that the built-in ordering schemes are not the best for the target applications. It is sometimes better to substitute an external ordering scheme for the built-in one. Many solvers provide well-defined interfaces so that the user can make this substitution easily. One should read the solver documentation to see how to do this, as well as to find out the recommended ordering methods.

10.3.4 Direct Solvers for Structured Matrices

The linear system of equations involving the structured matrices described in §10.2.3 have the attractive property of being solvable in $O(n^2)$ time instead of $O(n^3)$.

The common property that makes it possible to solve such systems fast (in $O(n^2)$ time) is the fact that they have a *low displacement rank* [257].

Displacement rank may be defined as follows: For given matrices A and F and a matrix T we define the operator

$$\Delta_{A,F}(T) = A \cdot T - T \cdot F$$

and denote $\alpha = \operatorname{rank}(\Delta_{A,F}(T))$. We say that α is the "displacement rank" of T with respect to the operator $\Delta_{A,F}$. If α is small (usually $O(1)$) we say that the matrix T has a "low displacement rank." Since $\operatorname{rank}(\Delta_{A,F}(T)) = \alpha$ there exist $n \times \alpha$ matrices G and B such that $\Delta_{A,F}(T) = G \cdot B^T$. The matrices G and B are called *generators* of T. We will refer to matrices with low displacement rank as having *displacement structure*.

For an example for a Cauchy matrix $C = (C_{ij}) = (1/(x_i - y_j))$ we have

$$\Delta_{\operatorname{diag}(x),\operatorname{diag}(y)}(C) = \operatorname{diag}(x) \cdot C - C \cdot \operatorname{diag}(y) = (1, 1, \ldots, 1)^T \cdot (1, 1, \ldots, 1).$$

Therefore, a Cauchy matrix has displacement rank 1 and generators $G = B = (1, 1, \ldots, 1)^T$.

A matrix with low displacement rank α (not necessarily equal to 1) with respect to the operator $\Delta_{\operatorname{diag}(x),\operatorname{diag}(y)}$ is called Cauchy-like. In a similar way one defines Vandermonde-like matrices, Toeplitz-like matrices, etc. These higher displacement rank systems are solvable in $O(\alpha n^2)$ time.

Fast algorithms exploit the displacement equation in order to produce an LU factorization of the matrix F. The algorithms work on the generators of a matrix instead of the matrix itself, which permits them to go much faster.

The block-Toeplitz matrices and matrices of type $T^T T$, where T is Toeplitz, also have a low displacement rank [257].

If A and B have displacement structure, then systems of type $(A+\sigma B)x = b$ can be solved using the fast low displacement rank methods only if $A + \sigma B$ has displacement

structure as well (with respect to a possibly different displacement operator). For example, if A is Hankel and $B = I$ then $A + \sigma B$ is Toeplitz-plus-Hankel, which also has displacement structure. If $A + \sigma B$ does not have displacement structure then one would be restricted to using zero-shifts $\sigma = 0$ or fast iterative methods for which only a fast matrix-vector product is needed [257].

Some of the methods impose additional restrictions on the matrices to be symmetric (Toeplitz) or positive definite. This may result in additional restrictions on the choice of shifts available.

Special attention should be exercised when using fast algorithms for structured matrices since the speed often comes at the price of accuracy. Some displacement-rank algorithms simulate Gaussian elimination (with partial, complete, or no pivoting) by working on the generators instead of on the matrices themselves [193]. Even so, these algorithms need not have the same numerical properties as Gaussian Elimination because of the possibility of "generator growth" [257], which is uncommon, but can appear even for very well conditioned matrices.

There are many methods to stabilize the fast algorithms including the problem of generator growth [257, p. 111], and despite the occasional instabilities these methods are very attractive.

The fast algorithms are very well described in the literature, but reliable software libraries are not available. One should also be aware of the constants hidden in the $O(n^2)$ notation and that those constants make the fast methods faster than the traditional $O(n^3)$ methods only for n sufficiently large (usually at least a few hundred, depending on the structure). The traditional $O(n^3)$ algorithms are often optimized to use BLAS 3 and use the computer's memory hierarchy efficiently in order for the code to run near the full speed of the processor (see §10.2.1). The fast algorithms can usually only use BLAS 2 and it may be difficult or impossible to optimize the fast algorithms to make efficient use of the computer's cache and fast memory. This means that even if a fast $O(n^2)$ algorithm and a slow $O(n^3)$ algorithm perform the same number of floating point operations, the "slow" algorithm is very likely to be faster because of these optimization issues.

We should also mention the Bjorck–Pereyra-type algorithms for solving Vandermonde and three-term Vandermonde systems [51, 230]. These $O(n^2)$ methods have remarkable numerical properties. Under some additional restrictions on the order of the nodes in the Vandermonde matrix and the sign pattern of the right-hand side, it is possible to solve those systems to the full machine precision.

10.4 A Brief Survey of Iterative Linear Solvers

H. van der Vorst

In the context of eigenproblems it may be necessary to solve a linear system, for instance, in the following situations:

- The Jacobi–Davidson methods require the (approximate) solution of a linear correction equation.
- Inexact methods, like those discussed in Chapter 11, are based on an approximate shift-and-invert step, for which a linear system needs to be approximately solved.

- Given a good approximation for an eigenvalue, the corresponding left or right eigenvector can be computed from a linear system with the shifted matrix.

In all these cases, one has to solve a linear system with a shifted matrix $A - \theta I$, sometimes embedded in projections as in the Jacobi–Davidson methods. If such systems have to be solved accurately, then direct (sparse) solvers may be considered first. Often one has to solve more than one system with the same shifted matrix, which helps to amortize large costs involved in making a sparse LU decomposition. If the systems need not be solved accurately, or if direct solution methods are too expensive, then iterative methods may be considered. A set of popular iteration methods has been described in *Templates for the Solution of Linear Systems* [41]. Before we present a short overview, we warn the reader that the linear systems related to eigenproblems usually have an indefinite matrix, because of the involved shift. If the shift is near an exterior eigenvalue, then this is not necessarily an obstacle in all cases, but there will certainly be convergence problems for interior shifts. Also, when the shift is very close to an eigenvalue, for instance, if one wants to determine a left or right eigenvalue, then one should realize that iterative methods have great difficulty solving almost singular systems. This holds in particular for the situations of interest, where one is interested in the (almost) singular directions, as is the case in inverse iteration for left and right eigenvectors. It is commonly accepted that iterative methods need efficient preconditioners in order to be attractive. This is in particular the case for shifted matrices. Unfortunately, the construction of effective preconditioners for indefinite matrices is slippery ground to treat upon. See Chapter 11 for more on this.

The currently most popular iterative methods belong to the class of Krylov subspace methods. These methods construct approximations for the solution from the so-called Krylov subspace. The Krylov subspace $\mathcal{K}^i(A; r_0)$ of dimension i, associated with a linear system $Ax = b$ (where A and b may be the preconditioned values, if preconditioning is included) for a starting vector x_0 with residual vector $r_0 = b - Ax_0$ is defined as the subspace spanned by the vectors $\{r_0, Ar_0, A^2r_0, \ldots, A^{i-1}r_0\}$.

The different methods can be classified as follows:

(a) If A is symmetric positive definite, then the *conjugate gradient* method [226] generates, using two-term recurrences, the x_i for which $(x - x_i, A(x - x_i))$ (the so-called A-norm or energy norm) is minimized over all vectors in the current Krylov subspace $\mathcal{K}^i(A; r_0)$.

(b) If A is only symmetric but not positive definite, then the *Lanczos* [286] and the *MINRES* methods [350] may be considered. In MINRES, the $x_i \in \mathcal{K}^i(A; r_0)$ is determined for which the 2-norm of the residual ($\|b - Ax_i\|_2$) is minimized, while the Lanczos method leads to an x_i for which $b - Ax_i$ is perpendicular to the Krylov subspace.

(c) If A is unsymmetric, it is in general not possible to determine an optimal $x_i \in K^i(A, r_0)$ with short recurrences. This was proved in [163]. However, with short recurrences as in conjugate gradients (and MINRES), we can compute the $x_i \in \mathcal{K}^i(A; r_0)$, for which $b - Ax_i \perp \mathcal{K}^i(A^T; s_0)$ (usually, one selects $s_0 = r_0$). This leads to the *bi-conjugate gradient method* [169]. A clever variant is quasi-minimal residual (QMR) [179], which has smoother convergence behavior and is more robust than bi-conjugate gradients.

(d) If A is unsymmetric, then we can compute the $x_i \in \mathcal{K}^i(A, r_0)$, for which the residual is minimized in the Euclidean norm. This is done by the GMRES method [389]. This requires i inner products at the ith iteration step, as well as i vector updates, which means that the iteration costs, that come in addition to operations with A, grow linearly with i.

(e) The operations with A^T in the bi-conjugate gradient method can be replaced by operations with A itself, by using the observation that $\langle x, A^T y\rangle$ equals $\langle Ax, y\rangle$, where $\langle \ldots \rangle$ represents the inner-product computation. Since the function of the multiplications by A^T in bi-conjugate gradient serves only to maintain the dual space to which residuals are orthogonalized, the replacement of this operator by A allows us to expand the Krylov subspace and to find better approximations for virtually the same costs per iteration as for bi-conjugate gradients. This leads to so-called hybrid methods such as conjugate gradients squared [418], Bi-CGSTAB [445], Bi-CGSTAB(ℓ) [409], TFQMR [174], and hybrids of QMR [78].

(f) For indefinite systems it may also be effective to apply the conjugate gradient method for the normal equations $A^T Ax = A^T b$. Carrying this out in a straight-forward manner may lead to numerical instabilities, because $A^T A$ has a squared condition number. A clever and robust implementation is provided in least squares QR [351].

Many of these methods have been described in [41], and software for them is available. See this book's homepage, ETHOME, for guidelines on how to obtain the software. For some methods, descriptions in templates style have been given in [135]. That book also contains an overview on various preconditioning approaches.

10.5 Parallelism

J. Dongarra and X. Li

In this section we discuss aspects of parallelism in the iterative methods discussed in this book. Since the iterative methods share most of their computational kernels we will discuss these independent of the method. The basic time-consuming kernels of iterative schemes are:

- inner products;
- vector updates;
- matrix-vector products, e.g., $Ap^{(i)}$ (for some methods also $A^T p^{(i)}$);
- solvers for $(A - \sigma B)x = b$.

Inner Products. The computation of an inner product of two vectors can be easily parallelized; each processor computes the inner product of corresponding segments of each vector (local inner products (LIPs)). On distributed memory machines the LIPs then have to be sent to other processors to be combined for the global inner product. This can be done either with an all-to-all send where every processor performs the summation of the LIPs, or by a global accumulation in one processor, followed by a broadcast of the final result. Clearly, this step requires communication.

For shared memory machines, the accumulation of LIPs can be implemented as a critical section where all processors add their local result in turn to the global result, or as a piece of serial code, where one processor performs the summations.

Vector Updates. Vector updates are trivially parallelizable: each processor updates its own segment.

Matrix-Vector Products. The matrix-vector products are often easily parallelized on shared memory machines by splitting the matrix in strips of rows corresponding to the vector segments. Each processor then computes the matrix-vector product of one strip. For distributed memory machines, there may be a problem if each processor has only a segment of the vector in its memory. Depending on the bandwidth of the matrix, we may need communication for other elements of the vector, which may lead to communication bottlenecks. However, many sparse matrix problems arise from a network in which only nearby nodes are connected. For example, matrices stemming from finite difference or finite element problems typically involve only local connections: matrix element $a_{i,j}$ is nonzero only if variables i and j are physically close. In such a case, it seems natural to subdivide the network, or grid, into suitable blocks and to distribute them over the processors. When computing $Ap^{(i)}$, each processor requires the values of $p^{(i)}$ at some nodes in neighboring blocks. If the number of connections to these neighboring blocks is small compared to the number of internal nodes, then the communication time can be overlapped with computational work.

More recently, *graph partitioning* has been used as a powerful tool to deal with the general problems with nonlocal connections. Consider $y \leftarrow Ax$ and the undirected graph G of the (symmetric) matrix A. Assume vertex i stores x_i, y_i, and the nonzero $a_{i,j}$ for all j. Let vertex i represent the job of computing $y_i = \sum_j a_{i,j}x_j$. We can assign weight of vertex i to be the number of operations for computing y_i, and the weight of edge (i, j) to be 1, which represents the cost of sending x_j to vertex i if vertices i and j were mapped on different processors. A good graph partitioning heuristic would partition G into P subsets of vertices corresponding to P processors, such that:

- the sums of vertex weights are approximately equal among the subsets,
- the sum of edge cuts crossing the subsets is minimized.

This ensures a good load balance and minimizes communication. Good graph partitioning software includes Chaco [223] and METIS [259] (also ParMETIS, the parallel version).

After partitioning, further optimizations can be performed to reduce communication. For example, if a subset of vertices contains several edges to another subset, the corresponding elements of the vector can be packed into one message, so that each processor sends no more than one message to another processor. For more detailed discussions on implementation aspects for distributed memory systems, see De Sturler and van der Vorst [111, 112] and Pommerell [369].

High-quality parallel algorithms for distributed memory machines are implemented in software packages such as Aztec [236] and PETSc [38]. The software can be obtained via the book's homepage, ETHOME.

Solvers. In addition to the three kernels above, the iterative methods using shift-and-invert often require the direct solutions of the linear systems. Both matrix factorization and triangular solve involve much more complicated parallel algorithms, especially on massively parallel machines. There has been a large amount of research activity in this area. Many state-of-the-art parallel algorithms for dense and band matrices are implemented in ScaLAPACK (see § 10.3), and those for sparse matrices are implemented in the software packages surveyed in Table 10.1.