

Chapter 2

Fundamental Kernels

In this chapter we discuss the fundamental operations, that are the building blocks of dense and sparse matrix computations. They are termed kernels because in most cases they account for most of the computational effort. Because of this, their implementation directly impacts the overall efficiency of the computation. They occur often at the lowest level where parallelism is expressed.

Most basic kernels are of the form $C = C + AB$, where A , B and C can be matrix, vector and possibly scalar operands of appropriate dimensions. For dense matrices, the community has converged into a standard application programming interface, termed *Basic Linear Algebra Subroutines* (BLAS) that have specific syntax and semantics. The set is organized into three separate sets of instructions. The first part of this chapter describes these sets. It then considers several basic sparse matrix operations that are essential for the implementation of algorithms presented in future chapters. In this chapter we frequently make explicit reference to communication costs, on account of the well known growing discrepancy, in the performance characteristics of computer systems, between the rate of performing computations (typically measured by a base unit of the form flops per second) and the rate of moving data (typically measured by a base unit of the form words per second).

2.1 Vector Operations

Operations on vectors are known as Level_1 Basic Linear Algebraic Subroutines (BLAS1) [1]. The two most frequent vector operations are the `_AXPY` and the `_DOT`:

- `_AXPY`: given $x, y \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$, the instruction updates vector y by:
 $y = y + \alpha x$.
- `_DOT`: given $x, y \in \mathbb{R}^n$, the instruction computes the inner product of the two vectors: $s = x^\top y$.

A common feature of these instructions is that minimal number of data that needs to be read (loaded) into memory and then stored back in order for the operation to take place is $O(n)$. Moreover, the number of computations required on a uniprocessor is also $O(n)$. Therefore, the ratio of instructions to load from and store to memory relative to purely arithmetic operations is $O(1)$.

With $p = n$ processors, the `_AXPY` primitive requires 2 steps which yields a perfect speedup, n . The `_DOT` primitive involves a reduction with the sum of n numbers to obtain a scalar. We assume temporarily, for the sake of clarity, that $n = 2^m$. At the first step, each processor computes the product of two components, and the result can be expressed as the vector $(s_i^{(0)})_{1:n}$. This computation is then followed by m steps such that at each step k the vector $(s_i^{(k-1)})_{1:2^{m-k+1}}$ is transformed into the vector $(s_i^{(k)})_{1:2^{m-k}}$ by computing in parallel $s_i^{(k)} = s_{2i-1}^{(k-1)} + s_{2i}^{(k-1)}$, for $i = 1, \dots, 2^{m-k}$, with the final result being the scalar $s_1^{(m)}$. Therefore, the inner product consumes $T_p = m + 1 = (1 + \log n)$ steps, with a speedup of $S_p = 2n/(1 + \log n)$ and an efficiency of $E_p = 2/(1 + \log n)$.

On vector processors, these procedures can obtain high performance, especially for the `_AXPY` primitive which allows chaining of the pipelines for multiplication and addition.

Implementing these instructions on parallel architectures is not a difficult task. It is realized by splitting the vectors in slices of the same length, with each processor performing the operation on its own subvectors. For the `_DOT` operation, there is an additional summation of all the partial results to obtain the final scalar. Following that, this result has to be broadcast to all the processors. These final steps entail extra costs for data movement and synchronization, especially for computer systems with distributed memory and a large number of processors.

We analyze this issue in greater detail next, departing on this occasion from the assumptions made in Chap. 1 and taking explicitly into account the communication costs in evaluating T_p . The message is that inner products are harmful to parallel performance of many algorithms.

Inner Products Inhibit Parallel Scalability

A major part of this book deals with parallel algorithms for solving large sparse linear systems of equations using preconditioned iterative schemes. The most effective classes of these methods are dominated by a combination of a “global” inner product, that is applied on vectors distributed across all the processors, followed by fan-out operations. As we will show, the overheads involved in such operations cause inefficiency and less than optimal speedup.

To illustrate this point, we consider such a combination in the form of the following primitive for vectors u, v, w of size n that appears often in many computations:

$$w = w - (u^\top v)u. \quad (2.1)$$

We assume that the vectors are stored in a consistent way to perform the operations on the components (each processor stores slices of components of the two vectors

with identical indices). The `_DOT` primitive involves a reduction and therefore an all-to-one (fan-in) communication. Since the result of a dot product is usually needed by all processors in the sequel, the communication actually becomes an all-to-all (fan-out) procedure.

To evaluate the weak scalability on p processors (see Definition 1.1) by taking communication into account (as mentioned earlier, we depart here from our usual definition of \mathbf{T}_p), let us assume that $n = pq$. The number of steps required by the primitive (2.1) is $4q - 1$. Assuming no overlap between communication and computation, the cost on p processors, $\mathbf{T}_p(pq)$, is the sum of the computational and communication costs: $\mathbf{T}_p^{\text{cal}}(pq)$ and $\mathbf{T}_p^{\text{com}}(pq)$, respectively. For the all-to-all communication, $\mathbf{T}_p^{\text{com}}(pq)$ is given by: $\mathbf{T}_p^{\text{com}}(pq) = K p^\gamma$ in which $1 < \gamma \leq 2$, with the constant K depending on the interconnection network technology. The computational load, which is well balanced, is given by $\mathbf{T}_p^{\text{cal}}(pq) = 4q - 1$, resulting in $\mathbf{T}_p(pq) = (4q - 1) + K p^\gamma$, which increases with the number of processors. In fact, once $\mathbf{T}_p^{\text{com}}(pq)$ dominates $\mathbf{T}_p^{\text{cal}}(pq)$, the total cost $\mathbf{T}_p(pq)$ increases almost quadratically with the number of processors.

This fact is of crucial importance in parallel implementation of inner products. It makes clear that an important goal of a designer of parallel algorithms on distributed memory architectures is to avoid distributed `_DOT` primitives as they are detrimental to parallel scalability. Moreover, because of the frequent occurrence and prominence of inner products in most numerical linear algebra algorithms, the aforementioned deleterious effects on the `_DOT` performance can be far reaching.

2.2 Higher Level BLAS

In order to increase efficiency, vector operations are often packed into a global task of higher level. This occurs for the multiplication of a matrix by a vector which in a code is usually expressed by a doubly nested loop. Classic kernels of this type are gathered into the set known as Level_2 Basic Linear Algebraic Subroutines (BLAS2) [2]. The most common operations of this type, assuming general matrices, are

- `_GEMV`: given $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$ this performs the matrix-vector multiplication and accumulate $y = y + Ax$. It is also possible to multiply (row) vector by matrix, scale the result before accumulating.
- `_TRSV`: given $b \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$ upper or lower triangular, this solves the triangular system $Ax = b$.
- `_GER`: given scalar α , $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$, this performs the rank-one update $A = A + \alpha xy^\top$.

A common feature of these instructions is that the smallest number of data that needs to be read into memory and then stored back in order for the operation to take place when $m = n$ is $O(n^2)$, arithmetic operations is $O(1)$. Moreover, the number of computations required on a uniprocessor is also $O(n^2)$. Therefore, the ratio of instructions to load from and store to memory relative to purely arithmetic ones is

$O(1)$. Typically, the constants involved are a little smaller than those for the BLAS1 instructions. On the other hand, of far more interest in terms of efficiency.

Although of interest, the efficiencies realized by these kernels are easily surpassed by those of (BLAS3) [3], where one additional loop level is considered, e.g. matrix multiplication, and rank- k updates ($k > 1$). The next section of this chapter is devoted to matrix-matrix multiplications.

The set of BLAS is designed for a uniprocessor and used in parallel programs in the sequential mode. Thus, an efficient implementation of the BLAS is of the utmost importance to enable high performance. Versions of the BLAS that are especially fine-tuned for certain types of processors are available (e.g. the Intel Math Kernel Library [4] or the open source set GOTOBLAS [5]). Alternately, one can create a parametrized BLAS set which can be tuned on any processor by an automatic code optimizer, e.g. ATLAS [6, 7]. Yet, it is hard to outperform well designed methods that are based on accurate architectural models and domain expertise; cf. [8, 9].

2.2.1 Dense Matrix Multiplication

Given matrices A , B and C of sizes $n_1 \times n_2$, $n_2 \times n_3$ and $n_1 \times n_3$ respectively, the general operation, denoted by `_GEMM`, is $C = C + AB$.

Properties of this primitive include:

- The computation involves three nested loops that may be permuted and split; such a feature provides great flexibility in adapting the computation for vector and/or parallel architectures.
- High performance implementations are based on the potential for high data locality that is evident from the relation between the lower bound on the number of data moved ($O(n^2)$) to arithmetic operations, $O(n^3)$ for the classical schemes and $O(n^{2+\mu})$ for some $\mu > 0$ for the “superfast” schemes described later in this section.

Hence, in dense matrix multiplication, it is possible to reuse data stored in cache memory.

Because of these advantages of dense matrix multiplication over lower level BLAS, there has been a concerted effort by researchers for a long time now (see e.g. [10]) to (re)formulate many algorithms in scientific computing to be based on dense matrix multiplications (such as `_GEMM` and variants).

A Data Management Scheme Dense Matrix Multiplications

We discuss an implementation strategy for `_GEMM`:

$$C = C + AB. \tag{2.2}$$

We adopt our discussion from [11], where the authors consider the situation of a cluster of p processors with a common cache and count loads and stores in their evaluation. We simplify that discussion and outline the implementation strategy for a uniprocessor equipped with a cache memory that is characterized by fast access. The purpose is to highlight some critical design decisions that will have to be faced by the sequential as well as by the parallel algorithm designer. We assume that reading one floating-point word of the type used in the multiplication from cache can be accomplished in one clock period. Since the storage capacity of a cache memory is limited, the goal of a code developer is to reuse, as much as possible, data stored in the cache memory.

Let M be the storage capacity of the cache memory and let us assume that matrices A , B and C are, respectively, $n_1 \times n_2$, $n_2 \times n_3$ and $n_1 \times n_3$ matrices. Partitioning these matrices into blocks of sizes $m_1 \times m_2$, $m_2 \times m_3$ and $m_1 \times m_3$, respectively, where $n_i = m_i k_i$ for all $i = 1, 2, 3$, our goal is then to estimate the block sizes m_i which maximize data reuse under the cache size constraint.

Instruction (2.2) can be expressed as the nested loop,

```

do  $i = 1 : k_1$ ,
  do  $k = 1 : k_2$ ,
    do  $j = 1 : k_3$ ,
       $C_{ij} = C_{ij} + A_{ik} \times B_{kj}$ ;
    end
  end
end

```

where C_{ij} , A_{ik} and B_{kj} are, respectively, blocks of C , A and B , with subscripts denoting here the appropriate block indices. The innermost loop, refers to the identical block A_{ik} in all its iterations. To put it in cache, its dimensions must satisfy $m_1 m_2 \leq M$. Actually, the blocks C_{ij} and B_{kj} must also reside in the cache and the condition becomes

$$m_1 m_3 + m_1 m_2 + m_2 m_3 \leq M. \quad (2.3)$$

Further, since the blocks are obviously smaller than the original matrices, we need the additional constraints:

$$1 \leq m_i \leq n_i \text{ for } i = 1, 2, 3. \quad (2.4)$$

Evaluating the volume of the data moves using the number of data loads necessary for the whole procedure and assuming that the constraints (2.3) and (2.4) are satisfied, we observe that

- all the blocks of the matrix A are loaded only once;
- the blocks of the matrix B are loaded k_1 times;
- the blocks of the matrix C are loaded k_2 times.

Thus the total amount of loads is given by:

$$L = n_1 n_2 + n_1 n_2 n_3 \left(\frac{1}{m_1} + \frac{1}{m_2} \right). \quad (2.5)$$

Choosing $m_3 = 1$, and hence $k_3 = n_3$, (the multiplications of the blocks are performed by columns) and neglecting for simplicity the necessary storage of the columns of the blocks C_{ij} and B_{kj} , the values of m_1 and m_2 , which minimize $\frac{1}{m_1} + \frac{1}{m_2}$ under the previous constraints are obtained as follows:

```

if  $n_1 n_2 \leq M$  then
   $m_1 = n_1$  and  $m_2 = n_2$ ;
else if  $n_2 \leq \sqrt{M}$  then
   $m_1 = \frac{M}{n_2}$  and  $m_2 = n_2$ ;
else if  $n_1 \leq \sqrt{M}$  then
   $m_1 = n_1$  and  $m_2 = \frac{M}{n_1}$ ;
else
   $m_1 = \sqrt{M}$  and  $m_2 = \sqrt{M}$ ;
end if

```

In practice, M should be slightly smaller than the total cache volume to allow for storing the neglected vectors. With this parameter adjustment, at the innermost level, the block multiplication involves $2m_1 m_2$ operations and $m_1 + m_2$ loads as long as A_{ik} resides in the cache. This indicates why the matrix multiplication can be a compute bound program.

The reveal the important decisions that need to be made by the code developer, and leads to a scheme that is very similar to parallel multiplication.

In [11], the authors consider the situation of a cluster of p processors with a common cache and count loads and stores in their evaluation. However, the final decision tree is similar to the one presented here.

2.2.2 Lowering Complexity via the Strassen Algorithm

The classical multiplication algorithms implementing the operation (2.2) for dense matrices use $2n^3$ operations. We next describe the scheme proposed by Strassen which reduces the number of operations in the procedure [12]: assuming that n is even, the operands can be decomposed in 2×2 matrices of $\frac{n}{2} \times \frac{n}{2}$ blocks:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \quad (2.6)$$

Then, the multiplication can be performed by the following operations on the blocks

$$\begin{aligned}
P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), & C_{11} &= P_1 + P_4 - P_5 + P_7, \\
P_2 &= (A_{21} + A_{22})B_{11}, & C_{12} &= P_3 + P_5, \\
P_3 &= A_{11}(B_{12} - B_{22}), & C_{21} &= P_2 + P_4, \\
P_4 &= A_{22}(B_{21} - B_{11}), & C_{22} &= P_1 + P_3 - P_2 + P_6. \\
P_5 &= (A_{11} + A_{12})B_{22}, \\
P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}), \\
P_7 &= (A_{12} - A_{22})(B_{21} + B_{22}),
\end{aligned} \tag{2.7}$$

The computation of P_k and C_{ij} is referred as one Strassen step. This procedure involves 7 block multiplications and 18 block additions of blocks, instead of 8 block multiplications and 4 block additions as the case in the classical algorithm. Since the complexity of the multiplications is $O(n^3)$ whereas for an addition it is only $O(n^2)$, the Strassen approach is beneficial for large enough n . This approach was improved in [13] by the following sequence of 7 block multiplications and 15 block additions. It is implemented in the so-called Strassen-Winograd procedure (as expressed in [14]):

$$\begin{aligned}
T_0 &= A_{11}, & S_0 &= B_{11}, & Q_0 &= T_0 S_0, & U_1 &= Q_0 + Q_3, \\
T_1 &= A_{12}, & S_1 &= B_{21}, & Q_1 &= T_1 S_1, & U_2 &= U_1 + Q_4, \\
T_2 &= A_{21} + A_{22}, & S_2 &= B_{12} + B_{11}, & Q_2 &= T_2 S_2, & U_3 &= U_1 + Q_2, \\
T_3 &= T_2 - A_{12}, & S_3 &= B_{22} - S_2, & Q_3 &= T_3 S_3, & C_{11} &= Q_0 + Q_1, \\
T_4 &= A_{11} - A_{12}, & S_4 &= B_{22} - B_{12}, & Q_4 &= T_4 S_4, & C_{12} &= U_3 + Q_5, \\
T_5 &= A_{12} + T_3, & S_5 &= B_{22}, & Q_5 &= T_5 S_5, & C_{21} &= U_2 - Q_6, \\
T_6 &= A_{22}, & S_6 &= S_3 - B_{21}, & Q_6 &= T_6 S_6, & C_{22} &= U_2 + Q_2.
\end{aligned} \tag{2.8}$$

Clearly, (2.7) and (2.8) are still valid for rectangular blocks. If $n = 2^{\gamma}$, the approach can be repeated for implementing the multiplications of the blocks. If it is recursively applied up to 2×2 blocks, the total complexity of the process becomes $O(n^{\omega_0})$, where $\omega_0 = \log 7$. More generally, if the process is iteratively applied until we get blocks of order $m \leq n_0$, the total number of operations is

$$T(n) = c_s n^{\omega_0} - 5n^2, \tag{2.9}$$

with $c_s = (2n_0 + 4)/n_0^{\omega_0-2}$, which achieves its minimum for $n_0 = 8$; cf. [14].

The numerical stability of the above methods has been considered by several authors. In [15], it is shown that the rounding errors in the Strassen algorithm can be worse than those in the classical algorithm for multiplying two matrices, with the situation somewhat worse in Winograd's algorithm. However, ref. [15] indicates that it is possible to get a fast and stable version of _GEMM by incorporating in it steps from the Strassen or Winograd-type algorithms.

Both the Strassen algorithm (2.7), and the Winograd version (2.8), can be implemented on parallel architectures. In particular, the seven block multiplications are independent, as well as most of the block additions. Moreover, each of these operations has yet another inner level of parallelism.

A parallel implementation must allow for the recursive application of several Strassen steps while maintaining good data locality. The Communication-Avoiding Parallel Strassen (CAPS) algorithm, proposed in [14, 16], achieves this aim; cf. [14, 16]. In CAPS, the Strassen steps are implemented by combining two strategies: all the processors cooperate in computing the blocks P_k and C_{ij} whenever the local memories are not large enough to store the blocks. The remaining Strassen steps consist of block operations that are executed independently on seven sets of processors. The latter minimizes the communications but needs extra memory. CAPS is asymptotically optimal with respect to computational cost and data communication.

Theorem 2.1 ([14]) *CAPS has computational cost $\Theta(n^{\omega_0}/p)$ and requires bandwidth $\Theta(\max\{(n^{\omega_0}/p M^{\omega_0/2}) \log p, \log p\})$, assuming p processors, each with local memory of size M words.*

Experiments in [17] show that CAPS uses less communication than some communication optimal classical algorithms and much less than previous implementations of the Strassen algorithm. As a result, it can outperform both classical algorithms for large sized problems, because it requires fewer operations, as well as for small problems, because of lower communication costs.

2.2.3 Accelerating the Multiplication of Complex Matrices

Savings may be realized in multiplying two complex matrices, e.g. see [18]. Let $A = A_1 + iA_2$ and $B = B_1 + iB_2$ two complex matrices where $A_j, B_j \in \mathbb{R}^{n \times n}$ for $j = 1, 2$. The real and imaginary parts C_1 and C_2 of the matrix $C = AB$ can be obtained using only three multiplications of real matrices (and not four as in the classical expression):

$$\begin{aligned} T_1 &= A_1 B_1, & C_1 &= T_1 - T_2, \\ T_2 &= A_2 B_2, & C_2 &= (A_1 + A_2)(B_1 + B_2) - T_1 - T_2. \end{aligned} \quad (2.10)$$

The savings are realized through the way the imaginary part C_2 is computed. Unfortunately, the above formulation may suffer from catastrophic cancellations, [18].

For large n , there is a 25 % benefit in arithmetic operations over the conventional approach. Although remarkable, this benefit does not lower the complexity which remains the same, i.e. $O(n^3)$. To push such advantage further, one may use the Strassen's approach in the three matrix multiplications above to realize $O(n^{\omega_0})$ arithmetic operations.

Parallelism is achieved at several levels:

- All the matrix operations are additions and multiplications. They can be implemented with full efficiency. In addition, the multiplication can be realized through the Strassen algorithm as implemented in CAPS, see Sect. 2.2.2.
- The three matrix multiplications are independent, once the two additions are performed.

2.3 General Organization for Dense Matrix Factorizations

In this section, we describe the usual techniques for expressing parallelism in the factorization schemes (i.e. the algorithms that compute any of the well-known decompositions such as LU, Cholesky, or QR). More specific factorizations are included in the ensuing chapters of the book.

2.3.1 Fan-Out and Fan-In Versions

Factorization schemes can be based on one of two basic templates: the *fan-out* template (see Algorithm 2.1) and the *fan-in* version (see Algorithm 2.2). Each of these templates involves two basic procedures which we generically call `compute(j)` and `update(j, k)`. The two versions, however, differ only by a single loop interchange.

Algorithm 2.1 *Fan-out* version for factorization schemes.

```

do j = 1 : n,
  compute(j) ;
  do k = j + 1 : n,
    update(j, k) ;
  end
end

```

Algorithm 2.2 *Fan-in* version for factorization schemes.

```

do k = 1 : n,
  do j = 1 : k - 1,
    update(j, k) ;
  end
  compute(k) ;
end

```

The above implementations are also respectively named as the *right-looking* and the *left-looking* versions. The exact definitions of the basic procedures, when applied to a given matrix A , are displayed in Table 2.1 together with their arithmetic complexities on a uniprocessor. They are based on a column oriented organization. For the analysis of loop dependencies, it is important to consider that column j is unchanged by task `update(j, k)` whereas column k is overwritten by the same task; column j is overwritten by task `compute(j)`.

The two versions are based on vector operations (i.e. BLAS1). It can be seen, however, that for a given j , the inner loop of the *fan-out* algorithm is a rank-one update (i.e. BLAS2), with a special feature for the Cholesky factorization, where only the lower triangular part of A is updated.

Table 2.1 Elementary factorization procedures; MATLAB index notation used for submatrices

Factorization	Procedures	Complexity
Cholesky on $A \in \mathbb{R}^{n \times n}$	$C : A(j : n, j) = A(j : n, j) / \sqrt{A(j, j)}$ $U : A(k : n, k) = A(k : n, j) - A(k : n, j)A(k, j)$	$\frac{1}{3}n^3 + O(n^2)$
LU on $A \in \mathbb{R}^{n \times n}$ (no pivoting)	$C : A(j : n, j) = A(j : n, j) / A(j, j)$ $U : A(j : n, k) = A(j : n, j) - A(j : n, j)A(k, j)$	$\frac{2}{3}n^3 + O(n^2)$
QR on $A \in \mathbb{R}^{m \times n}$ (Householder)	$C : u = \text{house}(A(j : n, j))$ and $\beta = 2/\ u\ ^2$ $U : A(j : n, k) = A(j : n, j) - \beta u(u^\top A(j : n, j))$	$2n^2(m - \frac{1}{3}n) + O(mn)$
MGS on $A \in \mathbb{R}^{m \times n}$ (Modified Gram-Schmidt)	$C : A(1 : n, j) = A(1 : n, j) / \ A(1 : n, j)\ $ $U : A(1 : n, k) = A(1 : n, k) - A(1 : n, k)(A(1 : n, k)^\top A(1 : n, j))$	$2mn^2 + O(mn)$

Notations C: compute(j); U: update(j, k)

$v = \text{house}(u)$: computes the Householder vector (see [19])

2.3.2 Parallelism in the Fan-Out Version

In the *fan-out* version, the inner loop (loop k) of Algorithm 2.1 involves independent iterations whereas in the *fan-in* version, the inner loop (loop j) of Algorithm 2.2 must be sequential because of a recursion on vector k .

The inner loop of Algorithm 2.1 can be expressed as a **doall** loop. The resulting algorithm is referred to as Algorithm 2.3.

Algorithm 2.3 do/doall *fan-out* version for factorization schemes.

```

do  $j = 1 : n$ ,
  compute( $j$ ) ;
  doall  $k = j + 1 : n$ ,
    update( $j, k$ ) ;
  end
end

```

At the outer iteration j , there are $n - j$ independent tasks with identical cost. When the outer loop is regarded as a sequential one, idle processors will result at the end of most of the outer iterations. Let p be the number of processors used, and for the sake of simplicity, let $n = pq + 1$ and assume that the time spent by one processor in executing task compute(j) or task update(j, k) is the same which is taken as the time unit. Note that this last assumption is valid only for the Gram-Schmidt orthogonalization, since for the other algorithms, the cost of task compute(j) and task update(j, k) are proportional to $n - j$ or even smaller for the Cholesky factorization. A simple computation shows that the sequential process consumes $\mathbf{T}_1 = n(n + 1)/2$ steps, whereas the parallel process on p processors consumes $\mathbf{T}_p = 1 + p \sum_{i=2}^{q+1} i = \frac{pq(q+3)}{2} + 1 = \frac{(n-1)(n-1+3p)}{2p} + 1$ steps. For

Table 2.2 Benefit of pipelining the outer loop in MGS (QR factorization)

steps	parallel runs	steps	parallel runs
1	C(1)	1	C(1)
2	U(1,2) U(1,3) U(1,4) U(1,5)	2	U(1,2) U(1,3) U(1,4) U(1,5)
3	U(1,6) U(1,7) U(1,8) U(1,9)	3	C(2) U(1,6) U(1,7) U(1,8)
4	C(2)	4	U(1,9) U(2,3) U(2,4) U(2,5)
5	U(2,3) U(2,4) U(2,5) U(2,6)	5	C(3) U(2,6) U(2,7) U(2,8)
6	U(2,7) U(2,8) U(2,9)	6	U(2,9) U(3,4) U(3,5) U(3,6)
7	C(3)	7	C(4) U(3,7) U(3,8) U(3,9)
8	U(3,4) U(3,5) U(3,6) U(3,7)	8	U(4,5) U(4,6) U(4,7) U(4,8)
9	U(3,8) U(3,9)	9	C(5) U(4,9)
10	C(4)	10	U(5,6) U(5,7) U(5,8) U(5,9)
11	U(4,5) U(4,6) U(4,7) U(4,8)	11	C(6)
12	U(4,9)	12	U(6,7) U(6,8) U(6,9)
13	C(5)	13	C(7)
14	U(5,6) U(5,7) U(5,8) U(5,9)	14	U(7,8) U(7,9)
15	C(6)	15	C(8)
16	U(6,7) U(6,8) U(6,9)	16	U(8,9)
17	C(7)	17	C(9)
18	U(7,8) U(7,9)		
19	C(8)		
20	U(8,9)		
21	C(9)		

Notations : C(j) = compute(j)
 U(j,k) = update(j,k)

(a) Sequential outer loop.

(b) **doacross** outer loop.

instance, for $n = 9$ and $p = 4$, the parallel calculation is performed in 21 steps (see Table 2.2a) whereas the sequential algorithm requires 45 steps. In Fig. 2.1, the efficiency $E_p = \frac{n(n+1)}{(n-1)(n-1+3p)+2p}$ is displayed for $p = 4, 8, 16, 32, 64$ processors when dealing with vectors of length n , where $100 \leq n \leq 1000$.

The efficiency study above is for the Modified Gram-Schmidt (MGS) algorithm. Even though the analysis for other factorizations is more complicated, the general behavior of the corresponding efficiency curves with respect to the vector length, does not change.

The next question to be addressed is whether the iterations of the outer loop can be pipelined so that they can be implemented utilizing the **doacross**.

At step j , for $k = j + 1, \dots, n$, task $\text{update}(j, k)$ may start as soon as task $\text{compute}(j)$ is completed but $\text{compute}(j)$ may start as soon as all the tasks $\text{update}(l, j)$, for $l = 1, \dots, j - 1$ are completed. Maintaining the serial execution of tasks $\text{update}(l, j)$ for $l = 1, \dots, j - 1$ is equivalent to guaranteeing that any task $\text{update}(j, k)$ cannot start before completion of $\text{update}(j - 1, k)$. The resulting scheme is listed as Algorithm 2.4.

In our particular example, scheduling of the elementary tasks is displayed in Table 2.2b. Comparing with the non-pipelined scheme, we clearly see that the processors are fully utilized whenever the number of remaining vectors is large enough. On the other hand, the end of the process is identical for the two strategies. Therefore, pipelining the outer loop is beneficial except when p is much smaller than n .

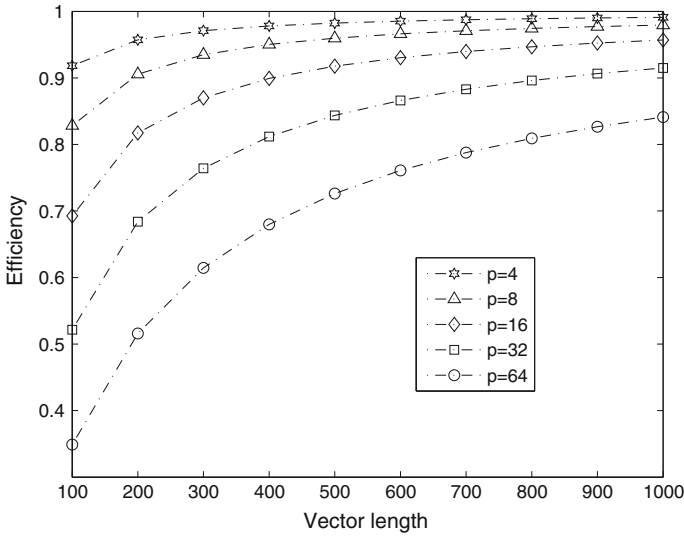


Fig. 2.1 Efficiencies of the **doall** approach with a sequential outer loop in MGS

Algorithm 2.4 **doacross/doall** version for factorization schemes (*fan-out*.)

```

doacross  $j = 1 : n$ ,
  if  $j > 1$ , then
    wait( $j$ );
  end if
  compute( $j$ );
  doall  $k = j + 1 : n$ ,
    update( $j, k$ );
    if  $k = j + 1$ , then
      post( $j + 1$ );
    end if
  end
end

```

2.3.3 Data Allocation for Distributed Memory.

The previous analysis is valid for shared or distributed memory architectures. However, for distributed memory systems we need to discuss the data allocation. As an illustration consider a ring of p processors, numbered from 0 to $p - 1$, on which r consecutive columns of A are stored in a round-robin mode. By denoting $\tilde{j} = j - 1$, column j is stored on processor s when $\tilde{j} = r(pv + t) + s$ with $0 \leq s < r$ and $0 \leq t < p$.

As soon as column j is ready, it is broadcast to the rest of the processors so they can start tasks $\text{update}(j, k)$ for the columns k which they own. This implements the **doacross/doall** strategy of the *fan-out* approach, listed as Algorithm 2.5.

To reduce the number of messages, one may transfer only the blocks of r consecutive vectors when they are all ready to be used (i.e. the corresponding $\text{compute}(j)$ tasks are completed). The drawback of this option is increasing the periods during which there are idle processors. Therefore, the block size r must be chosen so as to obtain a better trade-off between using as many processors as possible and reducing communication cost. Clearly, the optimum value is architecture dependent as it depends on the smallest efficient task granularity.

Algorithm 2.5 Message passing *fan-out* version for factorization schemes.

Input: processor # q owns the set C_q of columns of A .

```

do  $j = 1 : n$ ,
  if  $j \in C_q$ , then
    compute( $j$ ) ;
    sendtoall( $j$ ) ;
  else
    receive( $j$ ) ;
  end if
  do ( $k \in C_q$ ) & ( $k > j$ ),
    update( $j, k$ ) ;
  end
end

```

The discussion above could easily be extended to the case of a torus configuration where each processor of the previous ring is replaced by a ring of q processors. Every column of the matrix A is now distributed into slices on the corresponding ring in a round-robin mode. This, in turn, implies global communication in each ring of q processors.

2.3.4 Block Versions and Numerical Libraries

We have already seen that it is useful to block consecutive columns of A . Actually, there is benefit of doing so, even on a uniprocessor. In Algorithms 2.1 and 2.2, tasks $\text{compute}(j)$ and $\text{update}(j, k)$ can be redefined to operate on a block of vectors rather than on a single vector. In that case, indices j and k would refer to blocks of r consecutive columns. In Table 2.1 the scalar multiplications correspond now to matrix multiplications involving BLAS3 procedures. It can be shown that for all the above mentioned factorizations, task $\text{compute}(j)$ becomes the task performing the original factorization scheme on the corresponding block; cf. [19]. Task $\text{update}(j, k)$ remains formally the same but involving blocks of vectors (rank- r update) instead of individual vectors (rank-1 update).

The resulting block algorithms are mathematically equivalent to their vector counterparts but they may have different numerical behavior, especially for the Gram-Schmidt algorithm. This will be discussed in detail in Chap. 7.

Well designed block algorithms for matrix multiplication and rank- k updates for hierarchical machines with multiple levels of memory and parallelism are of critical importance for the design of solvers for the problems considered in this chapter that demonstrate high performance and scalability. The library LAPACK [20], that solves the classic matrix-problems, is a case in point by being based on BLAS3 as well as its parallel version ScaLAPACK [21]:

- LAPACK: This is the main reference for a software library for numerical linear algebra. It provides routines for solving systems of linear equations and linear least squares, eigenvalue problems, and singular value decomposition. The involved matrices can be stored as dense matrices or band matrices. The procedures are based on BLAS3 and are proved to be backward stable. LAPACK was originally written in FORTRAN 77, but moved to Fortran 90 in version 3.2 (2008).
- ScaLAPACK: This library can be seen as the parallel version of the LAPACK library for distributed memory architectures. It is based on the Message Passing Interface standard MPI [22]. Matrices and vectors are stored on a process grid into a two-dimensional block-cyclic distribution. The library is often chosen as the reference to which compare any new developed procedure.

In fact, many users became fully aware of these gains even when using high-level problem solving environments like MATLAB (cf. [23]). As early works on the subject had shown (we consider it rewarding for the reader to consider the pioneering analyses in [11, 24]), the task of designing primitives is far from simple, if one desires to provide a design that closely resembles the target computer model. The task becomes more difficult as the complexity of the computer architectures increases. It becomes even harder when the target is to build methods that can deliver high performance for a spectrum of computer architectures.

2.4 Sparse Matrix Computations

Most large scale matrix computations in computational science and engineering involve sparse matrices, that is matrices with relatively few nonzero elements, e.g. $n_{nz} = O(n)$, for square matrices of order n . See [25] for instances of matrices from a large variety of applications.

For example, in numerical simulations governed by partial differential equations approximated using finite difference or finite elements, the number of nonzero entries per row is related to the topology of the underlying finite element or finite difference grid. In two-dimensional problems discretized by a 5-point finite difference discretization scheme, the number of nonzeros is about $n_{nz} = 5n$ and the density of the resulting sparse matrix (i.e. the ratio between nonzeros entries and all entries) is $d \approx \frac{5}{n}$, where n is the matrix order.

Methods designed for dense matrix computations are rarely suitable for sparse matrices since they quickly destroy the sparsity of the original matrix leading to the need of storing a much larger number of nonzeros. However, with the availability of large memory capacities in new architectures, factorization methods (LU and QR) exist that control fill-in and manage the needed extra storage. We do not present such algorithms in this book but refer the reader to existing literature, e.g. see [26–28]. Another option is to use *matrix-free* methods in which the sparse matrix is not generated explicitly but used as an operator through the matrix-vector multiplication kernel.

To make feasible large scale computations that involve sparse matrices, they are encoded in some suitable sparse matrix storage format in which only nonzero elements of the matrix are stored together with sufficient information regarding their row and column location to access them in the course of operations.

2.4.1 Sparse Matrix Storage and Matrix-Vector Multiplication Schemes

Let $A = (\alpha_{ij}) \in \mathbb{R}^{n \times n}$ be a sparse matrix, and n_{nz} the number of nonzero entries in A .

Definition 2.1 (*Graph of a sparse matrix*) The *graph* of the matrix is given by the pair of nodes and edges $(\langle 1 : n \rangle, G)$ where G is characterized by

$$((i, j) \in G) \text{ iff } \alpha_{ij} \neq 0.$$

The adjacency matrix C of the matrix A is $C = (\gamma_{ij}) \in \mathbb{R}^{n \times n}$ such that $\gamma_{ij} = 1$ if $(i, j) \in G$ otherwise $\gamma_{ij} = 0$.

The most common sparse storage schemes are presented below together with their associated kernels: MV for matrix-vector multiplication and MTV for the multiplication by the transpose. For a complete description and some additional storage types see [29].

Compressed Row Sparse Storage (CRS)

All the nonzero entries are successively stored, row by row, in a one-dimensional array a of length n_{nz} . Column indices are stored in the same order in a vector ja of the same length n_{nz} . Since the entries are stored row by row, it is sufficient to define a third vector ia to store the indices of the beginning of each row in a . By convention, the vector is extended by one entry: $ia_{n+1} = n_{nz} + 1$. Therefore, when scanning vector a_k for $k = 1, \dots, n_{nz}$, the corresponding row index i and column index j are obtained from the following

$$a_k = \alpha_{ij} \Leftrightarrow \begin{cases} j = ja_k, \\ ia_i \leq k < ia_{i+1}. \end{cases} \quad (2.11)$$

The corresponding MV kernel is given by Algorithm 2.6. The inner loop implements a sparse inner product through a so-called gather procedure.

Algorithm 2.6 CRS-type MV.

Input: CRS storage (a, ja, ia) of $A \in \mathbb{R}^{n \times n}$ as defined in (2.11); $v, w \in \mathbb{R}^n$.

```

1: do  $i = 1 : n$ ,
2:   do  $k = ia_i : ia_{i+1} - 1$ ,
3:      $w_i = w_i + a_k v_{ja_k}$  ; //Gather
4:   end
5: end
```

Compressed Column Sparse Storage (CCS)

This storage is the dual of CRS: it corresponds to storing A^\top via a CRS format. Therefore, the nonzero entries are successively stored, column by column, in a vector a of length n_{nz} . Row indices are stored in the same order in a vector ia of length n_{nz} . The third vector ja stores the indices of the beginning of each column in a . By convention, the vector is extended by one entry: $ja_{n+1} = n_{nz} + 1$. Thus (2.11) is replaced by,

$$a_k = \alpha_{ij} \Leftrightarrow \begin{cases} j = ia_k, \\ ja_j \leq k < ja_{j+1}. \end{cases} \quad (2.12)$$

The corresponding MV kernel is given by Algorithm 2.7. The inner loop implements a sparse _AXPY through a so-called scatter procedure.

Algorithm 2.7 CCS-type MV.

Input: CCS storage (a, ja, ia) of $A \in \mathbb{R}^{n \times n}$ as defined in (2.12); $v, w \in \mathbb{R}^n$.

Output: $w = w + Av$.

```

1: do  $j = 1 : n$ ,
2:   do  $k = ja_j : ja_{j+1} - 1$ ,
3:      $w_{ia_k} = w_{ia_k} + a_k v_j$  ; //Scatter
4:   end
5: end
```

Compressed Storage by Coordinates (COO)

In this storage, no special order of the entries is assumed. Therefore three vectors a , ia and ja of length n_{nz} are used satisfying

$$a_k = \alpha_{ij} \Leftrightarrow \begin{cases} i = ia_k, \\ j = ja_k \end{cases} \quad (2.13)$$

The corresponding MV kernel is given by Algorithm 2.8. It involves both the scatter and gather procedures.

Algorithm 2.8 COO-type MV.

Input: COO storage (a, ja, ia) of $A \in \mathbb{R}^{n \times n}$ as defined in (2.13); $v, w \in \mathbb{R}^n$.

Output: $w = w + Av$.

```

1: do  $k = 1 : n_{nz}$ ,
2:    $w_{ia_k} = w_{ia_k} + a_k v_{ja_k}$  ;
3: end

```

MTV Kernel and Other Storages

When A is stored in one of the above mentioned compressed storage formats the MTV kernel

$$w = w + A^\top v,$$

is expressed for a CRS-stored matrix by Algorithm 2.7 and for a CCS-stored one by Algorithm 2.6. For a COO-stored matrix, the algorithm is obtained by inverting the roles of the arrays ia and ja in Algorithm 2.8.

Nowadays, the scatter-gather procedures (see step 3 in Algorithm 2.6 and step 3 in Algorithm 2.7) are pipelined on the architectures allowing vector computations. However, their startup time is often large (i.e. order of magnitude of $n_{1/2}$ —as defined in Sect. 1.1.2—is in the hundreds. If in MV a were a dense matrix $n_{1/2}$ would be in the tens). The vector lengths in Algorithms 2.6 and 2.7 are determined by the number of nonzero entries per row or per column. They often are so small that the computations are run at sequential computational rates. There have been many attempts to define sparse storage formats that favor larger vector lengths (e.g. see the jagged diagonal format mentioned in [30, 31]).

An efficient storage format which combines the advantages of dense and sparse matrix computations attempts to define a square block structure of a sparse matrix in which most of the blocks are empty. The non empty blocks are stored in any of the above formats, e.g. CSR, or the regular dense storage depending on the sparsity density. Such a sparse storage format is called either *Block Compressed Sparse storage* (BCRS) where the sparse nonempty blocks are stored using the CRS format, or *Block Compressed Column storage* (BCCS) where the sparse nonempty blocks are stored using the CCS format.

Basic Implementation on Distributed Memory Architecture

Let us consider the implementation of $w = w + Av$ and $w = w + A^\top v$ on a distributed memory parallel architecture with p processors where $A \in \mathbb{R}^{n \times n}$ and $v, w \in \mathbb{R}^n$. The first stage consists of partitioning the matrix and allocating respective parts to the local processor memories. Each processor P_q with $q = 1, \dots, p$, receives a block of rows of A and the corresponding slices of the vectors v and w :

$$\begin{array}{c}
P_1 : \\
P_2 : \\
\vdots \\
P_p :
\end{array}
A = \left(\begin{array}{c|c|c|c}
A_{1,1} & A_{1,2} & \cdots & A_{1,p} \\
A_{2,1} & A_{2,2} & \cdots & A_{2,p} \\
\vdots & \vdots & & \vdots \\
A_{p,1} & A_{p,2} & \cdots & A_{p,p}
\end{array} \right)
\quad
v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_p \end{pmatrix}
\quad
w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_p \end{pmatrix}$$

With the blocks $A_{q,j}$ ($j = 1, \dots, p$) residing on processor P_q , this partition determines the necessary communications for performing the multiplications. All the blocks are sparse matrices with some being empty. To implement the kernels $w = w + Av$ and $w = w + A^\top v$, the communication graph is defined by the sets $\mathcal{R}(q)$ and $\mathcal{C}(q)$ which respectively include the list of indices of the nonempty blocks $A_{q,j}$ of the block row q and $A_{j,q}$ of the block column q ($j = 1, \dots, p$). The two implementations are given by Algorithms 2.9 and 2.10, respectively.

Algorithm 2.9 MV: $w = w + Av$

Input: q : processor number.

$\mathcal{R}(q)$: list of indices of the nonempty blocks of row q .

$\mathcal{C}(q)$: list of indices of the nonempty blocks of column q .

- 1: **do** $j \in \mathcal{C}(q)$,
 - 2: send v_q to processor P_j ;
 - 3: **end**
 - 4: compute $w_q = w_q + A_{q,q}v_q$.
 - 5: **do** $j \in \mathcal{R}(q)$,
 - 6: receive v_j from processor P_j ;
 - 7: compute $w_q = w_q + A_{q,j}v_j$;
 - 8: **end**
-

Algorithm 2.10 MTV: $w = w + A^\top v$

Input: q : processor number.

$\mathcal{R}(q)$: list of indices of the nonempty blocks of row q .

$\mathcal{C}(q)$: list of indices of the nonempty blocks of column q .

- 1: **do** $j \in \mathcal{R}(q)$,
 - 2: compute $t_j = A_{q,j}^\top v_q$;
 - 3: send t_j to processor P_j ;
 - 4: **end**
 - 5: compute $w_q = w_q + A_{q,q}^\top v_q$;
 - 6: **do** $j \in \mathcal{C}(q)$,
 - 7: receive u_j from processor P_j ;
 - 8: compute $w_q = w_q + u_j$;
 - 9: **end**
-

The efficiency of the two procedures MV and MTV are often quite different, depending on the chosen sparse storage format.

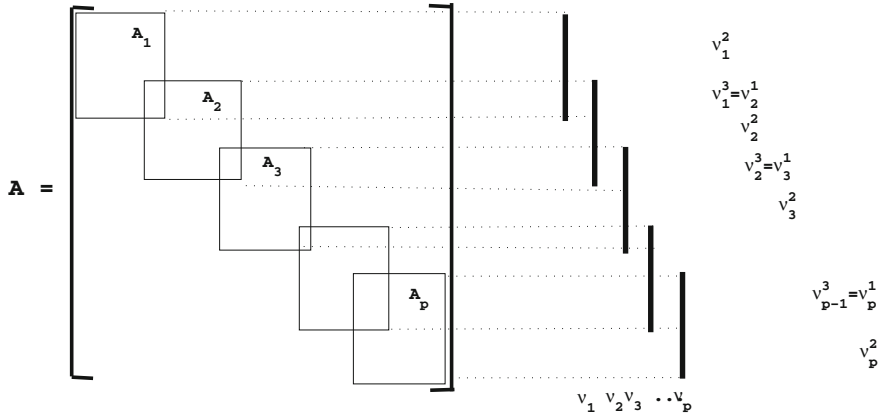


Fig. 2.2 Partition of a block-diagonal matrix with overlapping blocks; vector v is decomposed in overlapping slices

Scalable Implementation for an Overlapped Block-Diagonal Structure

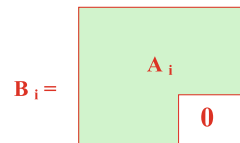
If the sparse matrix $A \in \mathbb{R}^{n \times n}$ can be reordered to result in p overlapping diagonal blocks denoted by A_q , $q = 1, \dots, p$ as shown in Fig. 2.2, then a matrix-vector multiplication primitive can be designed in high parallel scalability. Let block A_q be stored in the memory of the processor P_q and the vectors $v, w \in \mathbb{R}^n$ stored accordingly. It is therefore necessary to maintain the consistency between the two copies of the components corresponding to the overlaps.

To perform the MV computation $w = w + Av$, the matrix A may be considered as a sum of p blocks B_q ($q = 1, \dots, p$) where $B_p = A_p$ and all earlier blocks B_q are the same as A_q with the elements of lower right submatrix corresponding to the overlap are replaced by zeros (see Fig. 2.3).

Let vector v_q be the subvector of v corresponding to the q -block row indices. For $2 \leq q \leq p-1$, the vector v_q is partitioned into $v_q^\top = (v_q^{1\top}, v_q^{2\top}, v_q^{3\top})$, according to the overlap with the neighboring blocks. The first and the last subvectors are partitioned as $v_1^\top = (v_1^{2\top}, v_1^{3\top})$ and $v_p^\top = (v_p^{1\top}, v_p^{2\top})$.

Denoting \bar{B}_q and \bar{v}_q the prolongation by zeros of B_q and v_q to the full order n , the operation $w + Av = w + \sum_{q=1}^p \bar{B}_q \bar{v}_q$ can be performed via Algorithm 2.11. After completion, the vector w is correctly updated and distributed on the processors with consistent subvectors w_q (i.e. $w_{q-1}^3 = w_q^1$ for $q = 2, \dots, p$). This algorithm

Fig. 2.3 Elementary blocks for the MV kernel



Algorithm 2.11 Scalable MV multiplication $w = w + Av$.

Input: q : processor number.

In the local memory: $B_q, v_q^\top = [(v_q^1)^\top, (v_q^2)^\top, (v_q^3)^\top]$, and $w_q^\top = [(w_q^1)^\top, (w_q^2)^\top, (w_q^3)^\top]$.

Output: $w = w + Av$.

```

1:  $z_q = B_q v_q$ ;
2: if  $q < p$ , then
3:   send  $z_q^3$  to processor  $P_{q+1}$ ;
4: end if
5: if  $q > 1$ , then
6:   send  $z_q^1$  to processor  $P_{q-1}$ ;
7: end if
8:  $w_q = w_q + z_q$ ;
9: if  $q < p$ , then
10:  receive  $t$  from processor  $P_{q+1}$ ;
11:   $w_q^3 = w_q^3 + t$ ;
12: end if
13: if  $q > 1$ , then
14:  receive  $t$  from processor  $P_{q-1}$ ;
15:   $w_q^1 = w_q^1 + t$ ;
16: end if

```

does not involve global communications and it can be implemented on a linear array of processors in which every processor only exchanges information with its two immediate neighbors: each processor receives one message from each of its neighbors and it sends back one message to each.

Proposition 2.1 *Algorithm 2.11 which implements the MV kernel for a sparse matrix with an overlapped block-diagonal structure on a ring of p processors is weakly scalable as long as the number of nonzeros entries of each block and the overlap sizes are independent of the number of processors p .*

Proof Let T_{BMV} be the bound on the number of steps for the MV kernel of each individual diagonal block, and ℓ being the maximum overlap sizes, then on p processors the number of steps is given by

$$\mathbf{T}_p \leq T_{\text{BMV}} + 4(\beta + \ell\tau_c),$$

where β is the latency for a message and τ_c the time for sending a word to an immediate neighbouring node regardless of the latency. Since \mathbf{T}_p is independent of p , weak scalability is assured.

2.4.2 Matrix Reordering Schemes

Algorithms for reordering sparse matrices play a vital role in enhancing the parallel scalability of various sparse matrix algorithms and their underlying primitives, e.g., see [32, 33].

Early reordering algorithms such as minimum-degree and nested dissection have been developed for reducing fill-in in sequential direct methods for solving sparse symmetric positive definite linear systems, e.g., see [26, 34, 35]. Similarly, algorithms such as reverse Cuthill-McKee (RCM), e.g., see [36, 37], have been used for reducing the envelope (variable band or profile) of sparse matrices in order to: (i) enhance the efficiency of uniprocessor direct factorization schemes, (ii) reduce the cost of sparse matrix-vector multiplications in iterative methods such as the conjugate gradients method (CG), for example, and (iii) obtain preconditioners for the PCG scheme based on incomplete factorization [38, 39]. In this section, we here describe a reordering scheme that not only reduces the profile of sparse matrices, but also brings as many of the heaviest (larger magnitude) off-diagonal elements as possible close to the diagonal. For solving sparse linear systems, for example, one aims at realizing an overall cost, with reordering, that is much less than that without reordering. In fact, in many time-dependent computational science and engineering applications, this is possible. In such applications, the relevant nested computational loop occurs as shown in Fig. 2.4.

The outer-most loop deals with time-step t , followed by solving a nonlinear set of equations using a variant of Newton's method, with the inner-most loop dealing with solving a linear system in each Newton iteration to a relatively modest relative residual η_k . Further, it is often the case that it is sufficient to realize the benefits of reordering by keep using the permutation matrices obtained at time step t for several subsequent time steps. This results not only in amortization of the cost of reordering, but also in reducing the total cost of solving all the linear systems arising in such an application.

With such a reordering, we aim to obtain a matrix $C = PAQ$, where $A = (a_{ij})$ is the original sparse matrix, P and Q are permutation matrices, such that C can be split as $C = B + E$, with the most important requirements being that: (i) the sparse matrix E contains far fewer nonzero elements than A , and is of a much lower rank, and (ii) the central band B is a “generalized-banded” matrix with a Frobenius norm that is a substantial fraction of that of A .

Fig. 2.4 Common structure of programs in time-dependent simulations

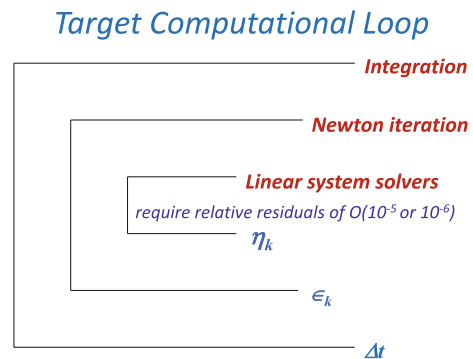
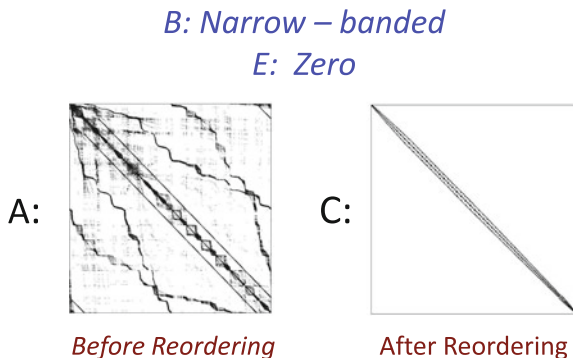


Fig. 2.5 Reordering to a narrow-banded matrix



Hence, depending on the original matrix A , the matrix B can be extracted as:

- (a) “narrow-banded” of bandwidth β much smaller than the order n of the matrix A , i.e., $\beta = 10^{-4}n$, for $n \geq 10^6$, for example (the most fortunate situation), see Fig. 2.5,
- (b) “medium-banded”, i.e., of the block-tridiagonal form $[H, G, J]$, in which the elements of the off-diagonal blocks H and J are all zero except for their small upper-right and lower-left corners, respectively, see Fig. 2.6, or
- (c) “wide-banded”, i.e., consisting of overlapped diagonal blocks, in which each diagonal block is a sparse matrix, see Fig. 2.7.

The motivation for desiring such a reordering scheme is three-fold. First, B can be used as a preconditioner of a Krylov subspace method when solving a linear system $Ax = f$ of order n . Since E is of a rank p much less than n , the preconditioned Krylov subspace scheme will converge quickly. In exact arithmetic, the Krylov subspace method will converge in exactly p iterations. In floating-point arithmetic, however, this translates into the method achieving small relative residuals in less than p iterations. Second, since we require the diagonal of B to be zero-free with the product of its entries maximized, and that the Frobenius norm of B is close to that of A , this will enhance the possibility that B is nonsingular, or close to a nonsingular matrix. Third, multiplying C by a vector can be implemented on a parallel architecture with higher efficiency by splitting the operation into two parts: multiplying the “generalized-banded” matrix B by a vector, and a low-rank sparse matrix E by a vector. The former, e.g. $v = Bu$, can be achieved with high parallel scalability on distributed-memory architectures requiring only nearest neighbor communication, e.g. see Sect. 2.4.1 for the scalable parallel implementation of an overlapped block diagonal matrix-vector multiplication scheme. The latter, e.g. $w = Eu$, however, incurs much less irregular addressing penalty compared to $y = Au$ since E contains far fewer nonzero entries than A .

Since A is nonsymmetric, in general, we could reduce its profile by using RCM (i.e. via symmetric permutations only) applied to $(|A| + |A^T|)$, [40], or by using the spectral reordering introduced in [41]; see also [42]. However, this will neither

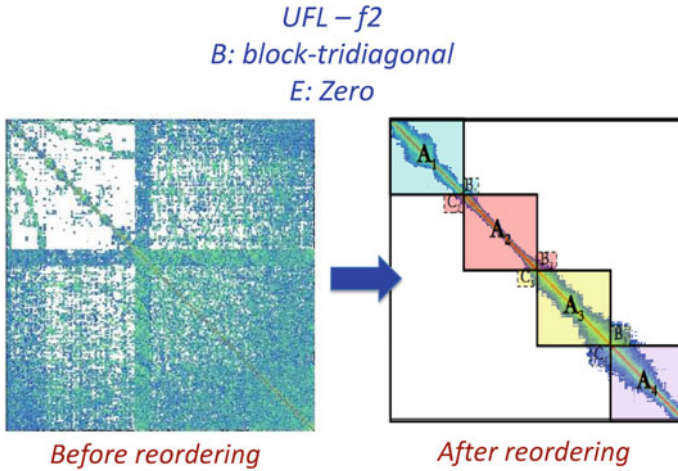


Fig. 2.6 Reordering to a medium-banded matrix

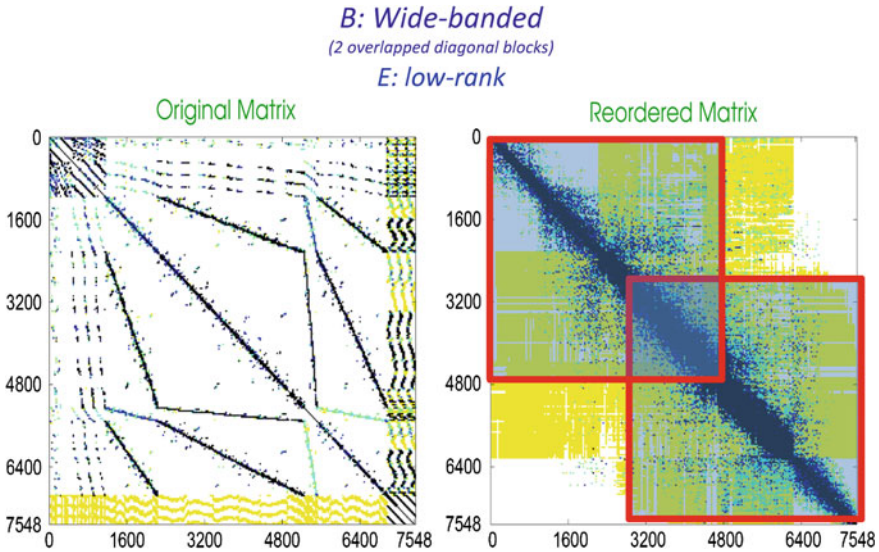


Fig. 2.7 Reordering to a wide-banded matrix

realize a zero-free diagonal, nor insure bringing the heaviest off-diagonal elements close to the diagonal. Consequently, RCM alone will not realize a central “band” B with its Frobenius norm satisfying: $\|B\|_F \geq (1 - \varepsilon) \|A\|_F$. In order to solve this *weighted bandwidth reduction* problem, we use a weighted spectral reordering technique which is a generalization of spectral reordering. To alleviate the shortcomings of using only symmetric permutations, and assuming that the matrix A is not structurally singular, this weighted spectral reordering will need to be coupled with

a nonsymmetric ordering technique such as the maximum traversal algorithm [43] to guarantee a zero-free diagonal, and to maximize the magnitude of the product of the diagonal elements, via the MPD algorithm (Maximum Product on Diagonal algorithm) [44, 45]. Such a procedure is implemented in the Harwell Subroutine Library [46] as (HSL-MC64).

Thus, the resulting algorithm, which we refer to as WSO (Weighted Spectral Ordering), consists of three stages:

Stage 1: Nonsymmetric Permutations

Here, we obtain a permutation matrix Q that maximizes the product of the absolute values of the diagonal entries of QA , [44, 45]. This is achieved by a maximum traversal search followed by a scaling procedure resulting in diagonal entries of absolute values equal to 1, and all off-diagonal elements with magnitudes less than or equal to 1. After applying this stage, a linear system $Ax = f$, becomes of the form,

$$(QD_2AD_1)(D_1^{-1}x) = (QD_2f) \quad (2.14)$$

in which each D_j , $j = 1, 2$ is a diagonal scaling matrix.

Stage 2: Checking for Irreducibility

In this stage, we need to detect whether the sparse matrix under consideration is irreducible, i.e., whether the corresponding graph has one strongly connected component. This is achieved via Tarjan's strongly connected component algorithm [47], see also related schemes in [48], or [49]. If the matrix is reducible, we apply the weighted spectral reordering simultaneously on each strongly connected component (i.e. on each sparse diagonal block of the resulting upper block triangular matrix). For the rest of this section, we assume that the sparse matrix A under consideration is irreducible, i.e., the corresponding graph has only one strongly connected component.

Stage 3: The Weighted Spectral Reordering Scheme [50]

As in other traditional reordering algorithms, we wish to minimize the half-bandwidth of a matrix A which is given by,

$$BW(A) = \max_{i,j:\alpha_{ij} \neq 0} |i - j|, \quad (2.15)$$

i.e., to minimize the maximum distance of a nonzero entry from the main diagonal. Let us assume for the time being that A is a symmetric matrix, and that we aim at extracting a central band $B = (\beta_{ij})$ of minimum bandwidth such that, for a given tolerance ε ,

$$\frac{\sum_{i,j} |\alpha_{ij} - \beta_{ij}|}{\sum_{i,j} |\alpha_{ij}|} \leq \varepsilon, \quad (2.16)$$

and

$$\begin{aligned}\beta_{ij} &= \alpha_{ij} \text{ if } |i - j| \leq k, \\ \beta_{ij} &= 0\end{aligned}\tag{2.17}$$

The idea behind this formulation is that if a significant part of the matrix is packed into a central band B , then the rest of the nonzero entries can be dropped to obtain an effective preconditioner. In order to find a heuristic solution to the weighted bandwidth reduction problem, we use a generalization of spectral reordering. Spectral reordering is a linear algebraic technique that is commonly used to obtain approximate solutions to various intractable graph optimization problems [51]. It has also been successfully applied to the bandwidth and envelope reduction problems for sparse matrices [41]. The core idea of spectral reordering is to compute a vector $x = (\xi_i)$ that minimizes

$$\sigma_A(x) = \sum_{i,j:\alpha_{ij}\neq 0} (\xi_i - \xi_j)^2,\tag{2.18}$$

subject to $\|x\|_2 = 1$ and $x^\top e = 0$. As mentioned above we assume that the matrix A is real and symmetric. The vector x that minimizes $\sigma_A(x)$ under these constraints provides a mapping of the rows (and columns) of matrix A to a one-dimensional Euclidean space, such that pairs of rows that correspond to nonzeros are located as close as possible to each other. Consequently, the ordering of the entries of the vector x provides an ordering of the matrix that significantly reduces the bandwidth.

Fiedler [52] first showed that the optimal solution to this problem is given by the eigenvector corresponding to the second smallest eigenvalue of the Laplacian matrix $L = (\lambda_{ij})$ of A ,

$$\begin{aligned}\lambda_{ij} &= -1 && \text{if } i \neq j \wedge \alpha_{ij} \neq 0, \\ \lambda_{ii} &= |\{j : \alpha_{ij} \neq 0\}|.\end{aligned}\tag{2.19}$$

Note that the matrix L is positive semidefinite, and the smallest eigenvalue of this matrix is equal to zero. The eigenvector x that minimizes $\sigma_A(x) = x^\top Lx$, such that $\|x\|_2 = 1$ and $x^\top e = 0$, is the eigenvector corresponding to the second smallest eigenvalue of the Laplacian, i.e. the symmetric eigenvalue problem

$$Lx = \lambda x,\tag{2.20}$$

and is known as the Fiedler vector. The Fiedler vector of a sparse matrix can be computed efficiently using any of the eigensolvers discussed in Chap. 11, see also [53].

While spectral reordering is shown to be effective in bandwidth reduction, the classical approach described above ignores the magnitude of nonzeros in the matrix. Therefore, it is not directly applicable to the weighted bandwidth reduction problem. However, Fiedler's result can be directly generalized to the weighted case [54]. More precisely, the eigenvector x that corresponds to the second smallest eigenvalue of the

weighted Laplacian L minimizes

$$\bar{\sigma}_A(x) = x^\top Lx = \sum_{i,j} |\alpha_{ij}| (\xi_i - \xi_j)^2, \quad (2.21)$$

where L is defined as

$$\begin{aligned} \lambda_{ij} &= -|\alpha_{ij}| \quad \text{if } i \neq j, \\ \lambda_{ii} &= \sum_j |\alpha_{ij}|. \end{aligned} \quad (2.22)$$

We now show how weighted spectral reordering can be used to obtain a continuous approximation to the weighted bandwidth reduction problem. For this purpose, we first define the relative *bandweight* of a specified band of the matrix as follows:

$$w_k(A) = \frac{\sum_{i,j: |i-j| < k} |\alpha_{ij}|}{\sum_{i,j} |\alpha_{ij}|}. \quad (2.23)$$

In other words, the bandweight of a matrix A , with respect to an integer k , is equal to the fraction of the total magnitude of entries that are encapsulated in a band of half-width k .

For a given α , $0 \leq \alpha \leq 1$, we define α -bandwidth as the smallest half-bandwidth that encapsulates a fraction α of the total matrix weight, i.e.,

$$BW_\alpha(A) = \min_{k: w_k(A) \geq \alpha} k. \quad (2.24)$$

Observe that α -bandwidth is a generalization of half-bandwidth, i.e., when $\alpha = 1$, the α -bandwidth is equal to the half-bandwidth of the matrix. Now, for a given vector $x = (\xi_1, \xi_2, \dots, \xi_n)^\top \in \mathbb{R}^n$, define an injective permutation function $\pi: \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$, such that, for $1 \leq i, j \leq n$, $\xi_{\pi_i} \leq \xi_{\pi_j}$ iff $i \leq j$. Here, n denotes the number of rows (columns) of the matrix A . Moreover, for a fixed k , define the function $\delta_k(i, j): \{1, 2, \dots, n\} \times \{1, 2, \dots, n\} \rightarrow \{0, 1\}$, which quantizes the difference between π_i and π_j with respect to k , i.e.,

$$\delta_k(i, j) = \begin{cases} 0 & \text{if } |\pi_i - \pi_j| \leq k, \\ 1 & \text{else} \end{cases} \quad (2.25)$$

Let \bar{A} be the matrix obtained by reordering the rows and columns of A according to π , i.e.,

$$\bar{A}(\pi_i, \pi_j) = \alpha_{ij} \text{ for } 1 \leq i, j \leq n. \quad (2.26)$$

Then $\delta_k(i, j) = 0$ indicates that α_{ij} is inside a band of half-width k in the matrix \bar{A} while $\delta_k(i, j) = 1$ indicates that it is outside the band. Defining

$$\hat{\sigma}_k(A) = \sum_{i,j} |\alpha_{ij}| \delta_k(i, j), \quad (2.27)$$

then,

$$\hat{\sigma}_k(A) = (1 - w_k(\bar{A})) \sum_{i,j} |\alpha_{ij}|. \quad (2.28)$$

Therefore, for a fixed α , the α -bandwidth of the matrix \bar{A} is equal to the smallest k that satisfies $\hat{\sigma}_A(k) / \sum_{i,j} |\alpha_{ij}| \leq 1 - \alpha$.

Note that the problem of minimizing $\bar{\sigma}_x(A)$ is a continuous relaxation of the problem of minimizing $\hat{\sigma}_k(A)$ for a given k . Therefore, the Fiedler vector of the weighted Laplacian L provides a good basis for reordering A to minimize $\hat{\sigma}_k(A)$. Consequently, for a fixed ε , this vector provides a heuristic solution to the problem of finding a reordered matrix $\bar{A} = (\bar{\alpha}_{ij})$ with minimum $(1 - \varepsilon)$ -bandwidth. Once the matrix is obtained, we extract the central band B as follows:

$$B = \{\beta_{ij} = \bar{\alpha}_{ij} \text{ if } |i - j| \leq BW_{1-\varepsilon}(\bar{A}), \text{ otherwise } \beta_{ij} = 0\}. \quad (2.29)$$

Clearly, B satisfies (2.16) and is of minimal bandwidth.

Note that spectral reordering is defined specifically for symmetric matrices, and the resulting permutation is symmetric as well. Since our main focus here concerns general nonsymmetric matrices, we apply spectral reordering to nonsymmetric matrices by computing the Laplacian matrix of $|A| + |A^\top|$ instead of $|A|$. We note also that this formulation results in a symmetric permutation for a nonsymmetric matrix, which may be considered overconstrained.

Once, the Fiedler vector yields the permutation P , we obtain the matrix C as,

$$C = (PQD_2AD_1P^\top), \quad (2.30)$$

and the linear system $Ax = f$ becomes of the final form,

$$(PQD_2AD_1P^\top)(PD_1^{-1}x) = (PQD_2f). \quad (2.31)$$

References

1. Lawson, C., Hanson, R., Kincaid, D., Krogh, F.: Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.* **5**(3), 308–323 (1979)
2. Dongarra, J., Croz, J.D., Hammarling, S., Hanson, R.: An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.* **14**(1), 1–17 (1988)
3. Dongarra, J., Du Croz, J., Hammarling, S., Duff, I.: A set of level-3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* **16**(1), 1–17 (1990)
4. Intel company: Intel Math Kernel Library. <http://software.intel.com/en-us/intel-mkl>

5. Texas advanced computer center, University of Texas: GotoBLAS2. <https://www.tacc.utexas.edu/tacc-software/gotoblas2>
6. Netlib Repository at UTK and ORNL: Automatically Tuned Linear Algebra Software (ATLAS). <http://www.netlib.org/atlas/>
7. Whaley, R., Dongarra, J.: Automatically tuned linear algebra software. In: Proceedings of 1998 ACM/IEEE Conference on Supercomputing, Supercomputing'98, pp. 1–27. IEEE Computer Society, Washington (1998). <http://dl.acm.org/citation.cfm?id=509058.509096>
8. Yotov, K., Li, X., Ren, G., Garzarán, M., Padua, D., Pingali, K., Stodghill, P.: Is search really necessary to generate high-performance BLAS? Proc. IEEE **93**(2), 358–386 (2005). doi:[10.1109/JPROC.2004.840444](https://doi.org/10.1109/JPROC.2004.840444)
9. Goto, K., van de Geijn, R.: Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. **34**(3), 12:1–12:25 (2008). doi:[10.1145/1356052.1356053](https://doi.org/10.1145/1356052.1356053). <http://doi.acm.org/10.1145/1356052.1356053>
10. Gallivan, K.A., Plemmons, R.J., Sameh, A.H.: Parallel algorithms for dense linear algebra computations. SIAM Rev. **32**(1), 54–135 (1990). doi:[http://dx.doi.org/10.1137/1032002](https://doi.org/10.1137/1032002)
11. Gallivan, K., Jalby, W., Meier, U.: The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory. SIAM J. Sci. Stat. Comput. **8**(6), 1079–1084 (1987)
12. Strassen, V.: Gaussian elimination is not optimal. Numerische Mathematik **13**, 354–356 (1969)
13. Winograd, S.: On multiplication of 2×2 matrices. Linear Algebra Appl. **4**(4), 381–388 (1971)
14. Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., Schwartz, O.: Communication-optimal parallel algorithm for Strassen matrix multiplication. Technical report UCB/EECS-2012-32, EECS Department, University of California, Berkeley (2012). <http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-32.html>
15. Higham, N.J.: Exploiting fast matrix multiplication within the level 3 BLAS. ACM Trans. Math. Softw. **16**(4), 352–368 (1990)
16. Ballard, G., Demmel, J., Holtz, O., Schwartz, O.: Graph expansion and communication costs of fast matrix multiplication. J. ACM **59**(6), 32:1–32:23 (2012). doi:[10.1145/2395116.2395121](https://doi.org/10.1145/2395116.2395121). <http://doi.acm.org/10.1145/2395116.2395121>
17. Lipshitz, B., Ballard, G., Demmel, J., Schwartz, O.: Communication-avoiding parallel Strassen: implementation and performance. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC'12, pp. 101:1–101:11. IEEE Computer Society Press, Los Alamitos (2012). <http://dl.acm.org/citation.cfm?id=2388996.2389133>
18. Higham, N.J.: Stability of a method for multiplying complex matrices with three real matrix multiplications. SIAM J. Matrix Anal. Appl. **13**(3), 681–687 (1992)
19. Golub, G., Van Loan, C.: Matrix Computations, 4th edn. Johns Hopkins (2013)
20. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia (1999)
21. Blackford, L., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.: ScaLAPACK User's Guide. SIAM, Philadelphia (1997). <http://www.netlib.org/scalapack>
22. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message Passing Interface. MIT Press, Cambridge (1994)
23. Moler, C.: MATLAB incorporates LAPACK. Mathworks Newsletter (2000). <http://www.mathworks.com/company/newsletters/articles/matlab-incorporates-lapack.html>
24. Gallivan, K., Jalby, W., Meier, U., Sameh, A.: The impact of hierarchical memory systems on linear algebra algorithm design. Int. J. Supercomput. Appl. **2**(1) (1988)
25. Davis, T., Hu, Y.: The University of Florida Sparse Matrix Collection. ACM Trans. Math. Softw. **38**(1), 1:1–1:25 (2011). <http://doi.acm.org/10.1145/2049662.2049663>
26. Duff, I., Erisman, A., Reid, J.: Direct Methods for Sparse Matrices. Oxford University Press Inc., New York (1989)
27. Davis, T.: Direct Methods for Sparse Linear Systems. SIAM, Philadelphia (2006)

28. Zlatev, Z.: *Computational Methods for General Sparse Matrices*, vol. 65. Kluwer Academic Publishers, Dordrecht (1991)
29. Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., van der Vorst, H.: *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia (2000)
30. Melhem, R.: Toward efficient implementation of preconditioned conjugate gradient methods on vector supercomputers. *Int. J. Supercomput. Appl.* **1**(1), 70–98 (1987)
31. Philippe, B., Saad, Y.: Solving large sparse eigenvalue problems on supercomputers. Technical report RIACS TR 88.38, NASA Ames Research Center (1988)
32. Schenk, O.: *Combinatorial Scientific Computing*. CRC Press, Switzerland (2012)
33. Kepner, J., Gilbert, J.: *Graph Algorithms in the Language of Linear Algebra*. SIAM, Philadelphia (2011)
34. George, J., Liu, J.: *Computer Solutions of Large Sparse Positive Definite Systems*. Prentice Hall (1981)
35. Pissanetzky, S.: *Sparse Matrix Technology*. Academic Press, New York (1984)
36. Cuthill, E., McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: *Proceedings of 24th National Conference Association Computer Machinery*, pp. 157–172. ACM Publications, New York (1969)
37. Liu, W., Sherman, A.: Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. *SIAM J. Numer. Anal.* **13**, 198–213 (1976)
38. D’Azevedo, E.F., Forsyth, P.A., Tang, W.P.: Ordering methods for preconditioned conjugate gradient methods applied to unstructured grid problems. *SIAM J. Matrix Anal.* **13**(3), 944–961 (1992)
39. Duff, I., Meurant, G.: The effect of ordering on preconditioned conjugate gradients. *BIT* **29**, 635–657 (1989)
40. Reid, J., Scott, J.: Reducing the total bandwidth of a sparse unsymmetric matrix. *SIAM J. Matrix Anal. Appl.* **28**(3), 805–821 (2005)
41. Barnard, S., Pothen, A., Simon, H.: A spectral algorithm for envelope reduction of sparse matrices. *Numer. Linear Algebra Appl.* **2**, 317–334 (1995)
42. Spielman, D., Teng, S.: Spectral partitioning works: planar graphs and finite element meshes. *Numer. Linear Algebra Appl.* **421**, 284–305 (2007)
43. Duff, I.: On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Softw.* **7**, 315–330 (1981)
44. Duff, I., Koster, J.: On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl.* **22**, 973–966 (2001)
45. Duff, I., Koster, J.: The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.* **20**, 889–901 (1999)
46. The HSL mathematical software library. See <http://www.hsl.rl.ac.uk/index.html>
47. Tarjan, R.: Depth-first search and linear graph algorithms. *SIAM J. Comput.* **1**(2), 146–160 (1972)
48. Cheriyan, J., Mehlhorn, K.: Algorithms for dense graphs and networks on the random access computer. *Algorithmica* **15**, 521–549 (1996)
49. Dijkstra, E.: *A Discipline of Programming*, Chapter 25. Prentice Hall, Englewood Cliffs (1976)
50. Manguoglu, M., Mehmet, K., Sameh, A., Grama, A.: Weighted matrix ordering and parallel banded preconditioners for iterative linear system solvers. *SIAM J. Sci. Comput.* **32**(3), 1201–1206 (2010)
51. Hendrickson, B., Leland, R.: An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput.* **16**(2), 452–469 (1995). <http://citeseer.nj.nec.com/hendrickson95improved.html>
52. Fiedler, M.: Algebraic connectivity of graphs. *Czechoslovak Math. J.* **23**, 298–305 (1973)
53. Krout, N.: A conjugate gradient method for the spectral partitioning of graphs. *Parallel Comput.* **22**, 1493–1502 (1997)
54. Chan, P., Schlag, M., Zien, J.: Spectral k-way ratio-cut partitioning and clustering. *IEEE Trans. CAD-Integr. Circuits Syst.* **13**, 1088–1096 (1994)

Parallelism in Matrix Computations

Gallopoulos, E.; Philippe, B.; Sameh, A.H.

2016, XXX, 473 p. 58 illus., Hardcover

ISBN: 978-94-017-7187-0