which shows that Formula (8.32) for $k+1$ is derived. In summary, the cyclic reduction for the discretized two-dimensional Poisson equation consists of the following two steps:

1. Elimination phase: For $k = 1, \ldots, \lfloor \log N \rfloor$, the matrices $B^{(k)}$ and the vectors $D_j^{(k)}$ are computed for $j = 2^k, \ldots, N$ with step-size 2^k according to Eq. (8.32).
2. Substitution phase: For $k = \lfloor \log N \rfloor, \ldots, 0$, the linear equation system

$$B^{(k)} Z_j = D_j^{(k)} + Z_{j-2^k} + Z_{j+2^k}$$

for $j = 2^k, \ldots, N$ with step-size 2^{k+1} is solved.

In the first phase, $\lfloor \log N \rfloor$ matrices and $O(N)$ subvectors are computed. The computation of each matrix includes a matrix multiplication with time $O(N^3)$. The computation of a subvector includes a matrix-vector multiplication with complexity $O(N^2)$. Thus, the first phase has a computational complexity of $O(N^3 \log N)$. In the second phase, $O(N)$ linear equation systems are solved. This requires time $O(N^3)$ when the special structure of the matrices $B^{(k)}$ is not exploited. In [67], it is shown how to reduce the time by exploiting this structure. A parallel implementation of the discretized Poisson equation can be done in an analogous way as shown in the previous section.

8.3 Iterative Methods for Linear Systems

In this section, we introduce classical iteration methods for solving linear equation systems, including the Jacobi iteration, the Gauss-Seidel iteration and the SOR method (successive over-relaxation), and discuss their parallel implementation. Direct methods as presented in the previous sections involve a factorization of the coefficient matrix. This can be impractical for large and sparse matrices, since fill-ins with nonzero elements increase the computational work. For banded matrices, special methods can be adapted and used as discussed in Sect.8.2. Another possibility is to use iterative methods as presented in this section.

Iterative methods for solving linear equation systems $Ax = b$ with coefficient matrix $A \in \mathbb{R}^{n \times n}$ and right-hand side $b \in \mathbb{R}^n$ generate a sequence of approximation vectors $\{x^{(k)}\}_{k=1,2,\ldots}$ that converges to the solution $x^* \in \mathbb{R}^n$. The computation of an approximation vector essentially involves a matrix-vector multiplication with the iteration matrix of the specific problem. The matrix A of the linear equation system is used to build this iteration matrix. For the evaluation of an iteration method, it is essential how quickly the iteration sequence converges. Basic iteration methods are the Jacobi- and the Gauss-Seidel methods, which are also called **relaxation methods** historically, since the computation of a new approximation depends on a combination of the previously computed approximation vectors. Depending on the specific problem to be solved, relaxation methods can be faster than direct solution

methods. But still these methods are not fast enough for practical use. A better convergence behavior can be observed for methods like the SOR method, which has a similar computational structure. The practical importance of relaxation methods is their use as preconditioner in combination with solution methods like the conjugate gradient method or the multigrid method. Iterative methods are a good first example to study parallelism typical also for more complex iteration methods. In the following, we describe the relaxation methods according to [23], see also [77, 186]. Parallel implementations are considered in [66, 67, 78, 172].

8.3.1 Standard Iteration Methods

Standard iteration methods for the solution of a linear equation system $Ax = b$ are based on a splitting of the coefficient matrix $A \in \mathbb{R}^{n\times n}$ into

$$A = M - N \quad \text{with } M, N \in \mathbb{R}^{n\times n},$$

where M is a nonsingular matrix for which the inverse M^{-1} can be computed easily, e.g., a diagonal matrix. For the unknown solution x^* of the equation $Ax = b$ we get

$$Mx^* = Nx^* + b.$$

This equation induces an iteration of the form $Mx^{(k+1)} = Nx^{(k)} + b$, which is usually written as

$$x^{(k+1)} = Cx^{(k)} + d \tag{8.36}$$

with iteration matrix $C := M^{-1}N$ and vector $d := M^{-1}b$. The iteration method is called *convergent* if the sequence $\{x^{(k)}\}_{k=1,2,\ldots}$ converges toward x^* independently from the choice of the start vector $x^{(0)} \in \mathbb{R}^n$, i.e., $\lim_{k\to\infty} x^{(k)} = x^*$ or $\lim_{k\to\infty} \|x^{(k)} - x^*\| = 0$. When a sequence converges then the vector x^* is uniquely defined by $x^* = Cx^* + d$. Subtracting this equation from equation (8.36) and using induction leads to the equality $x^{(k)} - x^* = C^k(x^{(0)} - x^*)$, where C^k denotes the matrix resulting from k multiplications of C. Thus, the convergence of (8.36) is equivalent to

$$\lim_{k\to\infty} C^k = 0\ .$$

A result from linear algebra shows the relation between the convergence criteria and the spectral radius $\rho(C)$ of the iteration matrix C. (The spectral radius of a matrix is the eigenvalue with the largest absolute value, i.e., $\rho(C) = \max_{\lambda\in EW} |\lambda|$ with $EW = \{\lambda \mid Cv = \lambda v, v \neq 0\}$.) The following properties are equivalent, see [186]:

(1) Iteration (8.36) converges for every $x^{(0)} \in \mathbb{R}^n$,
(2) $\lim_{k\to\infty} C^k = 0$,

(3) $\rho(C) < 1$.

Well-known iteration methods are the Jacobi, the Gauss-Seidel, and the SOR methods.

8.3.1.1 Jacobi iteration

The Jacobi iteration is based on the splitting $A = D - L - R$ of the matrix A with $D, L, R \in \mathbb{R}^{n\times n}$. The matrix D holds the diagonal elements of A, $-L$ holds the elements of the lower triangular of A without the diagonal elements and $-R$ holds the elements of the upper triangular of A without the diagonal elements. All other elements of D, L, R are zero. The splitting is used for an iteration of the form

$$Dx^{(k+1)} = (L + R)x^{(k)} + b$$

which leads to the iteration matrix $C_{Ja} := D^{-1}(L + R)$ or

$$C_{Ja} = (c_{ij})_{i,j=1,\ldots,n} \text{ with } c_{ij} = \begin{cases} -a_{ij}/a_{ii} & \text{for } j \neq i \ , \\ 0 & \text{otherwise} \ . \end{cases}$$

The matrix form is used for the convergence proof, not shown here. For the practical computation, the equation written out with all its components is more suitable:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)} \right) \ , \quad i = 1, \ldots, n \ . \tag{8.37}$$

The computation of one component $x_i^{(k+1)}$, $i \in \{1, \ldots, n\}$, of the $(k + 1)$th approximation requires all components of the kth approximation vector x^k. Considering a sequential computation in the order $x_1^{(k+1)}, \ldots, x_n^{(k+1)}$, it can be observed that the values $x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$ are already known when $x_i^{(k+1)}$ is computed. This information is exploited in the Gauss-Seidel iteration method.

8.3.1.2 Gauss-Seidel iteration

The Gauss-Seidel iteration is based on the same splitting of the matrix A as the Jacobi iteration, i.e., $A = D - L - R$, but uses the splitting in a different way for an iteration

$$(D - L)x^{(k+1)} = Rx^{(k)} + b \ .$$

Thus, the iteration matrix of the Gauss-Seidel method is $C_{Ga} := (D - L)^{-1}R$; this form is used for numerical properties like convergence proofs, not shown here. The component form for the practical use is

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) , \quad i = 1, \ldots, n . \tag{8.38}$$

It can be seen that the components of $x_i^{(k+1)}$, $i \in \{1, \ldots, n\}$, use the new information $x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$ already determined for that approximation vector. This is useful for a faster convergence in a sequential implementation but the potential parallelism is now restricted.

8.3.1.3 Convergence criteria

For the Jacobi and the Gauss-Seidel iterations, the following convergence criteria based on the structure of A are often helpful. The Jacobi and the Gauss-Seidel iterations converge if the matrix A is **strongly diagonal dominant**, i.e.,

$$|a_{ii}| > \sum_{j=1, j \neq i}^{n} |a_{ij}| , \quad i = 1, \ldots, n .$$

When the absolute values of the diagonal elements are large compared to the sum of the absolute values of the other row elements, this often leads to a better convergence. Also, when the iteration methods converge, the Gauss-Seidel iteration often converges faster than the Jacobi iteration, since always the most recently computed vector components are used. Still the convergence is usually not fast enough for practical use. Therefore, an additional relaxation parameter is introduced to speed up the convergence.

8.3.1.4 JOR method

The JOR method or Jacobi over-relaxation is based on the splitting $A = \frac{1}{\omega} D - L - R - \frac{1-\omega}{\omega} D$ of the matrix A with a **relaxation parameter** $\omega \in \mathbb{R}$. The component form of this modification of the Jacobi method is

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^{n} a_{ij} x_j^{(k)} \right) + (1 - \omega) x_i^{(k)} , \quad i = 1, \ldots, n . \tag{8.39}$$

More popular is the modification with a relaxation parameter for the Gauss-Seidel method.

8.3.1.5 SOR method

The SOR method or (**successive over-relaxation**) is a modification of the Gauss-Seidel iteration that speeds up the convergence of the Gauss-Seidel method by introducing a relaxation parameter $\omega \in \mathbb{R}$. This parameter is used to modify the way in which the combination of the previous approximation $x^{(k)}$ and the components of the current approximation $x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$ are combined in the computation of $x_i^{(k+1)}$. The $(k+1)$th approximation computed according to the Gauss-Seidel iteration (8.38) is now considered as intermediate result $\hat{x}^{(k+1)}$ and the next approximation $x^{(k+1)}$ of the SOR method is computed from both vectors $\hat{x}^{(k+1)}$ and $x^{(k+1)}$ in the following way:

$$\hat{x}_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) , \quad i = 1, \ldots, n , \tag{8.40}$$

$$x_i^{(k+1)} = x_i^{(k)} + \omega (\hat{x}_i^{(k+1)} - x_i^{(k)}) , \quad i = 1, \ldots, n . \tag{8.41}$$

Substituting Eq. (8.40) into Eq. (8.41) results in the iteration:

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) + (1 - \omega) x_i^{(k)} \tag{8.42}$$

for $i = 1, \ldots, n$. The corresponding splitting of the matrix A is $A = \frac{1}{\omega} D - L - R - \frac{1-\omega}{\omega} D$ and an iteration step in matrix form is

$$(D - wL) x^{(k+1)} = (1 - \omega) D x^{(k)} + \omega R x^{(k)} + \omega b .$$

The convergence of the SOR method depends on the properties of A and the value chosen for the relaxation parameter ω. For example the following property holds: if A is symmetric and positive definite and $\omega \in (0, 2)$, then the SOR method converges for every start vector $x^{(0)}$. For more numerical properties see books on numerical linear algebra, e.g., [23, 67, 77, 186].

8.3.1.6 Implementation using matrix operations

The iteration (8.36) computing $x^{(k+1)}$ for a given vector $x^{(k)}$ consists of

- a matrix-vector multiplication of the iteration matrix C with $x^{(k)}$ and
- a vector–vector addition of the result of the multiplication with vector d.

The specific structure of the iteration matrix, i.e., C_{Ja} for the Jacobi iteration and C_{Ga} for the Gauss-Seidel iteration, is exploited. For the Jacobi iteration with $C_{Ja} = D^{-1}(L + R)$ this results in the computation steps:

- a matrix-vector multiplication of $L + R$ with $x^{(k)}$,
- a vector-vector addition of the result with b and
- a matrix-vector multiplication with D^{-1} (where D is a diagonal matrix and thus D^{-1} is easy to compute).

A sequential implementation uses Formula (8.37) and the components $x_i^{(k+1)}$, $i = 1, \ldots, n$, are computed one after another. The entire vector $x^{(k)}$ is needed for this computation. For the Gauss-Seidel iteration with $C_{Ga} = (D-L)^{-1}R$ the computation steps are:

- a matrix-vector multiplication $Rx^{(k)}$ with upper triangular matrix R,
- a vector–vector addition of the result with b and
- the solution of a linear system with lower triangular matrix $(D - L)$.

A sequential implementation uses formula (8.38). Since the most recently computed approximation components are always used for computing a value $x_i^{(k+1)}$, the previous value $x_i^{(k)}$ can be overwritten. The iteration method stops when the current approximation is close enough to the exact solution. Since this solution is unknown, the relative error is used for error control and after each iteration step the convergence is tested according to

$$\|x^{(k+1)} - x^{(k)}\| \leq \varepsilon \|x^{(k+1)}\| \tag{8.43}$$

where ε is a predefined error value and $\|.\|$ is a vector norm such as $\|x\|_\infty = \max_{i=1,\ldots,n} |x|_i$ or $\|x\|_2 = (\sum_{i=1}^{n} |x_i|^2)^{\frac{1}{2}}$.

8.3.2 Parallel implementation of the Jacobi Iteration

In the Jacobi iteration (8.37), the computations of the components $x_i^{(k+1)}$, $i = 1, \ldots, n$, of approximation $x^{(k+1)}$ are independent of each other and can be executed in parallel. Thus, each iteration step has a maximum degree of potential parallelism of n and $p = n$ processors can be employed. For a parallel system with distributed memory, the values $x_i^{(k+1)}$ are stored in the individual local memories. Since the computation of one of the components of the next approximation requires all components of the previous approximation, communication has to be performed to create a replicated distribution of $x^{(k)}$. This can be done by multi-broadcast operation.

When considering the Jacobi iteration built up of matrix and vector operations, a parallel implementation can use the parallel implementations introduced in Sect. 3.7. The iteration matrix C_{Ja} is not built up explicitly but matrix A is used without its diagonal elements. The parallel computation of the components of $x^{(k+1)}$ corresponds to the parallel implementation of the matrix-vector product using the parallelization with scalar products, see Sect. 3.7. The vector addition can be done after the multi-broadcast operation by each of the processors or before the multi-broadcast operation in a distributed way. When using the parallelization of the linear

```
int Parallel_jacobi(int n, int p, int max_it, float tol)
{
 int i_local, i_global, j, i;
 int n_local, it_num;
 float x_temp1[GLOB_MAX], x_temp2[GLOB_MAX], local_x[GLOB_MAX];
 float *x_old, *x_new, *temp;

 n_local = n/p; /* local blocksize */
 MPI_Allgather(local_b, n_local, MPI_FLOAT, x_temp1, n_local,
              MPI_FLOAT, MPI_COMM_WORLD);
 x_new = x_temp1;
 x_old = x_temp2;
 it_num = 0;
 do {
   it_num ++;
   temp = x_new; x_new = x_old; x_old = temp;
   for (i_local = 0; i_local < n_local; i_local++) {
     i_global = i_local + me * n_local;
     local_x[i_local] = local_b[i_local];
     for (j = 0; j < i_global; j++)
       local_x[i_local] = local_x[i_local] -
                          local_A[i_local][j] * x_old[j];
     for (j = i_global+1 ; j < n; j++)
       local_x[i_local] = local_x[i_local] -
                          local_A[i_local][j] * x_old[j];
     local_x[i_local] = local_x[i_local]/ local_A[i_local][i_global];
   }
   MPI_Allgather(local_x, n_local, MPI_FLOAT, x_new, n_local,
                MPI_FLOAT, MPI_COMM_WORLD);
 } while ((it_num < max it) && (distance(x_old,x_new,n) >= tol));
 output(x_new,global_x);
 if (distance(x_old, x_new, n) < tol ) return 1;
 else return 0;
}
```

Fig. 8.13 Program fragment in C notation and with MPI communication operations for a parallel implementation of the Jacobi iteration. The arrays `local_x`, `local_b` and `local_A` are declared globally. The dimension of `local_A` is `n_local` × `n`. A pointer oriented storage scheme as shown in Fig. 8.3 is not used here, so that the array indices in this implementation differ from the indices in a sequential implementation. The computation of `local_x[i_local]` is performed in two loops with loop index j; the first loop corresponds to the multiplication with array elements in row `i_local` to the left of the main diagonal of A and the second loop corresponds to the multiplication with array elements in row `i_local` to the right of the main diagonal of A. The result is divided by `local_A[i_local][i_global]` which corresponds to the diagonal element of that row in the global matrix A.

combination from Sect. 3.7, the vector addition takes place after the accumulation operation. The final broadcast operation is required to provide $x^{(k+1)}$ to all processors also in this case.

Figure 8.13 shows a parallel implementation of the Jacobi iteration using C notation and MPI operations from [153]. For simplicity it is assumed that the matrix size

n is a multiple of the number of processors p. The iteration matrix is stored in a row blockwise way, so that each processor owns n/p consecutive rows of matrix A which are stored locally in array `local_A`. The vector b is stored in a corresponding blockwise way. This means that the processor `me`, $0 \leq$ `me` $< p$, stores the rows `me` $\cdot\, n/p + 1, \ldots,$ (`me` $+ 1) \cdot n/p$ of A in `local_A` and the corresponding components of b in `local_b`. The iteration uses two local arrays `x_old` and `x_new` for storing the previous and the current approximation vectors. The symbolic constant `GLOB_MAX` is the maximum size of the linear equation system to be solved. The result of the local matrix-vector multiplication is stored in `local_x`; `local_x` is computed according to (8.37). An `MPI_Allgather()` operation combines the local results, so that each processor stores the entire vector `x_new`. The iteration stops when a predefined number `max_it` of iteration steps has been performed or when the difference between `x_old` and `x_new` is smaller than a predefined value `tol`. The function `distance()` implements a maximum norm and the function `output(x_new,global_x)` returns array `global_x` which contains the last approximation vector to be the final result.

8.3.3 Parallel Implementation of the Gauss-Seidel Iteration

The Gauss-Seidel iteration (8.38) exhibits data dependencies, since the computation of the component $x_i^{(k+1)}$, $i \in \{1, \ldots, n\}$, uses the components $x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$ of the same approximation and the components of $x^{(k+1)}$ have to be computed one after another. Since for each $i \in \{1, \ldots, n\}$, the computation (8.38) corresponds to a scalar product of the vector

$$(x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}, 0, x_{i+1}^{(k)}, \ldots, x_n^{(k)})$$

and the ith row of A, this means the scalar products have to be computed one after another. Thus, parallelism is only possible within the computation of each single scalar product: each processor can compute a part of the scalar product, i.e., a local scalar product, and the results are then accumulated. For such an implementation, a column blockwise distribution of matrix A is suitable. Again, we assume that n is a multiple of the numbers of processors p. The approximation vectors are distributed correspondingly in a blockwise way. Processor P_q, $1 \leq q \leq p$, computes that part of the scalar product for which it owns the columns of A and the components of the approximation vector $x^{(k)}$. This is the computation

$$s_{qi} = \sum_{\substack{j=(q-1)\cdot n/p+1 \\ j<i}}^{q\cdot n/p} a_{ij} x_j^{(k+1)} + \sum_{\substack{j=(q-1)\cdot n/p+1 \\ j>i}}^{q\cdot n/p} a_{ij} x_j^{(k)} \,. \tag{8.44}$$

```
n_local = n/p;
do {
   delta_x = 0.0;
   for (i = 0; i < n; i++) {
      s_k = 0.0;
      for (j = 0; j < n_local; j++)
         if (j + me * n_local != i)
           s_k = s_k + local A[i][j] * x[j];
      root = i/n_local;
      i_local = i % n_local;
      MPI_Reduce(&s_k, &x[i_local], 1, MPI_FLOAT, MPI_SUM, root,
                 MPI_COMM_WORLD);
      if (me == root) {
        x new = (b[i_local] - x[i_local]) / local_A[i][i_local];
        delta_x = max(delta_x, abs(x[i_local] - x new));
        x[i_local] = x_new;
      }
   }
   MPI_Allreduce(&delta_x, &global_delta, 1, MPI_FLOAT,
                 MPI_MAX, MPI_COMM_WORLD);
} while(global_delta > tol);
```

Fig. 8.14 Program fragment in C notation and using MPI operations for a parallel Gauss-Seidel iteration for a dense linear equation system. The components of the approximations are computed one after another according to Formula (8.38), but each of these computations is done in parallel by all processors. The matrix is stored in a column blockwise way in the local arrays `local_A`. The vectors x and b are also distributed blockwise. Each processor computes the local error and stores it in `delta_x`. An `MPI_Allreduce()` operation computes the global error `global_delta` from these values, so that each processor can perform the convergence test `global_delta > tol`.

The intermediate results s_{qi} computed by processors P_q, $q = 1, \ldots, p$, are accumulated by a single-accumulation operation with the addition as reduction operation and the value $x_i^{(k+1)}$ is the result. Since the next approximation vector $x^{(k+1)}$ is expected in a blockwise distribution, the value $x_i^{(k+1)}$ is accumulated at the processor owning the ith component, i.e., $x_i^{(k+1)}$ is accumulated by processor P_q with $q = \lceil i/(n/p) \rceil$. A parallel implementation of the SOR method corresponds to the parallel implementation of the Gauss-Seidel iteration, since both methods differ only in the additional relaxation parameter of the SOR method.

Figure 8.14 shows a program fragment using C notation and MPI operations of a parallel Gauss-Seidel iteration. Since only the most recently computed components of an approximation vector are used in further computations, the component $x_i^{(k)}$ is overwritten by $x_i^{(k+1)}$ immediately after its computation. Therefore, only one array `x` is needed in the program. Again, an array `local_A` stores the local part of matrix A which is a block of columns in this case; `n_local` is the size of the block. The `for`-loop with loop index i computes the scalar products sequentially; within the loop body the parallel computation of the inner product is performed according to

Formula (8.44). An MPI reduction operation computes the components at differing processors `root` which finalizes the computation.

8.3.4 Gauss-Seidel Iteration for Sparse Systems

The potential parallelism for the Gauss-Seidel iteration or the SOR method is limited because of data dependences, so that a parallel implementation is only reasonable for very large equation systems. Each data dependency in Formula (8.38) is caused by a coefficient (a_{ij}) of matrix A, since the computation of $x_i^{(k+1)}$ depends on the value $x_j^{(k+1)}, j < i$, when $(a_{ij}) \neq 0$. Thus, for a linear equation system $Ax = b$ with sparse matrix $A = (a_{ij})_{i,j=1,\ldots,n}$ there is a larger degree of parallelism caused by less data dependences. If $a_{ij} = 0$, then the computation of $x_i^{(k+1)}$ does not depend on $x_j^{(k+1)}, j < i$. For a sparse matrix with many zero elements the computation of $x_i^{(k+1)}$ only needs a few $x_j^{(k+1)}, j < i$. This can be exploited to compute components of the $(k+1)$th approximation $x^{(k+1)}$ in parallel.

In the following, we consider sparse matrices with a banded structure like the discretized Poisson equation, see Eq. (8.13) in Sect. 8.2.1. The computation of $x_i^{(k+1)}$ uses the elements in the ith row of A, see Fig. 8.9, which has nonzero elements a_{ij} for $j = i - \sqrt{n}, i-1, i, i+1, i+\sqrt{n}$. Formula (8.38) of the Gauss-Seidel iteration for the discretized Poisson equation has the specific form

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \Big(b_i - a_{i,i-\sqrt{n}} \cdot x_{i-\sqrt{n}}^{(k+1)} - a_{i,i-1} \cdot x_{i-1}^{(k+1)} - a_{i,i+1} \cdot x_{i+1}^{(k)} - a_{i,i+\sqrt{n}} \cdot x_{i+\sqrt{n}}^{(k)} \Big) , \quad i = 1, \ldots, n . \tag{8.45}$$

Thus, the two values $x_{i-\sqrt{n}}^{(k+1)}$ and $x_{i-1}^{(k+1)}$ have to be computed before the computation of $x_i^{(k+1)}$. The dependences of the values $x_i^{(k+1)}, i = 1, \ldots, n$, on $x_j^{(k+1)}, j < i$, are illustrated in Fig. 8.15 (a) for the corresponding mesh of the discretized physical domain. The computation of $x_i^{(k+1)}$ corresponds to the mesh point i, see also Sect. 8.2.1. In this mesh, the computation of $x_i^{(k+1)}$ depends on all computations for mesh points which are located in the upper left part of the mesh. On the other hand, computations for mesh points $j > i$ which are located to the right or below mesh point i need value $x_i^{(k+1)}$ and have to wait for its computation.

The data dependences between computations associated to mesh points are depicted in the mesh by arrows between the mesh points. It can be observed that the mesh points in each diagonal from left to right are independent of each other; these independent mesh points are shown in Fig. 8.15 (b). For a square mesh of size $\sqrt{n} \times \sqrt{n}$ with the same number of mesh points in each dimension, there are at most

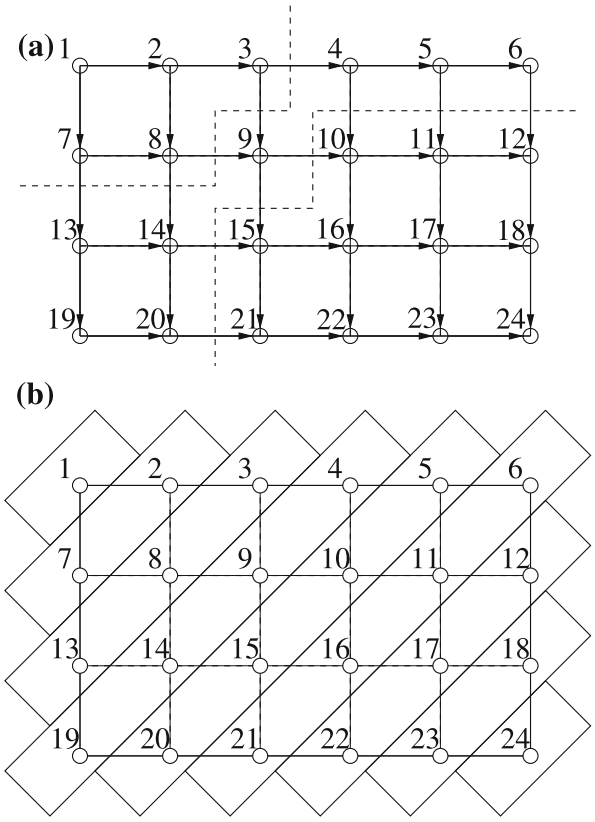

Fig. 8.15 Data dependence of the Gauss-Seidel and the SOR methods for a rectangular mesh of size 6×4 in the x-y plane. **(a)** The data dependences between the computations of components are depicted as arrows between nodes in the mesh. As an example, for mesh point 9 the set of nodes which have to be computed before point 9 and the set of nodes which depend on mesh point 9 are shown. **(b)** The data dependences lead to areas of independent computations; these are the diagonals of the mesh from the upper right to the lower left. The computations for mesh points within the same diagonal can be computed in parallel. The length of the diagonals is the degree of potential parallelism which can be exploited.

$\sqrt{n}$ independent computations in a single diagonal and at most $p = \sqrt{n}$ processors can be employed.

A parallel implementation can exploit the potential parallelism in a loop structure with an outer sequential loop and an inner parallel loop. The outer sequential loop visits the diagonals one after another from the upper left corner to the lower right corner. The inner loop exploits the parallelism within each diagonal of the mesh. The number of diagonals is $2\sqrt{n} - 1$ consisting of $\sqrt{n}$ diagonals in the upper left triangular mesh and $\sqrt{n} - 1$ in the lower triangular mesh. The first $\sqrt{n}$ diagonals $l = 1, \ldots, \sqrt{n}$ contain l mesh points i with

$$i = l + j \cdot (\sqrt{n} - 1) \quad \text{for} \quad 0 \leq j < l \; .$$

The last $\sqrt{n} - 1$ diagonals $l = 2, \ldots, \sqrt{n}$ contain $\sqrt{n} - l + 1$ mesh points i with

$$i = l \cdot \sqrt{n} + j \cdot (\sqrt{n} - 1) \quad \text{for} \quad 0 \leq j \leq \sqrt{n} - l \; .$$

For an implementation on a distributed memory machine, a distribution of the approximation vector x, the right-hand side b and the coefficient matrix A is needed. The elements a_{ij} of matrix A are distributed in such a way that the coefficients for the computation of $x_i^{(k+1)}$ according to Formula (8.45) are locally available. Because the

```
sqn = sqrt(n);
do {
   for (l = 1; l <= sqn; l++) {
      for (j = me; j < l; j+=p) {
         i = l + j * (sqn-1) - 1; /* start numbering with 0 */
         x[i] = 0;
         if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
         if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
         if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
         if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
         x[i] = (x[i] + b[i]) / a[i][i];
      }
      collect_elements(x,l);
   }
   for (l = 2; l <= sqn; l++) {
      for (j = me -l +1; j <= sqn -l; j+=p) {
         if (j >= 0) {
           i = l * sqn + j * (sqn-1) - 1;
           x[i] = 0;
           if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
           if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
           if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
           if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
           x[i] = (x[i] + b[i]) / a[i][i];
         }
      }
      collect_elements(x,l);
   }
} while(convergence_test() < tol);
```

Fig. 8.16 Program fragment of the parallel Gauss-Seidel iteration for a linear equation system with the banded matrix from the discretized Poisson equation. The computational structure uses the diagonals of the corresponding discretization mesh, see Fig. (8.15).

computations are closely related to the mesh, the data distribution is chosen for the mesh and not the matrix form.

The program fragment with C notation in Fig. 8.16 shows a parallel SPMD implementation. The data distribution is chosen such that the data associated to mesh points in the same mesh row are stored on the same processor. A row-cyclic distribution of the mesh data is used. The program has two loop nests: the first loop nest treats the upper diagonals and the second loop nest treats the last diagonals. In the inner loops, the processor with name `me` computes the mesh points which are assigned to it due to the row-cyclic distribution of mesh points. The function `collect_elements()` sends the data computed to the neighboring processor, which needs them for the computation of the next diagonal. The function `convergence_test()`, not expressed explicitly in this program, can be implemented similarly as in the program in Fig. 8.14 using the maximum norm for $x^{(k+1)} - x^{(k)}$.

The program fragment in Fig. 8.16 uses two-dimensional indices for accessing array elements of array `a`. For a large sparse matrix, a storage scheme for sparse matrices would be used in practice. Also, for a problem such as the discretized Poisson equation where the coefficients are known it is suitable to code them directly as constants into the program. This saves expensive array accesses but the code is less flexible to solve other linear equation systems.

For an implementation on a shared memory machine, the inner loop is performed in parallel by $p = \sqrt{n}$ processors in an SPMD pattern. No data distribution is needed but the same distribution of work to processors is assigned. Also, no communication is needed to send data to neighboring processors. However, a barrier synchronization is used instead to make sure that the data of the previous diagonal are available for the next one.

A further increase of the potential parallelism for solving sparse linear equation systems can be achieved by the method described in the next subsection.

8.3.5 Red-black Ordering

The potential parallelism of the Gauss-Seidel iteration or the successive overrelaxation for sparse systems resulting from discretization problems can be increased by an alternative ordering of the unknowns and equations. The goal of the reordering is to get an equivalent equation system in which more independent computations exist, and thus a higher potential parallelism results. The most frequently used reordering technique is the **red-black ordering**. The two-dimensional mesh is regarded as a checkerboard where the points of the mesh represent the squares of the checkerboard and get corresponding colors. The point (i, j) in the mesh is colored according to the value of $i + j$: if $i + j$ is even, then the mesh point is red, and if $i + j$ is odd, then the mesh point is black.

The points in the grid now form two sets of points. Both sets are numbered separately in a row-wise way from left to right. First, the red points are numbered by $1, \ldots, n_R$ where n_R is the number of red points. Then, the black points are numbered by $n_R + 1, \ldots, n_R + n_B$ where n_B is the number of black points and $n = n_R + n_B$. The unknowns associated with the mesh points get the same numbers as the mesh points: There are n_R unknowns associated with the red points denoted as $\hat{x}_1, \ldots, \hat{x}_{n_R}$ and n_B unknowns associated with the black points denoted as $\hat{x}_{n_R+1}, \ldots, \hat{x}_{n_R+n_B}$. (The notation $\hat{x}$ is used to distinguish the new ordering from the original ordering of the unknowns x. The unknowns are the same as before but their positions in the system differ.) Figure 8.17 shows a mesh of size 6×4 in its original row-wise numbering in part (a) and a red-black ordering with the new numbering in part (b).

In a linear equation system using red-black ordering, the equations of red unknowns are arranged before the equations with the black unknown. The equation system $\hat{A}\hat{x} = \hat{b}$ for the discretized Poisson equation has the form:

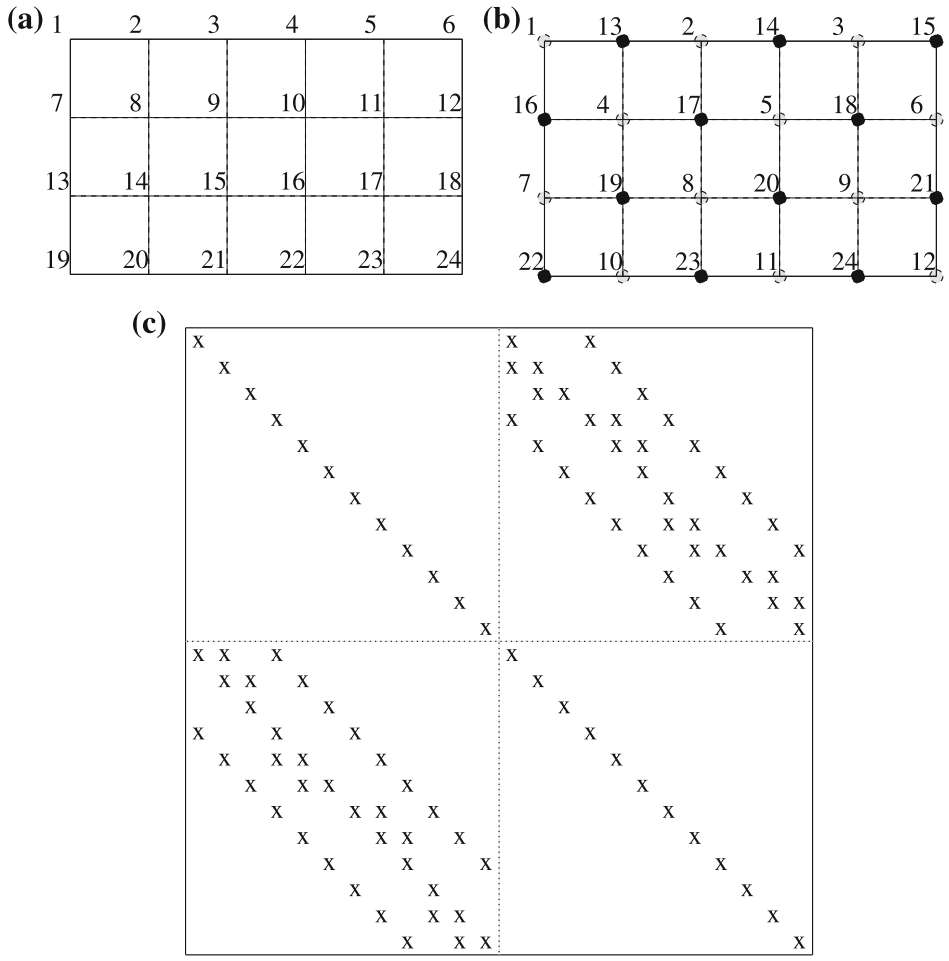

Fig. 8.17 Rectangular mesh in the x-y plane of size 6×4 with **(a)** row-wise numbering, **(b)** red-black numbering, and **(c)** the matrix of the corresponding linear equation system of the five-point formula with red-black numbering.

$$\hat{A} \cdot \hat{x} = \begin{pmatrix} D_R & F \\ E & D_B \end{pmatrix} \cdot \begin{pmatrix} \hat{x}_R \\ \hat{x}_B \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \end{pmatrix}, \tag{8.46}$$

where $\hat{x}_R$ denotes the subvector of size n_R of the first (red) unknowns and $\hat{x}_B$ denotes the subvector of size n_B of the last (black) unknowns. The right-hand side b of the original equation system is reordered accordingly and has subvector $\hat{b}_1$ for the first n_R equations and subvector $\hat{b}_2$ for the last n_B equations. The matrix $\hat{A}$ consists of four blocks $D_R \in \mathbb{R}^{n_R \times n_R}$, $D_B \in \mathbb{R}^{n_B \times n_B}$, $E \in \mathbb{R}^{n_B \times n_R}$ and $F \in \mathbb{R}^{n_R \times n_B}$. The submatrices D_R and D_B are diagonal matrices and the submatrices E and F are sparse banded matrices. The structure of the original matrix of the discretized Poisson equation in Fig. 8.9 in Sect.8.2.1 is thus transformed into a matrix $\hat{A}$ with the structure shown in Fig. 8.17 c).

The diagonal form of the matrix D_R and D_B shows that a red unknown $\hat{x}_i$, $i \in \{1, \ldots, n_R\}$, does not depend on the other red unknowns and a black unknown $\hat{x}_j$, $j \in$

$\{n_R + 1, ..., n_R + n_B\}$, does not depend on the other black unknowns. The matrices E and F specify the dependences between red and black unknowns. The row i of matrix F specifies the dependences of the red unknowns $\hat{x}_i$ $(i < n_R)$ on the black unknowns $\hat{x}_j, j = n_R + 1, ..., n_R + n_B$. Analogously, a row of matrix E specifies the dependences of the corresponding black unknowns on the red unknowns.

The transformation of the original linear equation system $Ax = b$ into the equivalent system $\hat{A}\hat{x} = \hat{b}$ can be expressed by a permutation $\pi : \{1, ..., n\} \to \{1, ..., n\}$. The permutation maps a node $i \in \{1, \ldots, n\}$ of the row-wise numbering onto the number $\pi(i)$ of the red-black numbering in the following way:

$$x_i = \hat{x}_{\pi(i)}, \quad b_i = \hat{b}_{\pi(i)}, \quad i = 1, ..., n \text{ or } x = P\hat{x} \text{ and } b = P\hat{b}$$

with a permutation matrix $P = (P_{ij})_{i,j=1,...,n}, P_{ij} = \begin{cases} 1 & \text{if } j = \pi(i) \\ 0 & \text{otherwise} \end{cases}$. For the matrices A and $\hat{A}$ the equation $\hat{A} = P^T AP$ holds. Since for a permutation matrix the inverse is equal to the transposed matrix, i.e., $P^T = P^{-1}$, this leads to $\hat{A}\hat{x} = P^T APP^T x = P^T b = \hat{b}$. The easiest way to exploit the red-black ordering is to use an iterative solution method as discussed earlier in this section.

8.3.5.1 Gauss-Seidel iteration for red-black systems

The solution of the linear equation system (8.46) with the Gauss-Seidel iteration is based on a splitting of the matrix $\hat{A}$ of the form $\hat{A} = \hat{D} - \hat{L} - \hat{U}, \hat{D}, \hat{L}, \hat{U} \in \mathbb{R}^{n \times n}$,

$$\hat{D} = \begin{pmatrix} D_R & 0 \\ 0 & D_B \end{pmatrix}, \quad \hat{L} = \begin{pmatrix} 0 & 0 \\ -E & 0 \end{pmatrix}, \quad \hat{U} = \begin{pmatrix} 0 & -F \\ 0 & 0 \end{pmatrix},$$

with a diagonal matrix $\hat{D}$, a lower triangular matrix $\hat{L}$, and an upper triangular matrix $\hat{U}$. The matrix 0 is a matrix in which all entries are 0. With this notation, iteration step k of the Gauss-Seidel method is given by

$$\begin{pmatrix} D_R & 0 \\ E & D_B \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k+1)} \\ x_B^{(k+1)} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} - \begin{pmatrix} 0 & F \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k)} \\ x_B^{(k)} \end{pmatrix} \tag{8.47}$$

for $k = 1, 2, \ldots$. According to equation system (8.46), the iteration vector is split into two subvectors $x_R^{(k+1)}$ and $x_B^{(k+1)}$ for the red and the black unknowns, respectively. (To simplify the notation, we use x_R instead of $\hat{x}_R$ in the following discussion of the red-black ordering.)

The linear equation system (8.47) can be written in vector notation for vectors $x_R^{(k+1)}$ and $x_B^{(k+1)}$ in the form

$$D_R \cdot x_R^{(k+1)} = b_1 - F \cdot x_B^{(k)} \quad \text{for } k = 1, 2, \ldots , \tag{8.48}$$

$$D_B \cdot x_B^{(k+1)} = b_2 - E \cdot x_R^{(k+1)} \quad \text{for } k = 1, 2, \ldots , \tag{8.49}$$

in which the decoupling of the red subvector $x_R^{(k+1)}$ and the black subvector $x_B^{(k+1)}$ becomes obvious: In Eq. (8.48) the new red iteration vector $x_R^{(k+1)}$ depends only on the previous black iteration vector $x_B^{(k)}$ and in Eq. (8.49) the new black iteration vector $x_B^{(k+1)}$ depends only on the red iteration vector $x_R^{(k+1)}$ computed before in the same iteration step. There is no additional dependence. Thus, the potential degree of parallelism in each of the equations of (8.48) or (8.49) is similar to the potential parallelism in the Jacobi iteration. In each iteration step k, the components of $x_R^{(k+1)}$ according to Eq. (8.48) can be computed independently, since the vector $x_B^{(k)}$ is known, which leads to a potential parallelism with $p = n_R$ processors. Afterwards, the vector $x_R^{(k+1)}$ is known and the components of the vector $x_B^{(k+1)}$ can be computed independently according to (8.49), leading to a potential parallelism of $p = n_R$ processors.

For a parallel implementation, we consider the Gauss-Seidel iteration of the red-black ordering (8.48) and (8.49) written out in a component-based form:

$$\left(x_R^{(k+1)}\right)_i = \frac{1}{\hat{a}_{ii}} \left(\hat{b}_i - \sum_{j \in N(i)} \hat{a}_{ij} \cdot (x_B^{(k)})_j \right), \quad i = 1, \ldots, n_R ,$$

$$\left(x_B^{(k+1)}\right)_i = \frac{1}{\hat{a}_{i+n_R, i+n_R}} \left(\hat{b}_{i+n_R} - \sum_{j \in N(i)} \hat{a}_{i+n_R, j} \cdot (x_R^{(k+1)})_j \right), \; i = 1, \ldots, n_B .$$

The set $N(i)$ denotes the set of adjacent mesh points for mesh point i. According to the red-black ordering, the set $N(i)$ contains only black mesh points for a red point i and vice versa. An implementation on a shared memory machine can employ at most $p = n_R$ or $p = n_B$ processors. There are no access conflicts for the parallel computation of $x_R^{(k)}$ or $x_B^{(k)}$ but a barrier synchronization is needed between the two computation phases. The implementation on a distributed memory machine requires a distribution of computation and data. As discussed before for the parallel SOR method, it is useful to distribute the data according to the mesh structure, such that the processor P_q to which the mesh point i is assigned is responsible for the computation or update of the corresponding component of the approximation vector. In a row-oriented distribution of a squared mesh with $\sqrt{n} \times \sqrt{n} = n$ mesh points to p processors, $\sqrt{n}/p$ rows of the mesh are assigned to each processor $P_q, q \in \{1, \ldots, p\}$. In the red-black coloring this means that each processor owns $\frac{1}{2}\frac{n}{p}$ red and $\frac{1}{2}\frac{n}{p}$ black mesh points. (For simplicity we assume that $\sqrt{n}$ is a multiple of p.) Thus, the mesh points

```
local_nr = nr/p; local_nb = nb/p;
do {
   mestartr = me * local_nr;
   for  (i= mestartr; i < mestartr + local_nr; i++) {
       xr[i] = 0;
       for  (j ∈ N(i))
           xr[i] = xr[i] - a[i][j] * xb[j];
       xr[i] = (xr[i]+b[i]) / a[i][i] ;
   }
   collect_elements(xr);
   mestartb = me * local_nb + nr;
   for (i= mestartb; i < mestartb + local_nb; i++) {
       xb[i] = 0;
       for (j ∈ N(i))
           xb[i] = xb[i] - a[i+nr][j] * xr[j];
       xb[i]= (xb[i] + b[i+nr]) / a[i+nr][i+nr];
   }
   collect_elements(xb);
} while (convergence_test());
```

Fig. 8.18 Program fragment for the parallel implementation of the Gauss-Seidel method with the red-black ordering. The arrays `xr` and `xb` denote the unknowns corresponding to the red or black mesh points. The processor number of the executing processor is stored in `me`.

$$(q-1)\cdot\frac{n_R}{p}+1,\ldots,q\cdot\frac{n_R}{p}\quad\text{for } q=1,\ldots,p\quad\text{and}$$
$$(q-1)\cdot\frac{n_B}{p}+1+n_R,\ldots,q\cdot\frac{n_B}{p}+n_R\quad\text{for}\quad q=1,\ldots,p$$

are assigned to processor P_q. Figure 8.18 shows an SPMD program implementing the Gauss-Seidel iteration with red-black ordering. The coefficient matrix A is stored according to the pointer-based scheme introduced earlier in Fig. 8.3. After the computation of the red components `xr`, a function `collect_elements(xr)` distributes the red vector to all other processors for the next computation. Analogously, the black vector `xb` is distributed after its computation. The function `collect_elements()` can be implemented by a multi-broadcast operation.

8.3.5.2 SOR method for red-black systems

An SOR method for the linear equation system (8.46) with relaxation parameter ω can be derived from the Gauss-Seidel computation (8.48) and (8.49) by using the combination of the new and the old approximation vector as introduced in Formula (8.41). One step of the SOR method has then the form:

$$
\begin{aligned}
\tilde{x}_R^{(k+1)} &= D_R^{-1} \cdot b_1 - D_R^{-1} \cdot F \cdot x_B^{(k)} \ , \\
\tilde{x}_B^{(k+1)} &= D_B^{-1} \cdot b_2 - D_B^{-1} \cdot E \cdot x_R^{(k+1)} \ , \\
x_R^{(k+1)} &= x_R^{(k)} + \omega \left(\tilde{x}_R^{(k+1)} - x_R^{(k)} \right) \ , \\
x_B^{(k+1)} &= x_B^{(k)} + \omega \left(\tilde{x}_B^{(k+1)} - x_B^{(k)} \right) \ , \quad k = 1, 2, \ldots \ .
\end{aligned}
\tag{8.50}
$$

The corresponding splitting of matrix $\hat{A}$ is $\hat{A} = \frac{1}{\omega}\hat{D} - \hat{L} - \hat{U} - \frac{1-\omega}{\omega}\hat{D}$ with the matrices $\hat{D}, \hat{L}, \hat{U}$ introduced above. This can be written using block matrices:

$$
\begin{aligned}
&\begin{pmatrix} D_R & 0 \\ \omega E & D_B \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k+1)} \\ x_B^{(k+1)} \end{pmatrix} \\
&= (1-\omega) \begin{pmatrix} D_R & 0 \\ 0 & D_B \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k)} \\ x_B^{(k)} \end{pmatrix} - \omega \begin{pmatrix} 0 & F \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k)} \\ x_B^{(k)} \end{pmatrix} + \omega \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}
\end{aligned}
\tag{8.51}
$$

For a parallel implementation, the component form of this system is used. On the other hand, for the convergence results the matrix form and the iteration matrix have to be considered. Since the iteration matrix of the SOR method for a given linear equation system $Ax = b$ with a certain order of the equations and the iteration matrix of the SOR method for the red-black system $\hat{A}\hat{x} = \hat{b}$ are different, convergence results cannot be transferred. The iteration matrix of the SOR method with red-black ordering is:

$$
\hat{S}_\omega = \left(\frac{1}{\omega}\hat{D} - \hat{L} \right)^{-1} \left(\frac{1-\omega}{\omega}\hat{D} + \hat{U} \right).
$$

For a convergence of the method it has to be shown that $\rho(\hat{S}_\omega) < 1$ for the spectral radius of $\hat{S}_\omega$ and $\omega \in \mathbb{R}$. In general, the convergence cannot be derived from the convergence of the SOR method for the original system, since $P^T S_\omega P$ is not identical to $\hat{S}_\omega$, although $P^T A P = \hat{A}$ holds. However, for the specific case of the model problem, i.e., the discretized Poisson equation, the convergence can be shown. Using the equality $P^T A P = \hat{A}$, it follows that $\hat{A}$ is symmetric and positive definite, and thus the method converges for the model problem, see [67].

Figure 8.19 shows a parallel SPMD implementation of the SOR method for the red-black ordered discretized Poisson equation. The elements of the coefficient matrix are coded as constants. The unknowns are stored in a two-dimensional structure corresponding to the two-dimensional mesh and not as vector, so that unknowns appear as `x[i][j]` in the program. The mesh points and the corresponding computations are distributed among the processors; the mesh points belonging to a specific processor are stored in `myregion`. The color red or black of a mesh point (i, j) is an additional attribute which can be retrieved by the functions `is_red()` and `is_black()`. The value `f[i][j]` denotes the discretized right-

```
do {
   for ((i,j) ∈ myregion) {
      if (is_red(i,j))
        x[i][j] = omega/4 * (h*h*f[i][j] + x[i][j-1] + x[i][j+1]
                  + x[i-1][j] + x[i+1][j]) + (1- omega) * x[i][j];
   }
   exchange_red_borders(x);
   for ((i,j) ∈ myregion) {
      if (is_black(i,j))
        x[i][j] = omega/4 * (h*h*f[i][j] + x[i][j-1] + x[i][j+1]
                  + x[i-1][j] + x[i+1][j]) + (1- omega) * x[i][j];
   }
   exchange_black_borders(x);
} while (convergence_test());
```

Fig. 8.19 Program fragment of a parallel SOR method for a red-black ordered discretized Poisson equation.

hand side of the Poisson equation as described earlier, see Eq. (8.15). The functions `exchange_red_borders()` and `exchange_black_borders()` exchange the red or black data of the red or black mesh points between neighboring processors.

8.4 Conjugate Gradient Method

The conjugate gradient method or CG method is a solution method for linear equation systems $Ax = b$ with symmetric and positive definite matrix $A \in \mathbb{R}^{n \times n}$, which has been introduced in [96]. (A is symmetric if $a_{ij} = a_{ji}$ and positive definite if $x^T Ax > 0$ for all $x \in \mathbb{R}^n$ with $x \neq 0$.) The CG method builds up a solution $x^* \in \mathbb{R}^n$ in at most n steps in the absence of roundoff errors. Considering roundoff errors more than n steps may be needed to get a good approximation of the exact solution x^*. For sparse matrices, a good approximation of the solution can be achieved in less than n steps, also with roundoff errors [168]. In practice, the CG method is often used as preconditioned CG method which combines a CG method with a preconditioner [172]. Parallel implementations are discussed in [78, 151, 152, 172]; [173] gives an overview. In this section, we present the basic CG method and parallel implementations according to [23, 77, 186].

8.4.1 Sequential CG method

The CG method exploits an equivalence between the solution of a linear equation system and the minimization of a function.

More precisely, the solution x^* of the linear equation system $Ax = b$, $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$, is the minimum of the function $\Phi : M \subset \mathbb{R}^n \to \mathbb{R}$ with

$$\Phi(x) = \frac{1}{2}x^T Ax - b^T x \ , \tag{8.52}$$

if the matrix A is symmetric and positive definite. A simple method to determine the minimum of the function Φ is the method of the steepest gradient [77] which uses the negative gradient. For a given point $x_c \in \mathbb{R}^n$, the function decreases most rapidly in the direction of the negative gradient. The method computes the following two steps.

(a) Computation of the negative gradient $d_c \in \mathbb{R}^n$ at point x_c,

$$d_c = -\text{grad}\,\Phi(x_c) = -\left(\frac{\partial}{\partial x_1}\Phi(x_c), \ldots, \frac{\partial}{\partial x_n}\Phi(x_c)\right) = b - Ax_c$$

(b) Determination of the minimum of Φ in the set

$$\{x_c + td_c \mid t \geq 0\} \cap M \ ,$$

which forms a line in $\mathbb{R}^n$ (line search). This is done by inserting $x_c + td_c$ into Formula (8.52). sing $d_c = b - Ax_c$ and the symmetry of matrix A we get

$$\Phi(x_c + td_c) = \Phi(x_c) - td_c^T d_c + \frac{1}{2}t^2 d_c^T Ad_c \ . \tag{8.53}$$

The minimum of this function with respect to $t \in \mathbb{R}$ can be determined using the derivative of this function with respect to t. The minimum is

$$t_c = \frac{d_c^T d_c}{d_c^T Ad_c} \tag{8.54}$$

The steps (a) and (b) of the method of the steepest gradient are used to create a sequence of vectors $x_k, k = 0, 1, 2, \ldots$, with $x_0 \in \mathbb{R}^n$ and $x_{k+1} = x_k + t_k d_k$. The sequence $(\Phi(x_k))_{k=0,1,2,\ldots}$ is monotonically decreasing which can be seen by inserting Formula (8.54) into Formula (8.53). The sequence converges toward the minimum but the convergence might be slow [77].

The CG method uses a technique to determine the minimum which exploits orthogonal search directions in the sense of **conjugate** or A-**orthogonal** vectors d_k. For a given matrix A, which is symmetric and nonsingular, two vectors $x, y \in \mathbb{R}^n$ are called conjugate or A-orthogonal, if $x^T Ay = 0$. If A is positive definite, k pairwise conjugate vectors $d_0, \ldots, d_{n-1}$ (with $d_i \neq 0, i = 0, \ldots, k-1$ and $k \leq n$) are linearly independent [23]. Thus, the unknown solution vector x^* of $Ax = b$ can be represented as a linear combination of the conjugate vectors $d_0, \ldots, d_{n-1}$, i.e.,

$$x^* = \sum_{k=0}^{n-1} t_k d_k. \tag{8.55}$$

Since the vectors are orthogonal, $d_k^T A x^* = \sum_{l=0}^{n-1} d_k^T A t_l d_l = t_k d_k^T A d_k$. This leads to

$$t_k = \frac{d_k A x^*}{d_k^T A d_k} = \frac{d_k^T b}{d_k^T A d_k}$$

for the coefficients t_k. Thus, when the orthogonal vectors are known, the values t_k, $k = 0, \ldots, n-1$, can be computed from the right- hand side b.

The algorithm for the CG method uses a representation

$$x^* = x_0 + \sum_{i=0}^{n-1} \alpha_i d_i, \tag{8.56}$$

of the unknown solution vector x^* as a sum of a starting vector x_0 and a term $\sum_{i=0}^{n-1} \alpha_i d_i$ to be computed. The second term is computed recursively by

$$x_{k+1} = x_k + \alpha_k d_k, \quad k = 1, 2, \ldots, \quad \text{with} \tag{8.57}$$

$$\alpha_k = \frac{-g_k^T d_k}{d_k^T A d_k} \quad \text{and} \quad g_k = A x_k - b\ . \tag{8.58}$$

The Formulas (8.57) and (8.58) determine x^* according to (8.56) by computing α_i and adding $\alpha_i d_i$ in each step, $i = 1, 2, \ldots$. Thus, the solution is computed after at most n steps. If not all directions d_k are needed for x^*, less than n steps are required.

Algorithms implementing the CG method do not choose the conjugate vectors $d_0, \ldots, d_{n-1}$ before but compute the next conjugate vector from the given gradient g_k by adding a correction term. The basic algorithm for the CG method is given in Fig. 8.20.

8.4.2 Parallel CG Method

The parallel implementation of the CG method is based on the algorithm given in Fig. 8.20. Each iteration step of this algorithm implementing the CG method consists of the following basic vector and matrix operations.

8.4.2.1 Basic operations of the CG algorithm

The basic operations of the CG algorithm are:

(1) a matrix-vector multiplication Ad_k,

Select $x_0 \in \mathbb{R}^n$
Set $d_0 = -g_0 = b - Ax_0$
While ($\|g_k\| > \varepsilon$) compute for $k = 0, 1, 2, \ldots$

(1) $w_k = Ad_k$

(2) $\alpha_k = \dfrac{g_k^T g_k}{d_k^T w_k}$

(3) $x_{k+1} = x_k + \alpha_k d_k$

(4) $g_{k+1} = g_k + \alpha_k w_k$

(5) $\beta_k = \dfrac{g_{k+1}^T g_{k+1}}{g_k^T g_k}$

(6) $d_{k+1} = -g_{k+1} + \beta_k d_k$

Fig. 8.20 Algorithm of the CG method. (1) and (2) compute the values α_k according to Eq. (8.58). The vector w_k is used for the intermediate result Ad_k. (3) is the computation given in Formula (8.57). (4) computes g_{k+1} for the next iteration step according to Formula (8.58) in a recursive way. $g_{k+1} = Ax_{k+1} - b = A(x_k + \alpha_k d_k) - b = g_k + A\alpha_k d_k$. This vector g_{k+1} represents the error between the approximation x_k and the exact solution. (5) and (6) compute the next vector d_{k+1} of the set of conjugate gradients.

(2) two scalar products $g_k^T g_k$ and $d_k^T w_k$,
(3) a so-called *axpy*-operation $x_k + \alpha_k d_k$,
(The name *axpy* comes from *a x* plus *y* describing the computation.)
(4) an *axpy*-operation $g_k + \alpha_k w_k$,
(5) a scalar product $g_{k+1}^T g_{k+1}$, and
(6) an *axpy*-operation $-g_{k+1} + \beta_k d_k$.

The result of $g_k^T g_k$ is needed in two consecutive steps and so the computation of one scalar product can be avoided by storing $g_k^T g_k$ in the scalar value γ_k. Since there are mainly one matrix-vector product and scalar products, a parallel implementation can be based on parallel versions of these operations.

Like the CG method many algorithms from linear algebra are built up from basic operations like matrix-vector operations or *axpy*-operations and efficient implementations of these basic operations lead to efficient implementations of entire algorithms. The **BLAS** (*Basic Linear Algebra Subroutines*) library offers efficient implementations for a large set of basic operations. This includes many *axpy*-operations which denote that a vector x is multiplied by a scalar value a and then added to another vector y. The prefixes s in saxpy or d daxpy denote axpy operations for *simple precision* and *double precision*, respectively. Introductory descriptions of the BLAS library are given in [49] or [66]. A standard way to parallelize algorithms for linear algebra is to provide efficient parallel implementations of the BLAS operations and to build up a parallel algorithm from these basic parallel operations. This technique is ideally suited for the CG method since it consists of such basic operations.

Here, we consider a parallel implementation based on the parallel implementations for matrix-vector multiplication or scalar product for distributed memory machines as presented in Sect. 3. These parallel implementations are based on a data distribution of the matrix and the vectors involved. For an efficient implementation of the CG

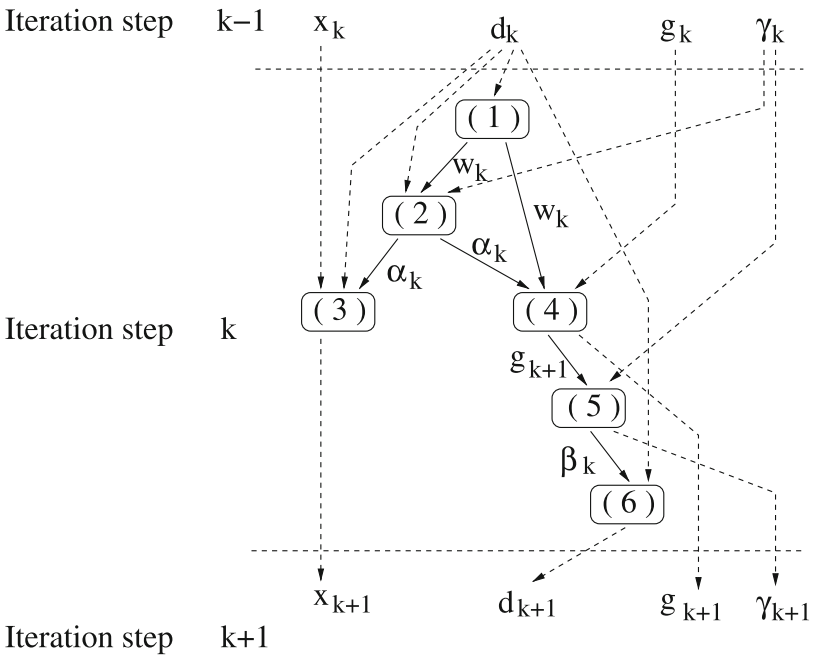

Fig. 8.21 Data dependences between the computation steps (1)–(6) of the CG method in Fig. 8.20. Nodes represent the computation steps of one iteration step k. Incoming arrows are annotated by the data required and outgoing arrows are annotated by the data produced. Two nodes have an arrow between them if one of the nodes produces data which are required by the node with the incoming arrow. The data dependences to the previous iteration step $k-1$ or the next iteration step $k+1$ are given as dashed arrows. The data are named in the same way as in Fig. 8.20; additionally the scalar γ_k is used for the intermediate result $\gamma_k = g_k^T g_k$ computed in step (5) and required for the computations of α_k and β_k in computation steps (2) and (5) of the next iteration step.

method, it is important that the data distributions of different basic operations fit together in order to avoid expensive data re-distributions between the operations. Figure 8.21 shows a data dependence graph in which the nodes correspond to the computation steps (1)–(6) of the CG algorithm in Fig. 8.20 and the arrows depict a data dependency between two of these computation steps. The arrows are annotated with data structures computed in one step (outgoing arrow) and needed for another step with incoming arrow. The data dependence graph for one iteration step k is a directed acyclic graph (DAG). There are also data dependences to the previous iteration step $k-1$ and the next iteration step $k+1$, which are shown as dashed arrows.

There are the following dependences in the CG method: The computation (2) needs the result w_k from computation (1) but also the vector d_k and the scalar value γ_k from the previous iteration step $k-1$; γ_k is used to store the intermediate result $\gamma_k = g_k^T g_k$. Computation (3) needs α_k from computation step (2) and the vectors x_k, d_k from the previous iteration step $k-1$. Computation (4) also needs α_k from computation step (2) and vector w_k from computation (1). Computation (5) needs vector g_{k+1} from computation (4) and scalar value γ_k from the previous iteration step $k-1$; computation (6) needs the scalar value from β_k from computation (5) and

vector d_k from iteration step $k-1$. This shows that there are many data dependences between the different basic operations. But it can also be observed that computation (3) is independent from the computations (4)–(6). Thus, the computation sequence (1),(2),(3),(4),(5),(6) as well as the sequence (1),(2),(4),(5),(6),(3) can be used. The independence of computation (3) from computations (4)–(6) is also another source of parallelism, which is a coarse-grain parallelism of two linear algebra operations performed in parallel, in contrast to the fine-grained parallelism exploited for a single basic operation. In the following, we concentrate on the fine-grained parallelism of basic linear algebra operations.

When the basic operations are implemented on a distributed memory machine, the data distribution of matrices and vectors and the data dependences between operations might require data re-distribution for a correct implementation. Thus, the data dependence graph in Fig. 8.21 can also be used to study the communication requirements for re-distribution in a message-passing program. Also the data dependences between two iteration steps may lead to communication for data re-distribution.

To demonstrate the communication requirements, we consider an implementation of the CG method in which the matrix A has a row-wise block-wise distribution and the vectors d_k, ω_k, g_k, x_k and r_k have a block-wise distribution. In one iteration step of a parallel implementation, the following computation and communication operations are performed.

8.4.2.2 Parallel CG implementation with block-wise distribution

The parallel CG implementation has to consider data distributions in the following way:

(0) Before starting the computation of iteration step k, the vector d_k computed in the previous step has to be re-distributed from a block-wise distribution of step $k-1$ to a replicated distribution required for step k. This can be done with a multi-broadcast operation.

(1) The matrix-vector multiplication $w_k = Ad_k$ is implemented with a row block-wise distribution of A as described in Sect. 3.7. Since d_k is now replicated, no further communication is needed. The result vector w_k is distributed in a block-wise way.

(2) The scalar products $d_k^T w_k$ are computed in parallel with the same block-wise distribution of both vectors. (The scalar product $\gamma_k = g_k^T g_k$ is computed in the previous iteration step.) Each processor computes a local scalar product for its local vectors. The final scalar product is then computed by the root processor of a single-accumulation operation with the addition as reduction operation. This processor owns the final result α_k and sends it to all other processors by a single-broadcast operation.

(3) The scalar value α_k is known by each processor and thus the *axpy*-operation $x_{k+1} = x_k + \alpha_k d_k$ can be done in parallel without further communication. Each

processor performs the arithmetic operations locally and the vector x_{k+1} results in a block-wise distribution.

(4) The *axpy*-operation $g_{k+1} = g_k + \alpha_k w_k$ is computed analogously to computation step (3) and the result vector g_{k+1} is distributed in a block-wise way.

(5) The scalar products $\gamma_{k+1} = g_{k+1}^T g_{k+1}$ are computed analogously to computation step (2). The resulting scalar value β_k is computed by the root processor of a single-accumulation operation and then broadcasted to all other processors.

(6) The *axpy*-operation $d_{k+1} = -g_{k+1} + \beta_k d_k$ is computed analogously to computation step (3). The result vector d_{k+1} has a block-wise distribution.

8.4.2.3 Parallel execution time

The parallel execution time of one iteration step of the CG method is the sum of the parallel execution times of the basic operations involved. We derive the parallel execution time for p processors; n is the system size. It is assumed that n is a multiple of p. The parallel execution time of one *axpy*-operation is given by

$$T_{axpy} = 2 \cdot \frac{n}{p} \cdot t_{op} \ ,$$

since each processor computes n/p components and the computation of each component needs one multiplication and one addition. As in earlier sections, the time for one arithmetic operation is denoted by t_{op}. The parallel execution time of a scalar product is

$$T_{scal\_prod} = 2 \cdot \left(\frac{n}{p} - 1 \right) \cdot t_{op} + T_{acc}(+)(p, 1) + T_{sb}(p, 1) \ ,$$

where $T_{acc}(op)(p, m)$ denotes the communication time of a single-accumulation operation with reduction operation op on p processors and message size m. The computation of the local scalar products with n/p components requires n/p multiplications and $n/p - 1$ additions. The distribution of the result of the parallel scalar product, which is a scalar value, i.e., has size 1, needs the time of a single-broadcast operation $T_{sb}(p, 1)$. The matrix-vector multiplication needs time

$$T_{math\_vec\_mult} = 2 \cdot \frac{n^2}{p} \cdot t_{op} \ ,$$

since each processor computes n/p scalar products. The total computation time of the CG method is

$$T_{CG} = T_{mb}(p, \frac{n}{p}) + T_{math\_vec\_mult} + 2 \cdot T_{scal\_prod} + 3 \cdot T_{axpy},$$

where $T_{mb}(p, m)$ is the time of a multi-broadcast operation with p processors and message size m. This operation is needed for the re-distribution of the direction vector d_k from iteration step k.

8.5 Cholesky Factorization for Sparse Matrices

Linear equation systems arising in practice are often large but have sparse coefficient matrices, i.e., they have many zero entries. For sparse matrices with regular structure, like banded matrices, only the diagonals with nonzero elements are stored and the solution methods introduced in the previous sections can be used. For an unstructured pattern of nonzero elements in sparse matrices, however, a more general storage scheme is needed and other parallel solution methods are applied. In this section, we consider the Cholesky factorization as an example of such a solution method. The general sequential factorization algorithm and its variants for sparse matrices are introduced in Sect. 8.5.1. A specific storage scheme for sparse unstructured matrices is given in Sect. 8.5.2. In Sect. 8.5.3, we discuss parallel implementations of sparse Cholesky factorization for shared memory machines.

8.5.1 Sequential Algorithm

The Cholesky factorization is a direct solution method for a linear equation system $Ax = b$. The method can be used if the coefficient matrix $A = (a_{ij}) \in \mathbb{R}^{n\times n}$ is symmetric and positive definite, i.e., if $a_{ij} = a_{ji}$ and $x^T Ax > 0$ for all $x \in \mathbb{R}^n$ with $x \neq 0$. For a symmetric and positive definite $n \times n$-matrix $A \in \mathbb{R}^{n\times n}$ there exists a unique triangular factorization

$$A = LL^T \tag{8.59}$$

where $L = (l_{ij})_{i,j=1,\ldots,n}$ is a lower triangular matrix, i.e., $l_{ij} = 0$ for $i < j$ and $i, j \in \{1, \ldots, n\}$, with positive diagonal elements, i.e., $l_{ii} > 0$ for $i = 1, \ldots, n$; L^T denotes the transposed matrix of L, i.e., $L^T = (l^T_{ij})_{i,j=1,\ldots,n}$ with $l^T_{ij} = l_{ji}$ [186]. Using the factorization in Eq. (8.59), the solution x of a system of equations $Ax = b$ with $b \in \mathbb{R}^n$ is determined in two steps by solving the triangular systems $Ly = b$ and $L^T x = y$ one after another. Because of $Ly = LL^T x = Ax = b$, the vector $x \in \mathbb{R}^n$ is the solution of the given linear equation system.

The implementation of the Cholesky factorization can be derived from a column-wise formulation of $A = LL^T$. Comparing the elements of A and LL^T, we obtain:

$$a_{ij} = \sum_{k=1}^{n} l_{ik} l^T_{kj} = \sum_{k=1}^{n} l_{ik} l_{jk} = \sum_{k=1}^{j} l_{ik} l_{jk} = \sum_{k=1}^{j} l_{jk} l_{ik}$$