Parallel Geometric Multigrid

Frank Hülsemann[1], Markus Kowarschik[1], Marcus Mohr[2], and Ulrich Rüde[1]

[1] System Simulation Group, University of Erlangen, Germany
[frank.huelsemann,markus.kowarschik,ulrich.ruede]@cs.fau.de
[2] Department for Sensor Technology, University of Erlangen, Germany
marcus.mohr@lse.eei.uni-erlangen.de

Summary. Multigrid methods are among the fastest numerical algorithms for the solution of large sparse systems of linear equations. While these algorithms exhibit asymptotically optimal computational complexity, their efficient parallelisation is hampered by the poor computation-to-communication ratio on the coarse grids. Our contribution discusses parallelisation techniques for geometric multigrid methods. It covers both theoretical approaches as well as practical implementation issues that may guide code development.

1 Overview

Multigrid methods are among the fastest numerical algorithms for solving large sparse systems of linear equations that arise from appropriate discretisations of elliptic PDEs. Much research has focused and will continue to focus on the design of multigrid algorithms for a variety of application areas. Hence, there is a large and constantly growing body of literature. For detailed introductions to multigrid we refer to the earlier publications [7, 31] and to the comprehensive overview provided in [57]. A detailed presentation of the multigrid idea is further given in [10]. A long list of multigrid references, which is constantly being updated, can be found in the BibTEX file `mgnet.bib` [16].

Multigrid methods form a family of iterative algorithms for large systems of linear equations which is characterised by asymptotically optimal complexity. For a large class of elliptic PDEs, multigrid algorithms can be devised which require $\mathcal{O}(n)$ floating-point operations in order to solve the corresponding linear system with n degrees of freedom up to discretisation accuracy. In the case of parabolic PDEs, appropriate discretisations of the time derivative lead to series of elliptic problems. Hence, the application of multigrid methods to parabolic equations is straightforward.

Various other linear solvers such as Krylov subspace methods (e.g., the method of conjugate gradients and GMRES), for example, mainly consist of matrix-vector products as well as inner products of two vectors [22, 32]. Their

parallel implementation is therefore quite straightforward. The parallelisation of multigrid algorithms tends to be more involved. This is primarily due to the necessity to handle problems of different mesh resolutions which thus comprise significantly varying numbers of unknowns.

In this article, we will focus on parallelisation approaches for geometric multigrid algorithms; see also [15, 43] and the tutorial on parallel multigrid methods by Jones that can further be found at `http://www.mgnet.org`. We assume that each level of accuracy is represented by a computational grid which is distributed among the parallel resources (i.e., the processes) of the underlying computing environment. We further suppose that the processes communicate with each other via message passing.

Our contribution is structured as follows. In Sect. 2 we will present a brief and general introduction to geometric multigrid schemes. Section 3 describes elementary parallelisation techniques for multigrid algorithms. The case of multigrid methods for applications involving unstructured finite-element meshes is more complicated than the case of structured meshes and will be addressed subsequently in Sect. 4. Section 5 focuses on the optimisation of the single-node performance, which primarily covers the improvement of the utilisation of memory hierarchies. Advanced parallelisation approaches for multigrid will be discussed afterwards in Sect. 6. Conclusions will be drawn in Sect. 7.

2 Introduction to Multigrid

2.1 Overview

Generally speaking, all multigrid algorithms follow the same fundamental design principle. A given problem is solved by integrating different levels of resolution into the solution process. During this process, the contributions of the individual levels are combined appropriately in order to form the required solution.

In the classical sense, multigrid methods involve a hierarchy of computational grids of different mesh widths and can therefore be considered geometrically motivated. This approach has led to the notion of *geometric multigrid (GMG)*. In contrast, later research has additionally addressed the development and the analysis of *algebraic multigrid (AMG)* methods, which target a multigrid-like iterative solution of linear systems without using geometric information from a grid hierarchy, but only the original linear system itself[3]. For an introduction to parallel AMG, see Chap. **??**.

[3]The term *algebraic multigrid* may thus appear misleading, since an ideal AMG approach would dispense with any computational grid.

2.2 Preparations

We will first motivate the principles of a basic geometric multigrid scheme. For simplicity, we consider the example of a scalar elliptic boundary value problem

$$Lu = f \quad \text{in} \quad \Omega \ , \tag{1}$$

defined on the interval of unit length (i.e., $\Omega := (0,1)$), on the unit square (i.e., $\Omega := (0,1)^2$), or on the unit cube (i.e., $\Omega := (0,1)^3$). L denotes a second-order linear elliptic differential operator, the solution of (1) is denoted as $u : \Omega \to \mathbb{R}$, and the function $f : \Omega \to \mathbb{R}$ represents the given right-hand side.

We assume Dirichlet boundary conditions only; i.e.,

$$u = g \quad \text{on} \quad \partial\Omega \ . \tag{2}$$

We concentrate on the case of an equidistant regular grid. As usual, we use h to denote the mesh width in each dimension. Hence, $n_{\text{dim}} := h^{-1}$ represents the number of sub-intervals per dimension. In the 1D case, the grid nodes are located at positions

$$\{x = ih;\ 0 \le i \le n_{\text{dim}}\} \subset [0,1] \ .$$

In the 2D case, they are located at positions

$$\{(x_1, x_2) = (i_1 h, i_2 h);\ 0 \le i_1, i_2 \le n_{\text{dim}}\} \subset [0,1]^2 \ ,$$

and in the 3D case, the node positions are given by

$$\{(x_1, x_2, x_3) = (i_1 h, i_2 h, i_3 h);\ 0 \le i_1, i_2, i_3 \le n_{\text{dim}}\} \subset [0,1]^3 \ .$$

Consequently, the grid contains $n_{\text{dim}} + 1$ *nodes* per dimension. Since the outermost grid points represent Dirichlet boundary nodes, the corresponding solution values are fixed. Hence, since u is a scalar function, our grid actually comprises $n_{\text{dim}} - 1$ *unknowns* per dimension.

Our presentation is general enough to cover the cases of finite differences as well as finite element discretisations involving equally sized line elements in 1D, square elements in 2D, and cubic elements in 3D, respectively. We focus on the case of *standard coarsening* only. This means that the mesh width H of any coarse grid is obtained as $H = 2h$, where h denotes the mesh width of the respective next finer grid. See [57] for an overview of alternative coarsening strategies such as red-black coarsening and semi-coarsening, for example.

The development of multigrid algorithms is motivated by two fundamental and independent observations which we will describe in the following; the equivalence of the original equation and the residual equation as well as the convergence properties of basic iterative solvers.

2.3 The Residual Equation

A suitable discretisation of the continuous problem given by (1), (2) yields the linear system

$$A_h u_h = f_h \ , \tag{3}$$

where A_h denotes a sparse nonsingular matrix that represents the discrete operator. For the model case of a second-order finite difference discretisation of the negative Laplacian in 2D, A_h is characterised by the five-point stencil

$$\frac{1}{h^2} \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix} \ .$$

We refer to [57] for details including a description of this common stencil notation.

The exact solution of (3) is explicitly denoted as u_h^*, while u_h stands for an approximation to u_h^*. If necessary, we add superscripts to specify the iteration index; e.g., $u_h^{(k)}$ is used to denote the k-th iterate, $k \geq 0$. In the following, we further need to distinguish between approximations on grid levels with different mesh widths. Therefore, we use the indices h and H to indicate that the corresponding quantities belong to the grids of sizes h and H, respectively. The solutions of the linear systems under consideration represent function values located at discrete grid nodes. As a consequence, the terms *grid function* and *vector* are used interchangeably hereafter.

As usual, the *residual* r_h corresponding to the approximation u_h is defined as

$$r_h := f_h - A_h u_h \ . \tag{4}$$

The *(algebraic) error* e_h corresponding to the current approximation u_h is given by

$$e_h := u_h^* - u_h \ . \tag{5}$$

From these definitions, we obtain

$$A_h e_h = A_h \left(u_h^* - u_h\right) = A_h u_h^* - A_h u_h = f_h - A_h u_h = r_h \ ,$$

which relates the current error e_h to the current residual r_h. Hence, the *residual (defect) equation* reads as

$$A_h e_h = r_h \ . \tag{6}$$

Note that (6) is equivalent to (3), and the numerical solution of both linear systems is equally expensive since they involve the same system matrix A_h. The actual motivation for these algebraic transformations is not yet obvious and will be provided subsequently. In the following, we will briefly review the convergence properties of elementary iterative schemes and then illustrate the multigrid principle.

2.4 Convergence Behaviour of Elementary Iterative Methods

There is a downside to all elementary iterative solvers such as Jacobi's method, the method of Gauß-Seidel, and SOR. Generally speaking, when applied to large sparse linear systems arising in the context of numerical PDE solution, they cannot efficiently reduce *slowly oscillating (low-frequency, smooth)* discrete Fourier components of the algebraic error. However, they often succeed in efficiently eliminating *highly oscillating (high-frequency, rough)* error components [57].

This behaviour can be investigated analytically in detail as long as certain model problems (e.g., involving standard discretisations of the Laplacian) as well as the classical iterative schemes are used. This analysis is based on a decomposition of the initial error $e_h^{(0)}$. This vector is written as a linear combination of the eigenvectors of the corresponding iteration matrix. In the case of the model problems under consideration, these eigenvectors correspond to the discrete Fourier modes.

As long as standard model problems are considered, it can be shown that the spectral radius $\rho(M)$ of the corresponding iteration matrix M behaves like $1 - \mathcal{O}(h^2)$ for the method of Gauß-Seidel, Jacobi's method, and weighted Jacobi. Similarly, $\rho(M)$ behaves like $1 - \mathcal{O}(h)$ for SOR with optimal relaxation parameter [55]. This observation indicates that, due to their slow convergence rates, these methods are hardly applicable to large problems involving small mesh widths h.

A closer look reveals that the smooth error modes, which cannot be eliminated efficiently, correspond to those eigenvectors of M which belong to the relatively large eigenvalues; i.e, to the eigenvalues close to 1[4]. This fact explains the slow reduction of low-frequency error modes. In contrast, the highly oscillating error components often correspond to those eigenvectors of M which belong to relatively small eigenvalues; i.e., to the eigenvalues close to 0. As we have mentioned previously, these high-frequency error components can thus often be reduced quickly and, after a few iterations only, the smooth components dominate the remaining error.

Note that whether an iterative scheme has this so-called *smoothing property* depends on the problem to be solved. For example, Jacobi's method cannot be used in order to eliminate high-frequency error modes quickly, if the discrete problem is based on a standard finite difference discretisation of the Laplacian, see [57]. In this case, high-frequency error modes can only be eliminated efficiently, if a suitable relaxation parameter is introduced; i.e., if the weighted Jacobi scheme is employed instead.

[4]Note that non-convergent iterative schemes involving iteration matrices M with $\rho(M) \geq 1$ may be used as smoothers as well.

2.5 Aspects of Multigrid Methods

Coarse Grid Representation of the Residual Equation

As was mentioned in Sect. 2.4, many basic iterative schemes possess the smoothing property; within a few iterations only, the highly oscillating error components can often be eliminated and the smooth error modes remain. As a consequence, a coarser grid (i.e., a grid with fewer grid nodes) may still be sufficiently fine to represent this smooth error accurately enough. Note that, in general, it is the algebraic error (and not the approximation to the solution of the original linear system itself) that is smooth after a few steps of an appropriate basic iterative scheme have been performed.

The observation that the error is smooth after a few iterations motivates the idea to apply a few iterations of a suitable elementary iterative method on the respective fine grid. This step is called *pre-smoothing*. Then, an approximation to the remaining smooth error can be computed efficiently on a coarser grid, using a coarsened representation of (6); i.e., the residual equation. Afterwards, the smooth error must be interpolated back to the fine grid and, according to (5), added to the current fine grid approximation in order to correct the latter.

In the simplest case, the coarse grid is obtained by standard coarsening; i.e., by omitting every other row of fine grid nodes in each dimension, cf. Sect. 2.2. This coarsening strategy results in an equidistant coarse grid with mesh width $H = 2h$. As usual, we use Ω^h and Ω^H to represent the fine grid and the coarse grid, respectively. Furthermore, we assume that n_h and n_H denote the total numbers of unknowns corresponding to the fine grid and the coarse grid, respectively. Note that standard coarsening reduces the number of unknowns by a factor of approximately 2^{-d}, where d is the dimension of the problem.

The coarse representation of (6), which is used to approximate the current algebraic fine grid error e_h, reads as

$$A_H e_H = r_H \ , \tag{7}$$

where $A_H \in \mathbb{R}^{n_H \times n_H}$ stands for the coarse grid operator and $e_H, r_H \in \mathbb{R}^{n_H}$ are suitable coarse grid representations of the algebraic fine grid error e_h and the corresponding fine grid residual r_h, respectively. Equation (7) must be solved for e_H.

Inter-Grid Transfer Operators

The combination of the fine grid solution process and the coarse grid solution process requires the definition of *inter-grid transfer operators*, which are necessary to map grid functions from the fine grid Ω^h to the coarse grid Ω^H, and vice versa. In particular, we need an *interpolation (prolongation) operator*

$$I_H^h : \mathbb{R}^{n_H} \rightarrow \mathbb{R}^{n_h} \ ,$$

which maps a coarse grid function to a fine grid function, as well as a *restriction operator*

$$I_h^H : \mathbb{R}^{n_h} \rightarrow \mathbb{R}^{n_H} \ ,$$

which maps a fine grid function to a coarse grid function. Note that, in the following, I_H^h and I_h^H are also used to denote the corresponding matrix representations of the interpolation and the restriction operators, respectively.

The restriction operator I_h^H is used to transfer the fine grid residual r_h to the coarse grid, yielding the right-hand side of the coarse grid representation (7) of the fine grid residual equation:

$$r_H := I_h^H r_h \ .$$

In the case of discrete operators A_h and A_H with slowly varying coefficients, typical choices for the restriction operator are *full weighting* or *half weighting*. These restriction operators compute weighted averages of the components of the fine grid function to be restricted. They do not vary from grid point to grid point. In 2D, for example, they are given as follows:

- Full weighting:
$$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}_h^H$$
- Half weighting:
$$\frac{1}{8} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix}_h^H$$

Here, we have used the common stencil notation for restriction operators. The entries of these stencils specify the weights for the values of the respective grid function, when transferred from the fine grid Ω^h to the corresponding coarser grid Ω^H. This means that the function value at any (interior) coarse grid node is computed as the weighted average of the function values at the respective neighbouring fine grid nodes, see [10] for example. Representations of the full weighting and the half weighting operators in 3D are provided in [57].

After the coarse representation e_H of the algebraic error has been determined, the interpolation operator is employed to transfer e_H back to the fine grid Ω^h:

$$\tilde{e}_h := I_H^h e_H \ .$$

We use $\tilde{e}_h$ to denote the resulting fine grid vector, which is an approximation to the actual fine grid error e_h. Ideally, the smoother would yield a fine grid error e_h which lies in the range of I_H^h such that it could be eliminated completely by the correction $\tilde{e}_h$.

A typical choice for the prolongation operator is *linear interpolation*. In 2D, for example, this constant operator is given as follows:

$$\frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}_{H}^{h}$$

Here, we have employed the common stencil notation for interpolation operators. The entries of the interpolation stencils specify the weights for the values of the respective grid function, when prolongated from the coarse grid Ω^H to the corresponding finer grid Ω^h. This means that the function value at any (interior) coarse grid node is propagated to the respective neighbouring fine grid nodes using these weights [10]. A representation of the linear interpolation operator in 3D is again provided in [57].

If the stencil coefficients vary significantly from grid node to grid node, it may be necessary to employ operator-dependent inter-grid transfer operators, which do not just compute weighted averages when mapping fine grid functions to the coarse grid, and vice versa [1]. See also [8] for a discussion of how to select appropriate restriction and prolongation operators depending on the order of the differential operator L in (1).

Ideally, the high-frequency components dominate the fine grid error after the coarse grid representation e_H of the error has been interpolated to the fine grid Ω^h and added to the current fine grid approximation u_h; i.e., after the *correction*

$$u_h \leftarrow u_h + \tilde{e}_h$$

has been carried out. In addition, the interpolation of the coarse grid approximation e_H to the fine grid Ω^h usually even amplifies oscillatory error components. Therefore, a few further iterations of the smoother are typically applied to the fine grid solution u_h. This final step is called *post-smoothing*.

Note that the discrete operator and the stencils that represent the inter-grid transfer operators are often characterised by *compact stencils*. In 2D, this means that each of these stencils only covers the current node and its eight immediate neighbours in the grid. In a regular 3D grid, each interior node has 26 neighbours instead. This property of compactness simplifies the parallelisation of the multigrid components. We will return to this issue in Sect. 3 in the context of grid partitioning.

Coarse Grid Operators

The coarse grid operator A_H can be obtained by discretising the continuous differential operator L from (1) on the coarse grid Ω^H anew. Alternatively, A_H can be computed as the so-called *Galerkin product*

$$A_H := I_h^H A_h I_H^h \quad . \tag{8}$$

An immediate observation of this choice is the following. If the fine grid operator A_h as well as the inter-grid transfer operators I_h^H and I_H^h are characterised by compact stencils (i.e., 3-point stencils in 1D, 9-point stencils in 2D, or 27-point stencils in 3D) the resulting coarse grid operator A_H will be given by corresponding compact stencils as well. In a multigrid implementation, this property enables the use of simple data structures and identical parallelisation strategies on all levels of the grid hierarchy, see Sect. 3.

Note that, if the restriction operator corresponds to the transpose of the interpolation operator (up to a constant factor), and if (8) holds, a symmetric fine grid operator A_h yields a symmetric coarse grid operator A_H. If A_h is even symmetric positive definite and the interpolation operator I_H^h has full rank, the corresponding coarse grid operator A_H will again be symmetric positive definite.

As a consequence, if the matrix corresponding to the finest grid of the hierarchy is symmetric positive definite (and therefore nonsingular) and, furthermore, both the inter-grid transfer operators and the generation of the coarse grid matrices are chosen appropriately, each of the coarse grid matrices will be symmetric positive definite as well. This property of the matrix hierarchy often simplifies the analysis of multigrid methods.

Formulation of the Multigrid V-Cycle Correction Scheme

The previous considerations first lead to the two-grid *coarse grid correction (CGC)* V-cycle. This scheme assumes that, in each iteration, the coarse grid equation is solved exactly.

If, however, this linear system is still too large to be solved efficiently by using either a direct method or an elementary iterative method, the idea of applying a coarse grid correction (i.e., the idea of solving the corresponding residual equation on a coarser grid) can be applied recursively. This then leads to the class of multigrid schemes. See [10, 57], for example.

Algorithm 1 shows the structure of a single multigrid V(ν_1,ν_2)-cycle. The notation $S_h^\nu(\cdot)$ is introduced to indicate that ν iterations of an appropriate smoothing method are applied to the corresponding approximation on Ω^h. The parameters ν_1 and ν_2 denote the numbers of iterations of the smoother before and after the coarse grid correction, respectively. Typical values for ν_1 and ν_2 are 1, 2, or 3.

Due to the recursive formulation of Algorithm 1, it is sufficient to distinguish between a fine grid Ω^h and a coarse grid Ω^H. When the recursive scheme calls itself in Step 7, the current coarse grid Ω^H becomes the fine grid of the next deeper invocation of the multigrid V-cycle procedure. Typically, $u_H := 0$ is used as initial guess on Ω^H. We assume that the initial guess and the right-hand side on the finest grid level as well as the matrices on all grid levels of the hierarchy have been initialised beforehand.

An additional parameter γ may be introduced in order to increase the number of multigrid cycles to be executed on the coarse grid Ω^H in Step 7 of

Algorithm 1 Recursive definition of the multigrid CGC V(ν_1,ν_2)-cycle.

1: Perform ν_1 iterations of the smoother on Ω^h (pre-smoothing):

$$u_h^{(k+\frac{1}{3})} \leftarrow S_h^{\nu_1}\left(u_h^{(k)}\right)$$

2: Compute the residual on Ω^h:

$$r_h \leftarrow f_h - A_h u_h^{(k+\frac{1}{3})}$$

3: Restrict the residual from Ω^h to Ω^H and initialise the coarse grid approximation:

$$f_H \leftarrow I_h^H r_h \ , \ u_H \leftarrow 0$$

4: **if** Ω^H is the coarsest grid of the hierarchy **then**
5: Solve the coarse grid equation $A_H u_H = f_H$ on Ω^H exactly
6: **else**
7: Solve the coarse grid equation $A_H u_H = f_H$ on Ω^H approximately by (recursively) performing a multigrid V(ν_1,ν_2)-cycle starting on Ω^H
8: **end if**
9: Interpolate the coarse grid approximation (i.e., the error) from Ω^H to Ω^h:

$$\tilde{e}_h \leftarrow I_H^h u_H$$

10: Correct the fine grid approximation on Ω^h:

$$u_h^{(k+\frac{2}{3})} \leftarrow u_h^{(k+\frac{1}{3})} + \tilde{e}_h$$

11: Perform ν_2 iterations of the smoother on Ω^h (post-smoothing):

$$u_h^{(k+1)} \leftarrow S_h^{\nu_2}\left(u_h^{(k+\frac{2}{3})}\right)$$

Algorithm 1. This parameter γ is called the *cycle index*. The choice $\gamma := 1$ (as is implicitly the case in Algorithm 1) leads to multigrid V(ν_1,ν_2)-cycles, while different cycling strategies are possible. Another common choice is $\gamma := 2$, which leads to the multigrid W-cycle [57], see also Sect. 3.4. The names of these schemes are motivated by the order in which the various grid levels are visited during the multigrid iterations.

Figure 1 shows the algorithmic structure of a three-grid CGC V(ν_1,ν_2)-cycle. This figure illustrates the origin of the term V-cycle. We have used the level indices h, $2h$, and $4h$ in order to indicate that we generally assume the case of standard coarsening, see above.

Remarks on Multigrid Convergence Analysis

A common and powerful approach towards the quantitative convergence analysis (and the development) of multigrid methods is based on *local Fourier*

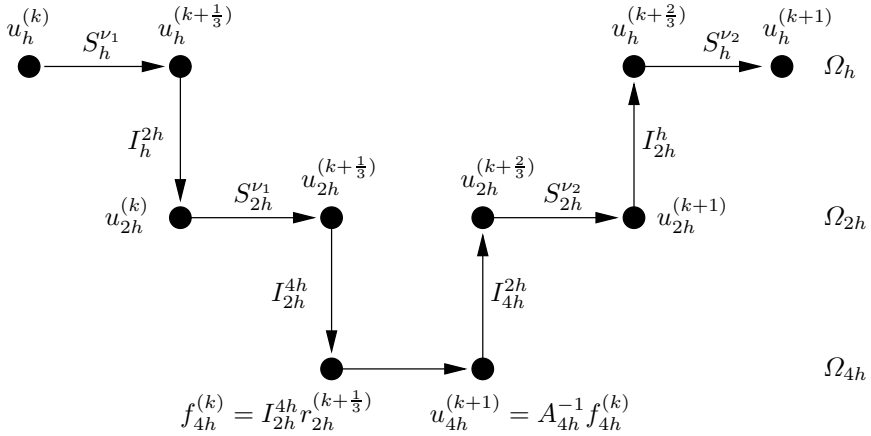


Fig. 1. Three-grid CGC V(ν_1,ν_2)-cycle.

analysis (LFA). The principle of the LFA is to examine the impact of the discrete operators, which are involved in the two-grid or in the multigrid setting, by representing them in the basis of the corresponding Fourier spaces. The LFA ignores boundary conditions and, instead, assumes infinite grids [57, 59].

Alternatively, the convergence analysis of the two-grid scheme and the multigrid scheme can be based on the notions of *smoothing property* and *approximation property*, which have been introduced by Hackbusch [31]. As these names suggest, the smoothing property states that the smoother eliminates high-frequency error components without introducing smooth ones. In contrast, the approximation property states that the CGC performs efficiently; i.e., that the inverse of the coarse grid operator represents a reasonable approximation to the inverse of the fine grid operator. In comparison with the aforementioned LFA, the current approach only yields qualitative results.

The convergence analysis of multigrid methods reveals for example that, for W-cycle schemes applied to certain model problems, these algorithms behave asymptotically optimal. It can be shown that, for these cases, multigrid W-cycles only require $\mathcal{O}(N \log \epsilon)$ operations, where N denotes the number of unknowns corresponding to the finest grid level and ϵ stands for the required factor by which the norm of the algebraic error shall be improved, see [57].

Full Approximation Scheme

So far, we have considered the CGC scheme, or simply the *correction scheme (CS)*. This means that, in each multigrid iteration, any coarse grid is employed to compute an approximation to the error on the next finer grid.

Alternatively, the coarse grid can be used to compute an approximation to the fine grid solution instead. This approach leads to the *full approxima-*

tion scheme/storage (FAS) method. The FAS method is primarily used, if the discrete operator is nonlinear or if adaptive grid refinement is introduced. In the latter case, the finer grids may not cover the entire domain in order to reduce both memory consumption and computational work.

It can be shown that, for the nonlinear case, the computational efficiency of the FAS scheme is asymptotically optimal, as is the case for the aforementioned correction scheme. In addition, the parallelisation of the FAS method resembles the parallelisation of the correction scheme. See [7, 57] for details on the FAS method.

Nested Iteration and Full Multigrid

In most cases, the solution times of iterative methods (i.e., the numbers of iterations required to fulfil the given stopping criteria) can be reduced drastically by choosing suitable initial guesses. When applied recursively, the idea of determining an approximation on a coarse grid first and interpolating this approximation afterwards in order to generate an accurate initial guess on a fine grid leads to the principle of *nested iteration* [32].

The combination of nested iteration and the multigrid schemes we have described so far leads to the class of *full multigrid (FMG) methods*, which typically represent the most efficient multigrid algorithms. For typical problems, the computational work required to solve the discrete problem on the finest grid level up to discretisation accuracy is of order $\mathcal{O}(N)$ only. This results from the observation that, on each level of the grid hierarchy, a constant number of V-cycles is sufficient to solve the corresponding linear system up to discretisation accuracy. See [8, 57] for details.

The FMG scheme generally starts on the coarsest level of the grid hierarchy. There, an approximation to the solution is computed and then interpolated to the next finer grid, yielding a suitable initial guess on this next finer grid. A certain number of multigrid cycles (either CGC-based or FAS-based) is applied to improve the approximation, before it is in turn interpolated to the next finer grid, and so on.

As Algorithm 2 shows, the FMG scheme can be formulated recursively. The linear system on the coarsest level of the grid hierarchy is assumed to be solved exactly. Since the approximations which are mapped from coarse to fine grids in Step 6 are not necessarily smooth and potentially large, it is commonly recommended to choose an interpolation operator $\tilde{I}_H^h$ of sufficiently high order [57].

Depending on the actual problem, the multigrid method applied in Step 7 of Algorithm 2 may either be based on the CGC scheme or on the FAS method. It may involve either V(ν_1,ν_2)-cycles or W(ν_1,ν_2)-cycles. The notation we have introduced in Step 7 is supposed to indicate that ν_0 multigrid cycles are performed on the linear system involving the matrix A_h and the right-hand side f_h, starting with the initial guess $u_h^{(0)}$ which has been determined previously by interpolation in Step 6.

Note that, in order to compute an approximation to the actual solution on each of the coarse grids, it is necessary to appropriately represent the original right-hand side; i.e., the right-hand side on the finest grid level. These coarse grid right-hand sides are needed whenever the corresponding coarse grid level is visited for the first time and, in the case of the FAS method, during the multigrid cycles as well. They can either be determined by successively restricting the right-hand side from the finest grid (as is the case in Algorithm 2, Step 4) or, alternatively, by discretising the continuous right-hand side (i.e., the function f in (1)) on each grid level anew [57].

3 Elementary Parallel Multigrid

In this section, we will introduce the basic concepts used in parallelising the geometric multigrid method.

3.1 Grid Partitioning

We begin with an outline of the parallelisation of a standard (geometric) multigrid method, as introduced in Sect. 2. In the simplest case, computations are performed on a hierarchy of L grids of $m_l \times n_l \, (\times o_l)$ grid lines for 2D (3D) problems and for $1 \leq l \leq L$. A vertex-centred discretisation will associate unknowns with the grid vertices and the grid edges represent data dependencies (though not necessarily all) induced by the discrete equations on level l when applying a stencil operation.

Our parallel machine model is motivated by current cluster architectures. In particular, we assume a distributed memory architecture and parallelisation

Algorithm 2 Recursive formulation of the FMG scheme on Ω^h.

1: **if** Ω^h is the coarsest grid of the hierarchy **then**
2: Solve $A_h u_h = f_h$ on Ω^h exactly
3: **else**
4: Restrict the right-hand side from Ω^h to the next coarser grid Ω^H:

$$f_H \leftarrow I_h^H f_h$$

5: Solve $A_H u_H = f_H$ using FMG on Ω^H recursively
6: Interpolate the coarse grid approximation from Ω^H to Ω^h in order to obtain a good initial guess on Ω^h:

$$u_h^{(0)} \leftarrow \tilde{I}_H^h u_H$$

7: Improve the approximation on Ω^h by applying ν_0 multigrid iterations:

$$u_h \leftarrow \mathrm{MG}_{\nu_1,\nu_2}^{\nu_0} \left(u_h^{(0)}, A_h, f_h \right)$$

8: **end if**

by message passing. The most common message passing standard, today, is the *message passing interface (MPI)*; see [28, 29], for example. Each compute node in this setting may itself be a (shared memory) multiprocessor.

In the following, we will adopt standard message passing terminology and use the notion of a *process* rather than a *processor*. In a typical cluster environment each processor of a compute node will execute one process. The message passing paradigm, however, also allows for situations where several processes are executed by a single processor. In a parallel environment with a number of compute nodes that is significantly smaller than the number of unknowns, the typical approach to parallelise any grid-based algorithm is to split the grid into several parts or *sub-grids* and assign each sub-grid and all of its grid nodes to one process. This approach is for obvious reasons denoted as *grid partitioning* or *domain partitioning*. See also Chap. **??**, for example.

Note that grid (domain) partitioning is sometimes also referred to as *domain decomposition*. This terminology is actually misleading, since domain decomposition denotes a special solution approach, where each process will solve a problem that is completely independent of the problems in the remaining sub-domains. The solution of the original problem posed on the global domain is then obtained by iteratively adapting the local problem on any sub-domain based on the solutions computed by other processes on the neighbouring sub-domains. The solution of the local problems does not induce a need for communication between the processes. This is restricted to the outer iteration in the adaptation phase. If the coupling is done in an appropriate way, the local solutions will converge to the global solution restricted to the local sub-domain. Thus, domain decomposition actually is a concept to design a new parallel algorithm using existing sequential ones as building blocks. For further details, we refer to Chap. **??**.

Grid (domain) partitioning, in contrast, denotes a strategy to parallelise an existing sequential grid-based algorithm. Here we will consider the parallelisation of the (geometric) multigrid algorithm. In this case, there exist no independent local problems on the individual sub-domains and the execution of the algorithm itself induces the need to communicate data between the different processes.

Before we consider this point of interdependence further, we briefly introduce some notation. Let us denote by Ω^l the grid on level l, by Ω^l_k its k-th sub-grid, by n^l_j, $1 \leq j \leq N^l$ a node of Ω^l, and by p^l_k the process responsible for sub-grid Ω^l_k.

As was illustrated in Sect. 2, all operations in a structured geometric multigrid method can in principle be expressed with the help of stencil operators that are applied to a grid function. Such stencils often have a compact support. Hence, when they are locally applied at a certain node $n^l_j \in \Omega^l_k$, this only involves values of the grid function at nodes that are neighbours of the node n^l_j. If the node n^l_j in question is located inside the sub-grid Ω^l_k, then the application of a stencil can be performed by process p^l_k independently. However,

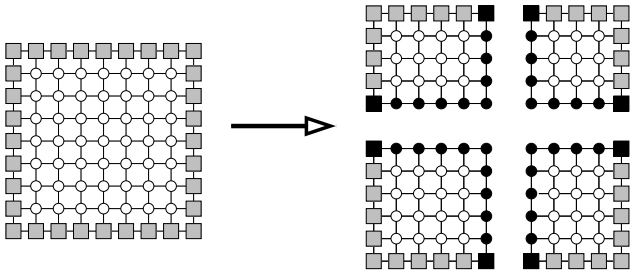


Fig. 2. Partitioning of a structured grid into four sub-domains with an overlap region of width one. We distinguish: inner nodes (*white circles*), boundary nodes (*grey squares*) and ghost nodes (*black circles / squares*)

if n_j^l is lying in the vicinity of the sub-domain boundary, then the application of a stencil may involve nodes that belong to neighbouring sub-domains and are thus not stored by process p_k^l.

The straightforward solution to this problem is to let p_k^l query its neighbours for the node values involved whenever it requires them. This, however, must be ruled out for efficiency reasons. Instead, the following approach is typically employed.

Each sub-domain Ω_k^l is augmented by a layer of so-called *ghost nodes* that surround it. This layer is denoted as *overlap region* but other names such as *halo* are also often used. The ghost nodes in the overlap region correspond to nodes in neighbouring sub-domains that p_k^l must access in order to perform computations on Ω_k^l. The width of the overlap region is therefore determined by the extent of the stencil operators involved. Figures 2 and 3 give two examples; one for a structured and one for an unstructured grid. The ghost nodes can be considered as a kind of read-only nodes. This means that process p_k^l will never change the respective function values directly by performing computations on them, but will only access their values when performing computations on its own nodes.

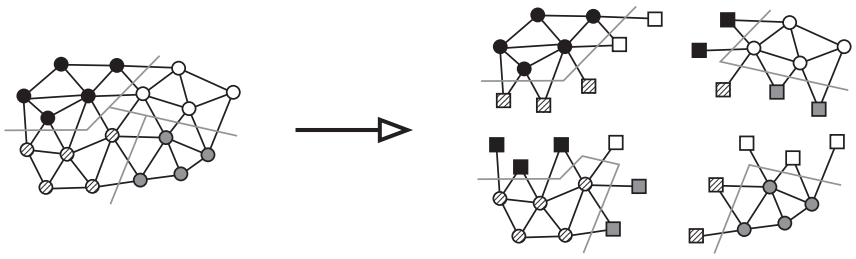


Fig. 3. Partitioning of a unstructured grid into four sub-domains with an overlap region of width one. Ghost nodes are coloured according to their corresponding master nodes

Keeping the values in the overlap region up-to-date requires communication with the neighbouring processes. As an example, Fig. 4 shows a strategy for updating the overlap regions in a partitioning with four sub-grids and also demonstrates how "diagonal" communication (in this example communication between the lower-left and top-right process as well as between the top-left and bottom-right process) can implicitly be avoided. The precise details of such an update and its frequency depend on the multigrid component involved and will be considered in the following sections.

When it can be assured that the function value at a ghost node g_j^l in the overlap region of Ω_k^l is always identical to the function value at its master node n_j^l when it is read by p_k^l, the computation will yield the same result as in the sequential case. One of the major challenges in designing parallel geometric multigrid methods is to employ appropriate multigrid components and to develop suitable implementations, such that this requirement is met with as little communication costs as possible, while on the other side retaining the fast convergence speed and high efficiency achieved by sequential multigrid algorithms.

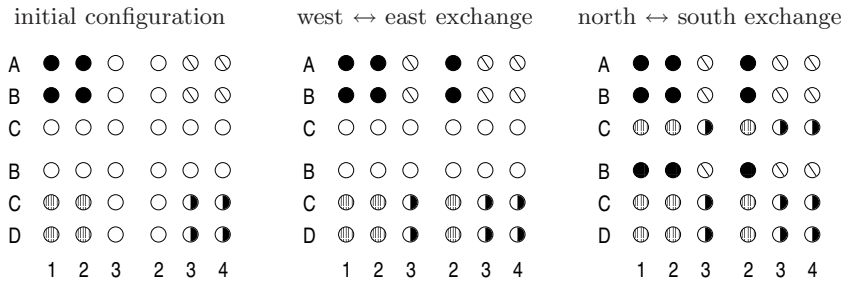


Fig. 4. Example update of ghost node values at the intersection of four sub-domains

Another important question is how to choose the grid partitioning. Since the size of the surface of a sub-grid determines the number of ghost nodes in the overlap region, it is directly related to the amount of data that must be transferred and thus to the time spent with communication. This *communication time* is a consequence of the parallel processing. This is complemented by the *computation time*, which comprises all operations that a sequential program would have to carry out for the local data set. In our application, this time is dominated by floating-point operations, hence the name.

In order to achieve a high computation-to-communication ratio resulting in reasonable parallel efficiency, one thus typically strives for a large volume-to-surface ratio when devising the grid partitioning. With structured grid applications, this initially led to the preference of 2D over 1D partitioning for 2D problems and 3D over 2D or 1D partitioning for 3D simulations.

In light of the increasing gap between CPU speed and main memory performance (cf. Sect. 5) the focus has started to change. Not only the amount of data to be communicated is now taken into account, but also the run-time costs for collecting and rearranging them. Here, lower dimensional splittings such as 1D partitionings in 3D, for example, can yield performance benefits. See [51], for example. They may further be advantageous in the case of the application of special smoothers such as line smoothers, cf. Sect. 6.1.

3.2 Parallel Smoothers

In this section, we will describe the parallelisation of the classical point-based smoothers that are typically used in geometric multigrid methods. More sophisticated smoothers are considered in Sect. 6.1. Since, in the context of multigrid methods, smoothers always operate on the individual grid levels, we drop the $.^l$ super-script in this section.

Parallelisation with Read-Only Ghost Nodes

We start our exposition with the simplest smoother, the weighted Jacobi method given by

$$u^{\text{new}}(n_k) = \frac{1}{\sigma(0,n_k)} \left[f(n_k) - \sum_{j \in \mathcal{K}(n_k) \setminus \{0\}} \sigma(j,n_k)\, u^{\text{old}}(n_{k+j}) \right] , \quad (9)$$

where $\mathcal{K}(n_k)$ denotes the support of the stencil representing the discrete operator at node n_k, $\sigma(j,n_k)$ is the stencil weight for the value at node n_{k+j}, and u and f denote the approximate solution and the discrete right-hand side, respectively. Equation (9) implies that the new iterate u^{new} at each node is computed based solely on values of the previous iterate u^{old}. Thus, if the values of u^{old} are up-to-date at the ghost nodes of the overlap region, each process can perform one Jacobi sweep on its sub-domain independently of the remaining processes. Once this is done, one communication phase is required to update the overlap regions again in order to prepare for the next sweep or the next phase of the multigrid algorithm.

Let us now turn to Gauß-Seidel smoothing and its weighted variant; the successive over-relaxation (SOR). The formula for this smoother is given by

$$u^{\text{new}}(n_k) = u^{\text{old}}(n_k) - \frac{\omega}{\sigma(0,n_k)} \left[f(n_k) - \sum_{j \in \mathcal{K}(n_k)} \sigma(j,n_k)\, v(n_{k+j}) \right] , \quad (10)$$

where the Gauß-Seidel method is obtained for $\omega = 1$. In (10), we have $v(n_{k+j}) = u^{\text{old}}(n_{k+j})$, if the node n_{k+j} has not yet been updated during the computation, and $v(n_{k+j}) = u^{\text{new}}(n_{k+j})$, otherwise. Thus, (10) shows

that in the case of SOR smoothing the ordering in which the nodes are updated may influence the properties and the qualities of the smoother. This turns out to be true and details can be found in [57], for instance.

In a parallel setting this inherently sequential dependency causes some difficulty. For example, assume a splitting of a regular grid into four subdomains as shown in Fig. 2. Choosing a lexicographic ordering of the nodes for the update and starting the Gauß-Seidel sweep from the lower left would imply that only the process responsible for the lower left sub-domain could perform any computations until the update reaches the top left sub-domain, and so on. As a consequence, three processes would be idle all the time. A different ordering is therefore required in the parallel setting.

Fortunately, the ordering chosen for most sequential multigrid applications is not a lexicographic one anyway. Instead, a so-called red-black or checkerboard ordering is frequently used. This ordering often has superior properties in the multigrid context. It is based on a splitting of the grid nodes into two groups depending on their positions; a red group and a black one. For the typical 5-point stencil representing the discrete Laplacian, see Sect. 2.3, the nodes in each of the groups are completely independent of each other. A red node only depends on black nodes for the update, and vice versa. The advantage of this splitting for parallelisation is that each process can update the red nodes in its sub-domain using the values of its black ghost nodes without the need to communicate with its neighbours. Once this is done, one communication phase is required in order to update the values at the red ghost nodes. Afterwards, the update of the black nodes can proceed, again independently, followed by a second communication phase for updating the black ghost nodes.

Compared to the Jacobi smoother, the SOR smoother thus requires one additional communication step. The amount of data that need to be transmitted remains the same, however. For larger discretisation stencils, more colours are needed; a compact 9-point-stencil requires a splitting into four sets of nodes, for example.

Trading Computation for Communication

Another possibility for parallelising the red-black SOR smoother and smoothers with similar dependency patterns is to discard the read-only property of the ghost nodes and to trade computation for communication. Assume that we perform a V(2,0)-cycle, for example. In the first approach, four communication steps are required for the two pre-smoothing steps. Let us denote by the k-th generation of ghost nodes all nodes that are required to update in the SOR sweep the nodes in the (k-1)-th generation, with the nodes in a sub-domain belonging to generation 0.

If we extend the overlap region to include all ghost nodes up to fourth generation, we can perform the four partial SOR sweeps without communication. This is possible, since in the overlap region the same values are computed as

in the neighbouring sub-domains. We must take into account, however, that each partial sweep invalidates one generation of ghost nodes; they cannot be updated anymore because their values would start to differ from those at the master nodes that they should mirror.

This second approach increases both the overlap region (and thus the amount of transferred data) as well as the computational work. Therefore, its advantage strongly depends on the parallel architecture employed for the simulation. In the first place, it is the ratio of the costs of floating-point computation, assembly of data into a message, transfer of a message depending on its size, and initialising a communication step between processes that are important here. Moreover, the size of the individual sub-domains plays an essential role as well. We will return to the latter aspect in Sect. 3.4.

Hybrid Smoothers

Let us conclude this subsection by mentioning another general concept for parallelising multigrid smoothers, which is known under the notion of *hybrid smoothing* [41]. See also Chap. **??** in this respect. The underlying idea is simple. One does not try to parallelise the chosen smoother, maintaining all data dependencies. Instead, this smoother is applied on each sub-domain independently of the other ones, and the values at the ghost nodes are only updated after each sweep. Hence, this scheme corresponds to an inexact block-Jacobi method. The impact of this approach on the smoothing property and the convergence speed of the multigrid method is, of course, strongly problem-dependent. It can be quite negative, though, and one must be cautious with this approach. However, there exist first approaches to improve this concept by the use of suitable weighting parameters; see Chap. **??** for more details.

3.3 Parallel Transfer Operators

As was introduced in Sect. 2, the use of grids of different resolutions in geometric multigrid necessitates the use of inter-grid transfer operators. These allow to restrict a grid function from the function space associated with a fine grid Ω^l to the space associated with a coarser grid Ω^{l-1} and to prolongate a grid function from Ω^{l-1} to Ω^l.

Both restriction and prolongation operators can typically be expressed by compact local stencils and, as with the Jacobi smoother of the previous section, the order in which the nodes are treated does not influence the final result. Using an overlap region of sufficient width, the multigrid transfer operators can be parallelised easily. Note, however, that using a vertex-centred discretisation, the grids Ω^{l-1} and Ω^l cover different areas. An example is given in Fig. 5, where a 1D prolongation by (piece-wise) linear interpolation is sketched. The overlap regions of the two sub-domains are shaded in grey. Thus, some care must be taken when performing prolongation and restriction operations at the interfaces of the sub-domains.

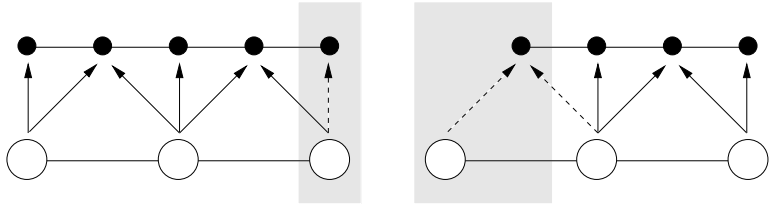


Fig. 5. Prolongation at the interface between two sub-domains in 1D

3.4 Parallel Multigrid Cycles

Multigrid methods may employ different cycling strategies, the most prominent being the V- and the W-cycle. In principle, a parallel multigrid method can also be used with different cycling strategies. However, again some extra issues come into play in the parallel setting. One of the questions in this respect is that of parallel efficiency. As was already mentioned in Sect. 3.1, the latter is coupled to the ratio of computation time to communication time. In order to illustrate how this ratio changes from one grid level to another, we have to determine the dominating factors for the two parts.

In our numerically intensive case, it is safe to assume that the computation time is linearly proportional to the number of nodes in a sub-domain Ω_k^l. For the communication part, we ignore for the moment the impact of message latencies and assume that the communication time is linearly proportional to the number of ghost nodes in the overlap region of Ω_k^l. The latter is, in practice, in good approximation a multiple of the number of nodes on the sub-domain boundary. Thus, the ratio of communication time to computation time is directly proportional to the surface-to-volume ratio of the sub-domain.

As an example, Table 1 presents the values of the surface-to-volume ratio for typical square sub-domains in 2D and cubic sub-domains in 3D. We assume a standard coarsening by doubling the mesh width in each dimension, see Fig. 6, and an overlap region of width 1. The example clearly shows that the surface-to-volume ratio on coarser grids is less favourable than on finer grids. Similar results hold for other partition geometries.

Depending on the representation of the discrete operator, partition sizes of a million or more unknowns for a scalar PDE are not uncommon. The surface-to-volume ratio for a single grid algorithm operating on a partition with two million unknowns is in the order of 5% or, in other words, there are twenty times more compute nodes than ghost nodes. However, on clusters with high performance processors but slow networks, even this factor of 20 may not be sufficient to hide the time for data exchange behind the time for the floating-point operations by using the technique of overlapping communication and computation discussed in Sect. 6.3.

In the case of multigrid algorithms, the network performance becomes even more important. One aim of the sequential multigrid algorithm is to perform as little work as possible on the finest grid and to do as much work as possible

Table 1. Surface-to-volume ratio in two and three spatial dimensions. l denotes the grid level in the hierarchy, $B(l)$ the number of ghost nodes for a sub-domain Ω_k^l, $V(l)$ the number of nodes in Ω_k^l and r is the quotient of $B(l)$ divided by $V(l)$

	2D			3D		
l	$B(l)$	$V(l)$	r	$B(l)$	$V(l)$	r
1	8	1	8.00	26	1	26.0
2	16	9	1.77	98	27	3.63
3	32	49	0.65	386	343	1.13
4	64	225	0.28	1,538	3,375	0.46
5	128	961	0.13	6,146	29,791	0.21
6	256	3,969	0.06	24,578	250,047	0.10
7	512	16,129	0.03	98,306	2,048,383	0.05
8	1024	65,025	0.02	393,218	16,581,375	0.02
9	2048	261,121	0.01	1,572,866	133,432,831	0.01
10	4096	1,046,529	0.004	6,291,458	1,070,599,167	0.006

on the coarser levels. The assumption behind this idea is that operations such as smoothing steps are much cheaper in terms of computing time on coarser grids than on finer ones. However, this is not necessarily valid in the parallel setting. Given that network data transfer is still significantly slower than accesses to main memory, it is safe to assume that, for instance, smoothing the 27 interior points on level 2 in a cube takes less time than communicating the 26 ghost values for the one unknown on grid level 1. Thus, in this case, it is less time-consuming to keep working on a finer level than to delegate the work to a coarser level, provided the work on the finer level is similarly efficient to solve the problem.

Depending on the processor and network specifications, the same reasoning may actually hold true for other levels as well, so that it may well be faster to stop the multigrid recursion at level three or level four, say, or to use fewer processes for the computations on the coarser grid levels. The problem then

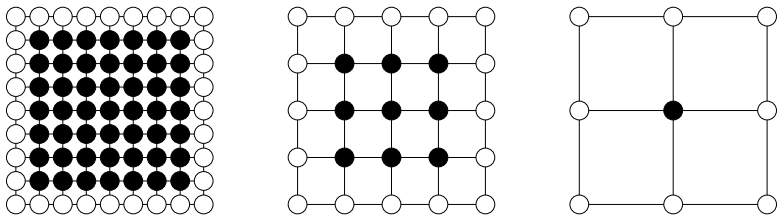


Fig. 6. Example of full coarsening in 2D. Hollow circles indicate ghost nodes, full circles denote the unknowns in the local partition

is to find an appropriate solver for the remaining coarsest level in terms of computation and communication.

Let us further examine the communication demands of the grid hierarchy in a parallel multigrid method from another perspective. Consider a 3D scalar elliptic PDE on a grid with $N = n^3$ unknowns. As can be seen from Sect. 2, the computational cost for a single V- ($\gamma = 1$) or W-cycle ($\gamma = 2$) in this case is of order $\mathcal{O}(N) = \mathcal{O}(n^3)$. The latter can be obtained from estimating the geometric series

$$n^3 + \gamma^1(n/2)^3 + \gamma^2(n/4)^3 + \gamma^3(n/8)^3 + \dots$$

in combination with a linear relationship of the number of nodes per level to the computational work on that level. Assuming that the domain is partitioned into a grid of $p^3 = P$ sub-domains of (nearly) equal size, each of these has $\mathcal{O}((n/p)^3)$ nodes, and therefore a number of ghost nodes of order $\mathcal{O}((n/p)^2)$. The *total* data volume of communication for a V- or W-cycle is then given by

$$\mathcal{V}_{\text{comm}}^{\text{3D}} = \mathcal{O}\left(p^3 \, \frac{n^2}{p^2} \, (1 + \gamma^1/4 + \gamma^2/16 + \gamma^3/64 + \dots)\right) = \mathcal{O}(p\,n^2) \ .$$

This consideration demonstrates that, even asymptotically (i.e., for $n \to \infty$), the aggregate costs of communication of a 3D multigrid V- or W-cycle differ from that of a single data exchange between the partitions on the finest grid only by a constant factor. In 2D, however, one obtains

$$\mathcal{V}_{\text{comm}}^{\text{2D}} = \mathcal{O}\left(p^2 \, \frac{n}{p} \, (1 + \gamma^1/2 + \gamma^2/4 + \gamma^3/8 + \dots)\right) \ .$$

Thus, asymptotically the argument in 2D only holds for the V-cycle, where we have $\mathcal{V}_{\text{comm}}^{\text{2D}} = \mathcal{O}(p\,n)$. In contrast, the W-cycle leads to $\mathcal{V}_{\text{comm}}^{\text{2D}} = \mathcal{O}(p\,n\log(n))$. In practice, of course, other effects will influence the run-time behaviour making the situation more complicated. The most noticeable additional effect on many current system architectures may be the startup cost *(latency)* of communication. When messages are small, as is the case when data is exchanged between sub-domains on coarse grids, then the time for initiating a message exchange may be so large that it cannot be neglected anymore. In contrast to the volume of communication, the number of messages is independent of the sub-domain size and thus will grow with $\log(n)$ for a standard multigrid method that employs all levels of the hierarchy. Also the commonly applied technique of overlapping communication and computation, see Sect. 6.3, that tries to reduce overall run-time requires that there is enough computational work behind which communication can be hidden. This of course becomes problematic, when sub-domains get too small on coarser grids. Additionally, the aggregate data volume for all processes says little about the time needed for communication. Depending on the network topology and allocation of the processes, network jams may occur, or bandwidth may depend on message sizes, etc.

Finally, it is the run-time of a parallel multigrid algorithm that is of major interest to the common user. The latter may be as much determined by the cost of communication as by the cost of computation, and finding ways to reduce communication cost may be more important than reducing the cost of computation.

As a consequence, the typical rule of thumb in parallel multigrid is to prefer cycling strategies that spend less time on coarser grids (e.g., V-cycles) to those that spend more time on coarser grids (e.g., W-cycles) as far as parallel efficiency is concerned. However, a concrete application problems may still require the use of W-cycles for its increased robustness.

The question of a suitable parallel multigrid cycling strategy is closely related to the question of how deep the grid hierarchy should be chosen. In multigrid, one often tries to coarsen the mesh as far as possible, in the extreme even to a level with only one unknown. It turns out that, in the sequential case, this is in fact a competitive strategy, not alone since the exact solution of the coarsest grid problem in this case can be performed with very small costs. In light of the above discussion, however, the question is whether it is sensible to have grid levels where the number of compute nodes exceeds that of grid nodes. One might even ask whether a grid level leading to sub-domains with a large ratio of ghost nodes to interior nodes is desirable. As often with parallel algorithms, there is no universal answer to these questions, since the best approach depends on the actual parallel environment and the application at hand.

However, two general strategies exist. One is known under the notion of *coarse grid agglomeration*. The idea here is to unite the sub-domains of different processes on one process once the ratio between interior and boundary nodes falls below a certain threshold. While this leaves processes idle and induces a significant communication requirement at those points in the multigrid cycle where one descends to or ascends from a grid level on which coarse grid agglomeration occurs, it can nevertheless be advantageous, since it reduces the communication work for performing operations (smoothing, residual computation, coarse grid correction, etc.) on this level and the lower ones. The extreme of this approach is to collect all sub-domains on one process on a certain level, and to perform a sequential multigrid algorithm on the grid levels below.

Unfortunately, while this technique is mentioned in nearly all publications on parallel multigrid, there appears, at least to our best knowledge, to exist no publication that thoroughly investigates the approach and gives the user some advice on how and when this should be done depending on the underlying parallel architecture and the specific multigrid algorithm.

Another strategy is to use a so-called *U-cycle*. The idea here is to use a comparatively flat multigrid hierarchy. While this induces the need to compute an exact solution for the coarsest grid problem already on a global grid with a larger number of unknowns, doing so can on one hand improve the convergence speed, which can alleviate the additional costs. On the other hand, one can

employ another parallel solution method, such as a parallel sparse direct solver or a parallel preconditioned Krylov subspace method for this purpose. For example, see [61] for a closer examination of this approach.

Generally, parallel multigrid designers should critically question what they are optimising for. Implementing a U-cycle which stops already at a very fine coarsest grid and simply solves the coarsest grid equations by a sufficient number of calls to the smoother may lead to excellent parallel efficiency and impressive aggregate Mflop/s rates[5]. However, in terms of run-time this will seldom be a high performance approach, since the coarse grid solver is of course inefficient and needs many iterations which incurs computation and communication time. This shows that all arguments about parallel efficiency in the multigrid context must be considered with some care.

At this point, it can also be pointed out that similar arguments should be considered when comparing multigrid algorithms with other iterative linear solvers. If, for example, a preconditioner achieves a condition number which depends logarithmically on the system size of a 3D problem (as is the case for some of the more advanced domain decomposition algorithms, for instance), then the CG solver will need $\mathcal{O}(\log n)$ iterations. Assuming that the preconditioner requires one next neighbour data exchange, the solver has an overall communication volume of order $\mathcal{O}(p\,n^2 \log n)$, which is asymptotically worse than for an equivalent multigrid method, while being asymptotically equivalent in the number of messages to be sent.

Summarising the argument in this section, we see that multigrid may make it difficult to obtain good parallel efficiency, but this should not misguide anyone to try to use too simplistic variants or be misinterpreted as necessarily leading to poor run-time performance. Asymptotically, multigrid is not only optimal in (sequential) computational complexity, but using grid partitioning, it also requires asymptotically only close to the minimal amount of communication.

3.5 Experiments

The computational results in this section have the very simple aim to illustrate the potential of parallel geometric multigrid methods and to show that, on suitable architectures, parallel multigrid can be designed to be extremely efficient both with respect to absolute timings and in terms of parallel efficiency. The test case is a Poisson problem with Dirichlet boundary conditions in three space dimensions. Though numerically no challenge, this is a hard test problem with respect to efficiency, since the ratio of computation to communication is very low and thus it is no trivial to achieve good speedup results.

For the scalability experiment, the problem domains consist of $9N$ unit cubes with N being the number of processes in the computation. Seven re-

[5] 1 Mflop/s = 10^6 floating-point operations per second, 1 Gflop/s = 10^9 floating-point operations per second

finement levels are used in the grid hierarchy. In the case of the speed-up experiment, the number of computational cells in the problem domain remains constant, of course. The L-shaped problem domain consists of 128 unit cubes and six refinement levels are generated by repeated subdivision. The problem domain is distributed to 2, 4, 8, 16, 32, and 64 processes. The single processor computing time is estimated by dividing the two processor execution time by two.

The algorithmic components of the multigrid method are well established. We employ a variant of the red-black Gauß-Seidel iteration as smoother and full weighting and trilinear interpolation as the inter-grid transfer operators. The cycling strategy is a full multigrid method in which we perform two V(2,2)-cycles on each grid level before prolongating the current approximation to the next finer grid in the hierarchy. A discretisation by trilinear finite elements results in a 27-point stencil in the interior of the domain.

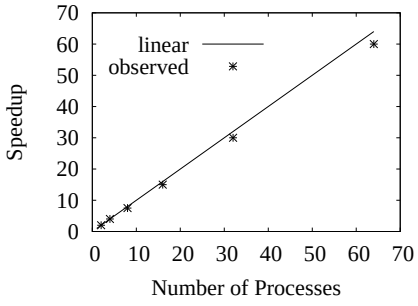
The experiments were carried out on the Hitachi SR8000 supercomputer at the Leibniz Computing Centre in Munich. The machine is made up of SMP nodes with eight processors each and a proprietary high speed network between the nodes. The program runs with an overall performance, including startup, reading the grid file and all communication, of 220 Mflop/s per process, which yields an agglomerated node performance of 1.76 Gflop/s out of the theoretical nodal peak performance of 12 Gflop/s. To put the solution time into perspective, we note that the individual processors have a theoretical peak performance of 1.5 Gflop/s, which is not much compared to the 6 Gflop/s and more of currently available architectures. Nevertheless, a linear algebraic system with more than 10^{10} unknowns distributed over 550 processes and as many processors is solved in less than 50 seconds. For information concerning the design principles and the data structures of this multigrid code, we refer to [35].

The scalability results owe much to the ability of the full multigrid algorithm to arrive at the result with a fixed number of cycles, independent of the problem size, cf. Sect 2.5. One might think that the speedup experiment represents a harder test, as the amount of communication over the network increases, while the amount of computations per process decreases. However, as shown in Fig. 7, the behaviour is close to optimal. In the experiment, an L-shaped domain consisting of 128 cubes is distributed to 2, 4, 8, 16, 32, and 64 processes. Each cube is regularly subdivided six times. The same Poisson problem is solved using the same multigrid algorithm as before.

Even the straightforward V-cycle can handle large problems in acceptable time frames, as the following experiment shows. We solve again a 3D Poisson problem with Dirichlet boundary conditions again on the Hitachi SR8000, but this time we employ the 7-point finite difference stencil on problem domains $\Omega_1 = [0,2] \times [0,2] \times [0,2]$, partitioned into 8 hexahedra, $\Omega_2 = [0,4] \times [0,2] \times [0,2]$, partitioned into 16 hexahedra, $\Omega_3 = [0,4] \times [0,4] \times [0,2]$, partitioned into 32 hexahedra. On each grid level, we perform two pre- and two post-smoothing steps. The fact that the number of V-cycles in Tab. 2 increases

CPU	Dof $\times 10^6$	Time in (s)
64	1179.48	44
128	2359.74	44
256	4719.47	44
512	9438.94	45
550	10139.49	48

(a)



(b)

Fig. 7. Parallel performance results in 3D: (**a**) Scalability experiments using a Poisson problem with Dirichlet boundary conditions on an L-shaped domain. *Dof* denotes the degrees of freedom in the linear system, the timing results given refer to the wall clock time for the solution of the linear system using a full multigrid solver. (**b**) Speedup results for the same Poisson problem

Table 2. Scale up results for parallel V(2,2)-cycle in 3D: *Dof* are the degrees of freedom, *Time* is the wall clock solution time for the linear system

CPU	Dof $\times 10^6$	Time in (s)	V-cycles
8	133.4	78	9
16	267.1	81	10
32	534.7	95	11

with the problem size is a consequence of the stopping criteria that imposes an absolute limit on the l_2-norm of the residual vector. Still, the V-cycle multigrid manages to solve a system with more than $500 \cdot 10^6$ unknowns in 95 seconds, in this case on 32 processors.

4 Parallel Multigrid for Unstructured Grid Applications

In this section, we examine the added complications that arise when parallelising a multigrid solver on unstructured grids. We first consider the single grid case before turning to the multigrid setting.

From a sufficiently abstract point of view, the steps in the parallelisation of operations on an unstructured grid are the same as in the structured case considered so far. First, the grid has to be partitioned, then the necessary ghost points have to be determined. After that, it has to be worked out for each ghost value from which partition the information can be retrieved before finally the communication structures between the partitions are set up.

Fortunately, the commonly used programs to perform the partitioning step, such as *Metis* or *ParMetis* [38, 39], are applicable to structured and unstructured grids alike. As in the structured case, the discretisation scheme determines which ghost values are needed around the partition boundary. One major difference between the two grid types concerns the number of neighbouring partitions with which a given partition has to exchange information. In the structured case, the number of neighbours is bounded independently of the number of partitions, whereas in the unstructured case, this number can become arbitrarily large. For operations such as a matrix vector product or an inner product between two distributed vectors, this of course affects the number of messages that have to be sent in order to keep the data synchronised across partitions, but the basic functionality is the same as in the structured case. However, for inherently sequential operations, such as the common Gauß-Seidel smoother in multigrid schemes, efficient parallelisation strategies that minimise the number of messages are more difficult to construct. The problem is linked to the one of the colouring of the data dependency graph between the partitions. Given that this graph depends on the distribution of the unstructured grid, it is only known at run-time. Hence, any parallelisation strategy has to be formulated and implemented in a general fashion, which is much more complicated than in the structured case, where the neighbourhood relationships are known at compile-time.

Turning now to multigrid methods on unstructured grids, the first problem concerns the construction of the grid hierarchy. The common approach to generating the hierarchy of nested approximation spaces needed for standard geometric multigrid on unstructured grids consists in carrying out several steps of repeated regular refinement. The reason for the popularity of this approach lies in the complexity and difficulty of any alternative. Coarsening a given unstructured grid so that the resulting approximation spaces are nested is often simply impossible. Hence, one would have to work on non-nested spaces. Multigrid methods for such a setting are much more complicated than those for their nested counterparts. In summary, although it is possible to construct geometric multigrid methods on hierarchies of repeatedly coarsened unstructured grids, this option is rarely chosen in practice.

Concerning the parallelisation, the simplest approach is to perform the grid distribution only once, on the unstructured input grid, and then to let the finer levels inherit the communication pattern from the coarsest grid. However, this strategy will amplify any kind of load imbalance present in the distribution of the initial grid. The other extreme approach is to partition each grid level individually. This avoids the load balancing problems of the other method at the cost of a more complex initialisation phase and potentially more complicated communication patterns in the multigrid algorithm. Which approach yields the smaller run-time depends on the concrete problem and the CPU and network specifications. We refer to Chap. **??** for a discussion of load balancing issues.

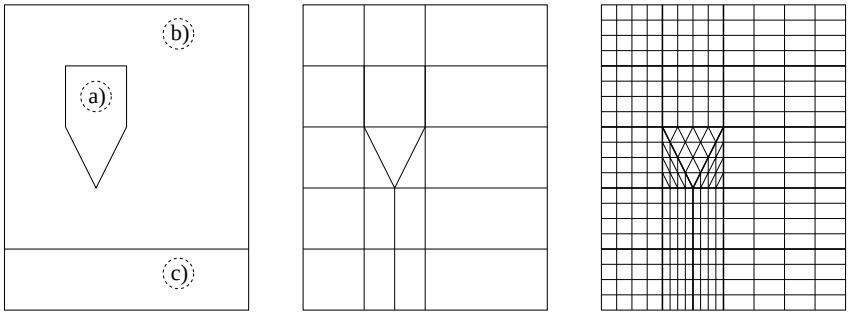


Fig. 8. Non-destructive testing example, from left to right: **I)** the problem domain with a) the coil that generates a magnetic field, b) air surrounding the coil, and c) the material to be tested. The material parameters (magnetic permeability) of the three different components differ from each other, but are assumed to be constant within each component (homogeneous materials). **II)** the coarse, unstructured grid that represents the problem geometry and **III)** the grid after two global regular subdivision steps

4.1 Hierarchical Hybrid Grids

The repeated refinement of an unstructured input grid described above opens up possibilities for performance improvement over standard unstructured implementations. The main observation is that, after a number of refinement steps, the resulting grids are still unstructured globally, but exhibit regular features locally.

We illustrate the generation of these locally regular structures using a simplified problem domain that arose in an electromagnetic field problem. Consider the setting in Fig. 8.

For the representation of the discrete operator on the grid in Fig. 8 II), it is appropriate to employ sparse matrix storage schemes. However, the situation on the grid in Fig. 8 III) is different. Within each cell of the unstructured input grid, the repeated refinement has resulted in regular patches. In the interior of each coarse grid cell, the grid is regular and the material parameter is constant. This implies that one stencil suffices to represent the discrete operator inside such a regular region.

The main idea is now to turn operations on the refined, globally unstructured grid into a collection of operations on block-structured parts where possible and resort to unstructured operations only where necessary. It turns out that the data structures used to capture the locally regular nature of the fine grids are well suited for parallel computations. Provided a sufficiently high level of refinement or, put differently, provided that the regular regions are sufficiently large in comparison to the remaining parts, this approach combines an improved single node floating-point performance with the geometric

flexibility of unstructured grids, at least at the input level. For more details, see [5, 34].

Assuming a vertex-based discretisation, the discrete operator can be expressed by a stencil with constant shape in each regular region of the refined grids as the neighbourhood relationship does not change. In some cases, the entries of the stencil are also constant over the object. Inside such a region, the representation of the operator can be reduced to one single stencil. Both properties, constant stencil shape and constant stencil entries, help to improve the performance of operations involving the discrete operator. The scale of the improvement depends on the computer architecture.

On general unstructured grids, the discrete operator is usually stored in one of the many sparse matrix formats. If one uses such general formats in operations such as the matrix-vector product or a Gauß-Seidel iteration, one usually has to resort to indirect indexing to access the required entries in the vector. Being able to represent the discrete operator in the regular regions by stencils with fixed shapes, we can express the memory access pattern for the operation explicity through index arithmetic and thus enable the compiler to analyse and optimise the memory accesses better. In the sparse matrix case, the memory access pattern is known at run-time, in our setting it is known at compile-time, at least for a significant subset of all points.

On the Hitachi SR8000 supercomputer at the Leibniz Computing Centre in Munich, the change from indirect indexing to index arithmetic improves the Mflop/s performance of a Gauß-Seidel iteration on a single processor from around 50 Mflop/s for a CRS (compressed row storage) implementation to 300 Mflop/s for the stencil-based computation. These values were obtained with a 27-point finite element discretisation inside a refined hexahedron on a processor with a theoretical peak performance of 1500 Mflop/s. On this high memory bandwidth architecture, the explicit knowledge about the structure of the operations results in a six-fold performance improvement. Many cache-based architectures do not offer such a high bandwidth to main memory. Given that a new stencil has to be fetched for each unknown, the memory traffic to access the stencil values slows down the computations on these machines. As an example for such machines, we consider an Intel Pentium 4 processor with 2.4 GHz clock speed, 533 MHz front side bus and dual channel memory access. Our CRS-based Gauß-Seidel iteration runs at 190 Mflop/s on a machine with a theoretical peak performance of 4800 Mflop/s[6]. With the fixed stencil shape implementation, we observe a performance between 470 and 490 Mflop/s, depending on the problem size, which is 2.5 times more than for the standard unstructured one. In all cases, the problem size did not fit into the caches on the machines. In the case of constant stencil entries for an element, the advantage of the structured implementation is even clearer. Instead of fetching 27 stencil values from memory for each unknown, the same stencil is applied

[6]The results on the PC architecture were obtained with the version 3.3.3 of the GNU compiler collection.

to all unknowns in the element. This obviously reduces the amount of memory traffic significantly. On the Hitachi SR8000, such a constant coefficient Gauß-Seidel iteration achieves 884 (out of 1500) Mflop/s on a hexahedron with 199^3 unknowns. Compared to the standard unstructured implementation, this amounts to a speed up factor of 17.5. Again, on the commodity architecture Pentium 4, the improvement is less impressive. For a problem of the same size, the constant coefficient iteration runs at 1045 Mflop/s, which is 5.5 times faster than its unstructured counterpart.

5 Single-Node Performance

5.1 Overview

In order to increase the run-time performance of any parallel numerical application, it is essential to address two related optimisation issues, each of which requires intimate knowledge in both the algorithm and the architecture of the parallel computing platform. Firstly, it is necessary to minimise the parallelisation overhead itself. This optimisation target represents the primary focus of this paper.

Secondly, it is essential to exploit the individual parallel resources as efficiently as possible; i.e., by achieving the highest possible performance on each node in the parallel environment. This is especially true for distributed memory systems found in clusters based on off-the-shelf workstations communicating via fast interconnection networks.

5.2 Memory Hierarchy Optimisations

According to Moore's law from 1975, the number of transistors on a silicon chip will double every 12 to 24 months. This prediction has already proved remarkably accurate for almost three decades. It has led to an average increase in CPU speed of approximately 55% every year. In contrast, DRAM speed has evolved rather slowly. Main memory latency and memory bandwidth have only been improving by about 5% and 10% per year, respectively [33]. The *International Technology Roadmap for Semiconductors*[7] predicts that this trend will continue further on and the gap between CPU speed and main memory performance will grow for more than another decade until technological limits will be reached. Therefore, today's computer architectures commonly employ *memory hierarchies* in order to hide both the relatively low main memory bandwidth as well as the rather high latency of main memory accesses.

A typical memory hierarchy covers the CPU registers, up to three levels of cache, and main memory. Cache memories are commonly based on fast semiconductor SRAM technology. They are intended to contain copies of main

[7] See `http://public.itrs.net`.

memory blocks to speed up accesses to frequently needed data. The reason why caches can substantially reduce program execution time is the principle of *locality of references*. This principle is empirically established and states that most programs do not access all code or data uniformly. Instead, recently used data and data stored nearby the currently referenced data is very likely to be accessed in the near future. These properties are referred to as *spatial* and *temporal* locality, respectively [33].

Only if the hierarchical memory architecture is respected by the code, can efficient program execution (in terms of arithmetic operations per time unit) be expected. Unfortunately, current optimising compilers are not able to synthesise chains of complicated cache-based code transformations. Hence, they rarely deliver the performance expected by the users and much of the tedious and error-prone work concerning the tuning of the memory efficiency (particularly the utilisation of the cache levels) is thus left to the software developer. Typical cache optimisations techniques cover both *data layout transformations* as well as *data access transformations*.

Data layout optimisations aim at enhancing code performance by improving the arrangement of the data in address space. On one hand, such techniques can be applied to change the mapping of array data to the cache frames, thereby reducing the number of cache conflict misses. This is achieved by a layout transformation called *array padding* [52]. On the other hand, data layout optimisations can be applied to increase spatial locality. They can be used to reduce the size of the working set of a process; i.e., the number of virtual pages which are referenced alternatingly and should therefore be kept in main memory [33]. Furthermore, data layout transformations can be introduced in order to increase the reuse of cache blocks once they have been loaded into cache. Since cache blocks contain several data items that are arranged next to each other in address space, it is reasonable to aggregate data items in address space which are likely to be referenced within a short period of time. This is primarily accomplished by the application of *array merging* [33].

In contrast, data access optimisations change the order in which iterations in a loop nest are executed. These transformations primarily strive to improve both spatial and temporal locality. Moreover, they can also expose parallelism and make loop iterations vectorisable. Typical examples of data access transformations are *loop interchange* and *loop blocking (loop tiling)*, see [2]. Loop interchange reverses the order of loops in a loop nest, thereby reducing the *strides* of array-based computations; i.e., the step sizes at which the corresponding arrays are accessed. This implies an improved reuse of cache blocks and thus causes an increase in spatial locality. In contrast, loop blocking is primarily used to improve temporal locality by enhancing the reuse of data in cache and reducing the number of cache capacity misses. Tiling a single loop replaces it by a pair of loops. The inner loop of the new loop nest traverses a block of the original iteration space with the same increment as the original loop. The outer loop traverses the original iteration space with an increment equal to the size of the block which is traversed by the inner loop. Thus, the

outer loop feeds blocks of the whole iteration space to the inner loop which then executes them step by step.

Depending on the properties of the underlying problem (2D/3D, constant/variable coefficients, etc.), the application of appropriate cache optimisation techniques can yield significant speedups of up to a factor of 5 on common cache-based architectures. The impact of these techniques is typically examined by applying suitable *code profiling tools* [33].

For further details on cache optimisation techniques for numerical computations, we refer to [40, 58] and to the research in our *DiME*[8] project. In particular, we refer to [18] for an overview of cache optimisation techniques for multigrid methods.

5.3 Optimising for SMP Nodes

It is often the case that the individual nodes of a parallel computing environment consist of parallel architectures themselves. For example, many of today's workstation clusters are composed of so-called *symmetric multiprocessors (SMPs)*. SMP nodes are shared memory machines commonly consisting of two, four, or eight CPUs that access the local shared memory modules through fast interconnects such as a crossbar switches [33]. Lower levels of cache may be shared as well.

Within each SMP node, the parallel execution of the code is based on a shared memory programming model using *thread parallelism*. Thread parallelism can either be introduced automatically by parallelising compilers or by explicit programming; e.g., by using *OpenMP* directives [12, 13]. The individual nodes, however, typically communicate with each other using a distributed memory programming paradigm such as message passing.

The conclusions from the previous section carry over from single-CPU nodes to SMP nodes. Obviously, in the latter case, the efficient utilisation of the memory hierarchies is crucial for run-time performance as well. However, the manual introduction of cache-based transformations into thread-parallel code using OpenMP typically turns out to be even more tedious and error-prone.

6 Advanced Parallel Multigrid

In this section we will present a selection of extensions of the basic parallel multigrid method, as described in Sect. 3.

6.1 Parallelisation of Specific Smoothers

Multigrid methods come in many variants as required by specific applications. The treatment of anisotropies in particular is important in practice and has

[8] See `http://www10.informatik.uni-erlangen.de/dime`.

been studied extensively in the literature, see e.g. [57] and the references cited therein. In order to maintain full multigrid efficiency in the presence of mild anisotropies, one can use SOR smoothers with special weights [62], which in terms of a parallel implementation poses no additional difficulties. However, in the case of strong anisotropies, the ideal multigrid efficiency can only be maintained by either the use of non-standard coarsening strategies such as semi-coarsening, see [57], or the use of more advanced smoothers.

Fundamentally, the stronger coupling of the unknowns in a specific direction, as it is the case in anisotropic problems, must be observed in the algorithm. One way to accomplish this is to use so-called *line-smoothers*. These basically operate in the same fashion as the point smoothers described in Sect. 3.2 with the difference that the unknowns belonging to all nodes of a complete grid line are relaxed concurrently. In order to do this, a (small) linear system of equations, most often with tri-diagonal or banded structure, must be solved for each line. In view of a grid partitioning, this is uncritical as long as these lines of dependencies do not intersect sub-domain interfaces. If the tri-diagonal systems must be distributed across processes, then the solution of these systems must be distributed accordingly, where the usual starting point is the *cyclic reduction* algorithm; see [20], for example.

In the literature, many variants and extensions have been described, of which we mention only a few. Line smoothing is usually applied in a so-called *zebra order*, for example, akin to the red-black ordering for Gauß-Seidel and SOR, to simplify parallelisation. Having such a set of tri-diagonal systems which can be solved independently may also be used to parallelise (or vectorise) the simultaneous solution. This may lead to better parallel efficiency (or better vectorisation), since it avoids the need to devise parallel strategies for the inherent dependencies in the tri-diagonal systems.

Unfortunately, anisotropies aligned with the grid lines of a structured grid are just the simplest case. More complicated forms of anisotropies will require more complex smoothing strategies, such as alternating the direction of the line smoothers. This, however, again leads to problems with the parallelisation, since the lines of dependencies will then necessarily intersect sub-domain boundaries.

Finally, it should be mentioned that some 3D applications not only exhibit lines of strongly coupled unknowns, but planes of strong coupling. Smoothers adapted to this situation will treat all the unknowns in such a plane simultaneously. An efficient technique for solving the resulting linear systems is to revert to a 2D multigrid method [56]. Again, it may be necessary to do this in alternating plane directions, and each of the 2D multigrid algorithms may need line smoothing in order to be efficient. Any such strategy is naturally problem-dependent. Hence, it is difficult to develop any generally usable, robust multigrid method based on these techniques.

Besides anisotropies, the treatment of convection-diffusion equations (with dominating convection) is of high practical interest. In this case, the multigrid theory is still much less developed. One algorithmic multigrid approach

to convection dominated PDEs is based on smoothers with *downstream relaxation*, where the smoother is designed such that it observes the (sequential) data dependencies along the characteristics (that is streamlines) of the flow. Again, a parallelisation of such a smoother is easy when the domains are split parallel to the lines of dependency, but unfortunately this is far from trivial to accomplish in most practical situations where the streamlines may change direction throughout the domain or — in the case of nonlinear equations — may depend on the flow itself. In these cases, no generally applicable rules for the parallelisation exist, but the best strategy depends on the application.

Another smoothing approach successfully applied to both of the above problems is to employ variants of the *incomplete LU decomposition (ILU)* method; See [54, 57, 60], for example. It has been shown that ILU smoothers lead to robust and efficient multigrid methods in the sequential setting, at least for 2D problems. The 3D case, however, remains problematic with respect to efficiency. Only for problems with dominating directions efficient 3D ILU smoothers exist so far, see the remarks in [57]. From the parallel point of view, the problem is that, while ILU algorithms are in principle point-based smoothers, they are intrinsically sequential and therefore difficult to parallelise, the more so, since their quality depends even stronger on the ordering of unknowns than this is the case for the SOR method. For more details on parallel ILU smoothers, see the references in Chap. **??**, for example.

At the end of this subsection, we wish to give a word of warning. For any algorithm designer, it is tempting to simply neglect certain complexities that may arise from an efficient treatment of either anisotropies or dominating convection. This may be the case for sequential multigrid and is of course even more tempting when all the difficulties of parallelisation need to be addressed. In this case, algorithm designers often modify the algorithm slightly to simplify the parallel implementation. For example, the tridiagonal systems of line smoothers can simply be replaced by a collection of smaller tridiagonal systems as dictated by sub-domain boundaries, thus neglecting the dependencies across the sub-domains in the smoother. These smaller systems can be solved in parallel, benefiting parallel efficiency and simplifying the implementation. Such a simplified method will still converge because it is embedded in the overall multigrid iteration. In the case of only few sub-domains and moderate anisotropy, this may in fact lead to a fully satisfactory solver.

However, if the physics of the problem and the mathematical model really dictate a global dependency along the lines of anisotropy, then such a simplified treatment which does not fully address this feature will be penalised; the convergence rate will deteriorate with an increasing number of sub-domains to the point that the benefit of using a multigrid method is completely lost. Eventually there is no way to cheat the physics and the resulting mathematical properties of the problem. Multigrid methods have the disadvantage (or is this an advantage?) that they mercilessly punish any disregard for the underlying physics of the problem. Optimal multigrid performance with the typical convergence rates of 0.1 per iteration will only be achieved, if all the essential

features of the problem are treated correctly — and this may be not easy at all, especially in parallel.

6.2 Alternative Partitioning Approaches

In this section, we will briefly introduce three different concepts that can be seen as alternatives to the standard grid partitioning approach described in Sect. 3.1.

Additive and Overlapping Storage

The grid partitioning introduced in Sect. 3.1 assumed that each node of the grid belongs to a unique sub-domain and ghost nodes were employed to handle sub-domain dependencies. In this subsection, we will briefly discuss an approach to grid partitioning that strikes a different path. For a detailed analysis, see [17, 30, 36]. It can most easily be explained from a finite element point of view.

In the first approach to grid partitioning, nodes uniquely belong to sub-domains *(node-oriented decomposition)*. Thus, the elements are intersected by sub-domain boundaries. In the second approach, one assigns each element to a unique sub-domain *(element-oriented decomposition)*. In this case, nodes on the sub-domain boundaries belong to more than one sub-domain. This introduces the question of how to store the values of a grid-function (i.e., an FEM vector) at these nodes. One combines two different schemes. The first one, denoted as *overlapping* storage, assumes that each process stores the function values at the respective nodes. In this case, the connection between the local vector v_k on process p_k and the global vector v can be expressed by a boolean matrix M_k as $v_k = M_k v$.

In the second scheme, the global function value $v(n_j)$ at a node n_j on the boundary is split between all sub-domains to which n_j belongs. If s_p is the number of sub-domains, then the global function v can be obtained from the local sub-domain functions v_k via

$$v = \sum_{k=1}^{s_p} M_k^T v_k \ .$$

This is denoted as *adding* storage. Note that the conversion of an adding type vector to a vector of overlapping type requires communication, while the reverse operation can be performed without communication. In a similar fashion, one can define operators of adding and of overlapping type. This can be imagined in the following way. For an adding type operator a process stores all stencils for nodes in its sub-grid as vectors of adding type and analogously for an operator of overlapping type.

The application of an adding type operator to an overlapping type vector can be performed without communication and will result in a vector of

adding type. Under some constraints regarding the dependency pattern of the respective operator, the application of an operator of overlapping type to a vector of either adding or overlapping type can also be performed without communication and results in a vector of the original type. Proofs as well as a detailed analysis of the limiting conditions can be found in [30].

The element-oriented decomposition in combination with the above storage concept can be used to parallelise a multigrid method in the following fashion. One stores the approximate solutions and the coarse grid corrections as vectors of overlapping type and the right hand sides and residuals as vectors of adding type. The transfer operators are stored in overlapping fashion, while the discrete differential operator is stored as operator of adding type.

Under these assumptions (and assuming that the dependency patterns of the operators fulfil the limiting conditions) one can show that none of the multigrid components prolongation, restriction, computation of the residual, and coarse grid correction step requires any communication. The only place where communication is required is the smoothing process. Here, typically the update to the old approximate can be computed as vector of adding type, which must be converted to overlapping storage before adding it to the old approximate. As is the case for node-oriented decompositions, each smoothing step will thus require one communication step.

Another interesting aspect, though less general, is the following. Assume that our multigrid method employs a hierarchy of regular grids Ω^l composed of $(2^{m_l} + 1) \times (2^{n_l} + 1)$ nodes. As was mentioned in Sect. 3.3, the use of a node-oriented decomposition will lead to a hierarchy, where sub-grids on different levels cover different areas. With the element-oriented decomposition this problem does not arise. The sub-grid boundaries coincide on all levels. Furthermore, we will obtain a perfect load balance, as far as the number of grid nodes in each sub-grid is concerned. This is not the case for standard grid partitioning, see Fig. 2 in this respect.

Full Domain Partitioning

The partitioning schemes presented so far result in each process storing its patch (sub-grid) of the global grid plus some data from the immediate neighbouring patches. In this case, on the finer grid levels, each process knows its part of the global grid, but no process works on the whole problem domain. In the *Full Domain Partition* approach, in short *FuDoP*, proposed by Mitchell [44, 45, 46], each process starts from a coarse grid representing the whole problem domain and then adaptively refines the grid until the resolution is sufficiently accurate in its area of responsibility. As a result, each process computes a solution for the whole domain, albeit with a high accuracy only in its patch of the global domain on a grid that becomes increasingly coarse the further the distance to that patch.

The advantage of this approach is that existing serial, adaptive codes can be re-used in a parallel context. In [44], a V-cycle multigrid method is pre-

sented that requires communication on the finest and on the coarsest level only, but not on the levels in between. However, on the downside, this approach requires all to all communication as now the grids on any two processes overlap. Whether this approach shows run-time advantages over standard partitioning schemes depends on the number of processes, the number of grid levels and, as usual, on the network characteristics.

Bank and Holst [3] promote a similar technique for parallel grid generation to limit load imbalances in parallel computations on adaptively refined grids. These authors also stress that existing adaptive components can be employed in a parallel setting with comparable ease.

Space-Filling Curves

The issues of partitioning and load balancing in the context of parallel adaptive multigrid have also been addressed through the use of so-called *space-filling curves*. A space-filling curve represents a mapping Φ of the unit interval $I := [0, 1]$ to the space $\mathbb{R}^d$, $d = 2, 3$, such that the image $\Phi(I)$ has a positive measure. See [27, 63], for example. The computational domain Ω is supposed to be a subset of $\Phi(I)$ such that all nodes of the adaptively refined discretisation grid are elements of $\Phi(I)$ and thus passed by the space-filling curve. At this point we should mention the following two aspects. The first one is that *space-filling curves* are typically constructed as the limit of a family of recursively defined functions. In practice no real space-filling curve, but only a finite approximation from such a family is employed in the method. Secondly, the assumption that the curve passes through all nodes of the grid is not a restrictive one. The space-filling curve by its nature can be chosen such that it completely covers a domain in 2D / 3D. Choosing a fine enough approximation will thus fulfil the assumption.

The fundamental advantage of the partitioning approach based on space-filling curves is that it allows for inexpensive load balancing. The idea, namely, is to use the inverse mapping of Φ and to divide the unit interval I into partitions that approximately contain the same numbers of pre-images of grid nodes. These partitions of I are then mapped to the available processes. Therefore, the computational load is balanced exactly. As a consequence of this decomposition approach, the original d-dimensional graph partitioning problem, which can be shown to be NP-hard [50], is approximated by a 1D problem that is easy to solve.

However, it has been demonstrated that the resulting partitioning of the actual d-dimensional computational grid can be suboptimal and may involve much more communication overhead than necessary. For a comprehensive presentation, we again point to [63] and the references therein.

6.3 Reduced and Overlapped Communication

Overlapped Communication

A technique to improve the parallel efficiency of any parallel algorithm is to overlap communication and computation, see also Chap. **??** in this context. In parallel multigrid, this can work as follows. Consider the Jacobi smoother discussed in Sect. 3.2 and assume that we are using a grid partitioning with ghost nodes. One sweep of the Jacobi method does not require any communication. However, after the sweep, the values at the ghost nodes must be updated. Remember that the order in which the nodes are treated during the Jacobi sweep does not play a role. Thus, we are free to alter it to our taste. By first updating all nodes at the boundary of a sub-grid we can overlap communication and computation, since the update of the ghost nodes can be performed while the new approximate solution is computed for nodes in the interior of the sub-grid. This idea directly carries over to other multigrid components that work in the same fashion; i.e., prolongation and restriction. More sophisticated smoothers, such as red-black SOR, for example, can also be treated in this way. Though, the order of treating nodes may become more intricate.

Reduced Communication

While overlapping communication with computation can significantly reduce the costs for communication, there further exist other more radical approaches. At this point, we briefly want to mention an algorithm introduced by Brandt and Diskin in [9, 14]. Their parallel multigrid method abandons communication on several of the finest levels of the grid hierarchy. In a standard parallel version of multigrid, this is the place where the largest chunk of communication occurs, as far as the amount of transferred data is concerned. In this respect, their approach differs from the ones discussed in Sect. 3.4 which addressed efficiency problems on coarser grids arising from their less favourable surface-to-volume ratio.

The core idea of the algorithm by Brandt and Diskin can be traced back to the *segmental-refinement-type* procedures originally proposed to overcome storage problems on sequential computers [8]. It can briefly be described as follows. The basic parallelisation is again done by grid partitioning, where each sub-grid is augmented by an extended overlap region in the same fashion as mentioned in Sect. 3.2. However, communication between processes is completely restricted to the sub-domains on the coarsest grid level. Here, communication between the processes is required to form a common coarsest grid problem.

The overlap region fulfils two purposes. On one hand, if an appropriate relaxation scheme such as red-black Gauß-Seidel is chosen, this buffer slows the propagation of errors due to inexact values at the interfaces. On the other

hand, in multigrid, the coarse grid correction typically introduces some high-frequency errors on the fine grid. Since values at the interfaces cannot be smoothed, the algorithm cannot eliminate these components. But in elliptic problems high-frequency components decay quickly. Hence, the overlap region keeps these errors from affecting the inner values too much.

It is obvious that the algorithm will in most cases not be able to produce an exact solution of the discrete problem. However, if one is solving a PDE problem, it is actually the continuous solution one is really interested in. Since the latter is represented by the discrete solution only up to a discretisation error, a discrete solution will be valuable, as long as its algebraic error remains in the same order as the discretisation error. For a detailed analysis of the communication pattern and the possible benefits of this approach, see [14, 48, 47].

6.4 Alternative Parallel Multigrid Algorithms

As was mentioned in Sect. 2.5, standard multigrid methods traverse the grid hierarchy sequentially, typically either in the form of W-cycles or V-cycles. We already discussed this inherently sequential aspect of multigrid in Sect. 3.4 and considered some standard approaches for dealing with it. Here, we want to briefly mention some further alternatives that have been devised.

The BPX variant of Multigrid

In this context, the BPX algorithm (as proposed in [6]) and its analysis (see also [49] and the references therein) is of special importance. The BPX algorithm is usually used as a preconditioner for a Krylov subspace method, but it shares many features with the multigrid method. In fact, it can be regarded as an *additive* variant of the multigrid method.

In classical multigrid, as discussed so far, each grid level contributes its correction sequentially. If expressed by operators, this results in a representation of the multigrid cycle as a product of operators corresponding to the corrections on each level. Here, each level needs the input from the previous level in the grid hierarchy; therefore, it results in a multiplicative structure. For example, on the traversal to coarser grids, the smoothing has to be applied first, before the residuals for the coarser grids can be computed.

In BPX in contrast, all these corrections are computed *simultaneously* and are then added. Formally, this corresponds to a sum of the correction operators on each level and, consequently, it seems that the need for a sequential treatment of the levels has been avoided. Since this approach will only result in a convergent overall correction, the BPX algorithm is usually not used by itself, but instead employed as preconditioner. In this role, it can be shown to be an asymptotically optimal preconditioner. Thus, when combined with Krylov subspace acceleration, BPX results in an asymptotically optimal algorithm.

Unfortunately, the hope of the original authors that their algorithm would show better parallel properties per se (the original paper [6] was entitled "Parallel Multilevel Preconditioners") has only partially become true, since a closer look reveals that the BPX algorithm must internally compute a hierarchical sum of contributions from all the grid levels. To be more precise, the residuals have to be summed according to the grid transfer operators from the finest level to any of the auxiliary coarser levels. The restriction of the residual to the coarsest level will usually require intermediate quantities that are equivalent to computing the restriction of the residual to all intermediate levels. Therefore, computing these restrictions in parallel does not only create duplicated work, it also does not reduce the time needed to traverse the grid hierarchy in terms of a faster parallel execution. The coarsest level still requires a hierarchical traversal through all intermediate levels. Consequently, the BPX algorithm essentially requires the same sequential treatment of the grid hierarchy as the conventional multiplicative multigrid algorithm.

The above argument is of course theoretically motivated. In practice, other considerations may be essential and may change the picture. For example, the BPX algorithm may still be easier to implement in parallel, or it may have advantages in a particular adaptive refinement situation. For a comparison, see also [4, 26, 37].

Point-Block Algorithm and the Fully Adaptive Multigrid Method (FAMe)

The invention of the additive variant of multilevel algorithms, however, has shown that the strict sequential treatment of the levels is unnecessary and this has spurred a number of other ideas. For example the point-block algorithm in [23, 25] and the representation of all levels of the grid hierarchy in a single system, see [24, 53], relies on the analysis of additive multilevel systems.

For the parallelisation of a multilevel method, this construction and analysis can be exploited by realizing that the traversal of the grid hierarchy and the processing of the individual domain partitions can be decoupled.

In the point-block algorithm, it is exploited that each node of the finest grid may be associated with several coarser levels, and therefore it may belong to several discrete equations corresponding to these levels. It is now possible to set up this system of equations for each node. In a traversal through all fine grid nodes, one can now solve this system for each node. The resulting algorithm becomes a block relaxation method, with blocks of 1 to $\log(n)$ unknowns, corresponding to the number of levels to which a node belongs, and can be shown to have asymptotically optimal complexity. Imposing a grid partitioning, this algorithm can be parallelised. Note that, since the *multiple stencils* may extend deep into neighbouring partitions, the communication is more complicated.

Possibly the most far-reaching result towards an asynchronous execution of multilevel algorithms is the *fully adaptive multigrid* method of [53]. Here an

active set strategy is proposed together with a meta-algorithm which permits many different concurrent implementations. In particular, it is possible to design a multilevel algorithm such that the traversal between the levels in the sub-domains can be performed completely asynchronously, as long as the essential data dependencies are being observed. The meta algorithm gives rigorous criteria which of these data dependencies must be observed. One possible realization is then to monitor the data dependencies dynamically at run-time within the algorithm. Any such dependency which extends across process boundaries need only be activated, if it is essential for the convergence of the algorithm, but it can be delayed until, for instance, enough such data have accumulated to amortise the startup cost.

The algorithmic framework of the fully adaptive multigrid method permits many different parallel implementations and includes as a special case the classical parallel multigrid algorithms as well as the point-block method.

Multiple Coarse Grid Algorithms and Concurrent Methods

For the sake of completeness, we also want to mention the algorithms by Fredrickson and McBryan [19], Chan and Tuminaro [11], as well as Gannon and van Rosendale [21]. While these methods also address the topic of the sequentiality of cycling through the multigrid hierarchy and the loss of parallel efficiency on coarser grids, they were designed primarily with massively parallel systems in mind. Such systems are denoted as *fine-grain*; i.e., the number p of processes is on the same order as the number N of unknowns. In contrast, we have considered *coarse-grain* parallel systems so far; i.e., systems with $p \ll N$. Nevertheless, the methods deserve to be mentioned here, since they complement the aforementioned ideas of this section.

The Chan-Tuminaro and the Gannon-van Rosendale algorithms both belong to the class of *concurrent methods*. Their idea, not quite unlike the one of the BPX or the point-block algorithm, is to break up the sequential cycling structure and to allow for a concurrent treatment of the different grid levels; hence the name concurrent methods. The basic approach is to generate independent sub-problems for the different grid levels by projection onto orthogonal sub-spaces. The algorithms differ in the way this decomposition is performed and the way solutions are combined again. For details, we refer to the original publications cited above.

The algorithm of Fredrickson and McBryan follows a completely different approach. In a fine-grain setting the sequential cycling structure of a standard multigrid method will lead to a large number of idle processes while coarser grids are treated. The Fredrickson–McBryan algorithm retains the sequential cycling structure, and instead tries to take advantage of these idle processes. Opposed to standard multigrid, the method does not employ a single grid to compute a coarse grid correction, but composes on each level several coarse grid problems. Ideally, the additional information obtained from these *multiple coarse grids* can be used to improve the convergence rate of the multigrid

method, thus improving not only parallel efficiency, but also actual run-time. Indeed, an impressive improvement in convergence speed can be demonstrated for selected applications, which lead the authors to denote this approach as *parallel superconvergent multigrid* (PSMG).

While all three algorithms can in principle also be employed in a coarse-grain setting, the (theoretical) considerations in [42] strongly indicate that from a pure run-time perspective this is very unlikely to pay-off. Even in a fine-grain environment, the additional work induced by the methods seems to clearly over-compensate for the gain in convergence speed and/or improved parallelism.

7 Conclusions

The parallelisation of multigrid algorithms is a multifaceted field of ongoing research. Due to the difficulties resulting from the decreasing problem sizes on the coarse grids, parallel multigrid implementations are often characterized by relatively poor parallel efficiency when compared to competing methods. Nevertheless, it is the time required to solve a problem that finally matters. From this perspective, suitably designed multigrid methods are often by far superior to alternative elliptic PDE solvers.

Since the multigrid principle leads to a large variety of multigrid algorithms for different applications, it is impossible to derive general parallelisation approaches that address all multigrid variants. Instead, as we have pointed out repeatedly, the choice of appropriate multigrid components and their efficient parallel implementation is highly problem-dependent. As a consequence, our contribution can definitely not be complete and should be understood as an introductory overview to the development of parallel multigrid algorithms.

References

1. Alcouffe, R. E., Brandt, A., Dendy, J. E., and Painter, J. W.: The multi–grid methods for the diffusion equation with strongly discontinuous coefficients. *SIAM J. Sci. Stat. Comput.*, 2:430–454, (1981)
2. Allen, R. and Kennedy, K.: *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, San Francisco, California, USA, (2001)
3. Bank, R. E. and Holst, M.: A new paradigm for parallel adaptive meshing algorithms. *SIAM J. Sci. Comput.*, 22(4):1411–1443, (2000)
4. Bastian, P., Hackbusch, W., and Wittum, G.: Additive and multiplicative multi-grid — a comparison. *Computing*, 60(4):345–364, (1998)
5. Bergen, B. and Hülsemann, F.: Hierarchical hybrid grids: data structures and core algorithms for multigrid. *Numerical Linear Algebra with Applications*, 11:279–291, (2004)
6. Bramble, J. H., Pasciak, J. E., and Xu, J.: Parallel multilevel preconditioners. *Math. Comp.*, 55:1–22, (1990)

7. Brandt, A.: Multi–level adaptive solutions to boundary–value problems. *Math. Comp.*, 31:333–390, (1977)
8. Brandt, A.: *Multigrid techniques: 1984 guide with applications to fluid dynamics.* GMD–Studien Nr. 85. Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin, (1984)
9. Brandt, A. and Diskin, B.: Multigrid solvers on decomposed domains. In *Domain Decomposition Methods in Science and Engineering: The Sixth International Conference on Domain Decomposition*, volume 157 of *Contemporary Mathematics*, pages 135–155, Providence, Rhode Island, (1994). American Mathematical Society
10. Briggs, W., Henson, V., and McCormick, S.: *A Multigrid Tutorial.* SIAM, 2. edition, (2000)
11. Chan, T. F. and Tuminaro, R. S.: Analysis of a parallel multigrid algorithm. In Mandel, J., McCormick, S. F., Dendy, J. E., Farhat, C., Lonsdale, G., Parter, S. V., Ruge, J. W., and Stüben, K., editors, *Proceedings of the Fourth Copper Mountain Conference on Multigrid Methods*, pages 66–86, Philadelphia, (1989). SIAM
12. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R.: *Parallel Programming in OpenMP.* Morgan Kaufmann, (2001)
13. Dagum, L. and Menon, R.: OpenMP: An industry-standard API for shared-memory programming. *IEEE Comp. Science and Engineering*, 5(1):46–55, (1998)
14. Diskin, B.: Multigrid Solvers on Decomposed Domains. Master's thesis, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, (1993)
15. Douglas, C. C.: A review of numerous parallel multigrid methods. In Astfalk, G., editor, *Applications on Advanced Architecture Computers*, pages 187–202. SIAM, Philadelphia, (1996)
16. Douglas, C. C. and Douglas, M. B.: MGNet Bibliography. Department of Computer Science and the Center for Computational Sciences, University of Kentucky, Lexington, KY, USA and Department of Computer Science, Yale University, New Haven, CT, USA, 1991–2002 (last modified on September 28, 2002); see http://www.mgnet.org/mgnet-bib.html
17. Douglas, C. C., Haase, G., and Langer, U.: *A Tutorial on Elliptic PDE Solvers and their Parallelization.* SIAM, (2003)
18. Douglas, C. C., Hu, J., Kowarschik, M., Rüde, U., and Weiß, C.: Cache optimization for structured and unstructured grid multigrid. *Elect. Trans. Numer. Anal.*, 10:21–40, (2000)
19. Frederickson, P. O. and McBryan, O. A.: Parallel superconvergent multigrid. In McCormick, S. F., editor, *Multigrid Methods: Theory, Applications, and Supercomputing*, volume 110 of *Lecture Notes in Pure and Applied Mathematics*, pages 195–210. Marcel Dekker, New York, (1988)
20. Freeman, T. L. and Phillips, C.: *Parallel numerical algorithms.* Prentice Hall, New York, (1992)
21. Gannon, D. B. and Rosendale, J. R.: On the structure of parallelism in a highly concurrent pde solver. *J. Parallel Distrib. Comput.*, 3:106–135, (1986)
22. Greenbaum, A.: *Iterative Methods for Solving Linear Systems.* SIAM, (1997)
23. Griebel, M.: Grid– and point–oriented multilevel algorithms. In Hackbusch, W. and Wittum, G., editors, *Incomplete Decompositions (ILU) – Algorithms,*

Theory, and Applications, Notes on Numerical Fluid Mechanics, pages 32–46. Vieweg, Braunschweig, (1993)
24. Griebel, M.: Multilevel algorithms considered as iterative methods on semidefinite systems. *SIAM J. Sci. Stat. Comput.*, 15:547–565, (1994)
25. Griebel, M.: Parallel point–oriented multilevel methods. In *Multigrid Methods IV, Proceedings of the Fourth European Multigrid Conference, Amsterdam, July 6-9, 1993*, volume 116 of *ISNM*, pages 215–232, Basel, (1994). Birkhäuser
26. Griebel, M. and Oswald, P.: On the abstract theory of additive and multiplicative Schwarz algorithms. *Numer. Math.*, 70:163–180, (1995)
27. Griebel, M. and Zumbusch, G. W.: Hash-storage techniques for adaptive multilevel solvers and their domain decomposition parallelization. In Mandel, J., Farhat, C., and Cai, X.-C., editors, *Proceedings of Domain Decomposition Methods 10, DD10*, number 218 in Contemporary Mathematics, pages 279–286, Providence, (1998). AMS
28. Gropp, W., Lusk, E., Doss, N., and Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, (1996)
29. Gropp, W., Lusk, E., and Skjellum, A.: *Using MPI, Portable Parallel Programming with the Mesage-Passing Interface*. MIT Press, second edition, (1999)
30. Haase, G.: *Parallelisierung numerischer Algorithmen für partielle Differentialgleichungen*. B. G. Teubner Stuttgart – Leipzig, (1999)
31. Hackbusch, W.: *Multigrid Methods and Applications*, volume 4 of *Computational Mathematics*. Springer–Verlag, Berlin, (1985)
32. Hackbusch, W.: *Iterative Solution of Large Sparse Systems of Equations*, volume 95 of *Applied Mathematical Sciences*. Springer, (1993)
33. Hennessy, J. and Patterson, D.: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publisher, Inc., San Francisco, California, USA, 3. edition, (2003)
34. Hülsemann, F., Bergen, B., and Rüde, U.: Hierarchical hybrid grids as basis for parallel numerical solution of PDE. In Kosch, H., Böszörményi, L., and Hellwagner, H., editors, *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 840–843, Berlin, (2003). Springer
35. Hülsemann, F., Meinlschmidt, S., Bergen, B., Greiner, G., and Rüde, U.: *gridlib* – a parallel, object-oriented framework for hierarchical-hybrid grid structures in technical simulation and scientific visualization. In *High Performance Computing in Science and Engineering, Munich 2004. Transactions of the Second Joint HLRB and KONWIHR Result and Reviewing Workshop*, pages 37–50, Berlin, (2004). Springer
36. Jung, M.: On the parallelization of multi–grid methods using a non–overlapping domain decomposition data structure. *Appl. Numer. Math.*, 23(1):119–137, (1997)
37. Jung, M.: Parallel multiplicative and additive multilevel methods for elliptic problems in three–dimensional domains. In Topping, B. H. V., editor, *Advances in Computational Mechanics with Parallel and Distributed Processing*, pages 171–177, Edinburgh, (1997). Civil–Comp Press. Proceedings of the EURO–CM–PAR97, Lochinver, April 28 – May 1, 1997
38. Karypis, G. and Kumar, V.: A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, (1999)
39. Karypis, G. and Kumar, V.: Parallel multilevel k-way partition scheme for irregular graphs. *SIAM Review*, 41(2):278–300, (1999)

40. Kowarschik, M.: *Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures*. PhD thesis, Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, Universität Erlangen-Nürnberg, Erlangen, Germany, (2004). SCS Publishing House
41. Lötzbeyer, H. and Rüde, U.: Patch-adaptive multilevel iteration. *BIT*, 37:739–758, (1997)
42. Matheson, L. R. and Tarjan, R. E.: Parallelism in multigrid methods: how much is too much? *Int. J. Paral. Prog.*, 24:397–432, (1996)
43. McBryan, O. A., Frederickson, P. O., Linden, J., Schuller, A., Solchenbach, K., Stuben, K., Thole, C.-A., and Trottenberg, U.: Multigrid methods on parallel computers — a survey of recent developments. *Impact Comput. Sci. Eng.*, 3:1–75, (1991)
44. Mitchell, W. F.: A parallel multigrid method using the full domain partition. *Elect. Trans. Numer. Anal.*, 6:224–233, (1997)
45. Mitchell, W. F.: The full domain partition approach to distributing adaptive grids. *Appl. Numer. Math.*, 26:265–275, (1998)
46. Mitchell, W. F.: Parallel adaptive multilevel methods with full domain partitions. *App. Num. Anal. and Comp. Math.*, 1:36–48, (2004)
47. Mohr, M.: Low Communication Parallel Multigrid: A Fine Level Approach. In Bode, A., Ludwig, T., Karl, W., and Wismüller, R., editors, *Proceedings of Euro-Par 2000: Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 806–814. Springer, (2000)
48. Mohr, M. and Rüde, U.: Communication Reduced Parallel Multigrid: Analysis and Experiments. Technical Report 394, Institut für Mathematik, Universität Augsburg, (1998)
49. Oswald, P.: *Multilevel Finite Element Approximation, Theory and Applications*. Teubner Skripten zur Numerik. Teubner Verlag, Stuttgart, (1994)
50. Pothen, A.: Graph partitioning algorithms with applications to scientific computing. In Keyes, D. E., Sameh, A. H., and Venkatakrishnan, V., editors, *Parallel Numerical Algorithms*, volume 4 of *ICASE/LaRC Interdisciplinary Series in Science and Engineering*. Kluwer Academic Press, (1997)
51. Prieto, M., Llorente, I., and Tirado, F.: A Review of Regular Domain Partitioning. *SIAM News*, 33(1), (2000)
52. Rivera, G. and Tseng, C.-W.: Data Transformations for Eliminating Conflict Misses. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Montreal, Canada, (1998)
53. Rüde, U.: *Mathematical and Computational Techniques for Multilevel Adaptive Methods*, volume 13 of *Frontiers in Applied Mathematics*. SIAM, Philadelphia, (1993)
54. Saad, Y.: *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, (2003)
55. Stoer, J. and Bulirsch, R.: *Numerische Mathematik 2*. Springer, 4. edition, (2000)
56. Thole, C.-A. and Trottenberg, U.: Basic smoothing procedures for the multigrid treatment of elliptic 3D–operators. *Appl. Math. Comput.*, 19:333–345, (1986)
57. Trottenberg, U., Oosterlee, C. W., and Schüller, A.: *Multigrid*. Academic Press, London, (2000)
58. Weiß, C.: *Data Locality Optimizations for Multigrid Methods on Structured Grids*. PhD thesis, Lehrstuhl für Rechnertechnik und Rechnerorganisation,

Institut für Informatik, Technische Universität München, Munich, Germany, (2001)
59. Wienands, R. and Oosterlee, C. W.: On three-grid fourier analysis for multigrid. *SIAM J. Sci. Comput.*, 23(2):651–671, (2001)
60. Wittum, G.: On the robustness of ILU–smoothing. *SIAM J. Sci. Stat. Comput.*, 10:699–717, (1989)
61. Xie, D. and Scott, L.: The Parallel U–Cycle Multigrid Method. In *Virtual Proceedings of the 8th Copper Mountain Conference on Multigrid Methods*, (1997). Available at `http://www.mgnet.org`
62. Yavneh, I.: On red-black SOR smoothing in multigrid. *SIAM J. Sci. Comput.*, 17:180–192, (1996)
63. Zumbusch, G.: *Parallel Multilevel Methods — Adaptive Mesh Refinement and Loadbalancing.* Advances in Numerical Mathematics. Teubner, (2003)