Stencil Computations for PDE-based Applications with Examples from DUNE and hypre

C. Engwer[1], R. D. Falgout[2] and U. M. Yang[2]

[1]*Institute for Computational und Applied Mathematics, University of Münster, Orleans-Ring 10, D-48149 Münster, Germany E-mail: christian.engwer@uni-muenster.de*
[2]*Lawrence Livermore National Laboratory, 7000 East Avenue, Livermore, CA 94550, E-mail: rfalgout@llnl.gov, umyang@llnl.gov.* *This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-JRNL-681537.*

SUMMARY

Stencils are commonly used to implement efficient on-the-fly computations of linear operators arising from partial differential equations (PDEs). At the same time the term 'stencil' is not fully defined and can be interpreted differently depending on the application domain and the background of the software developers. Common features in stencil codes are the preservation of the structure given by the discretization of the PDE and the benefit of minimal data storage. We discuss stencil concepts of different complexity, show how they are used in modern software packages like *hypre* and DUNE, and discuss recent efforts to extend the software to enable stencil computations of more complex problems and methods such as inf-sup-stable Stokes discretizations and mixed finite element discretizations.

Received . . .

KEY WORDS: Stencil Computations, Partial Differential Equations

1. INTRODUCTION

Partial differential equations (PDEs) appear in many applications. In order to solve these equations on a computer, they are first discretized by a process such as finite differences, finite volumes, or finite elements. These discretizations often admit structure in the form of *stencils* and this structure can be exploited to produce highly efficient computations. In general, stencils represent computational patterns that repeat across the computational domain. They apply to finite differences, finite volumes, and finite elements. The grids they are associated with have structure, but this structure need not be limited to rectangular Cartesian domains as is most commonly done. For example, in [3, 9, 10], stencils are defined on triangular and prismatic domains as well and they are closely tied to an underlying finite element discretization. In *hypre*, stencils are defined similarly, but they are not tied to a specific discretization. Instead, a stencil is interpreted simply as a row of a "structured" matrix. In DUNE, the finite element stiffness matrices themselves are thought of as a kind of stencil, since stiffness matrices also form a repeating pattern when the grid is structured.

Computations represented by simple stencil patterns have already been studied by computer scientists through what is commonly referred to as the Jacobi iteration. In numerical analysis, the Jacobi iteration is a simple algorithm for solving linear systems of equations and in general does not have to have an underlying stencil representation. From a computational perspective, it primarily involves a matrix-vector multiply and is equivalent to a Matvec operation in BLAS (see GEMV in [4]). There have been many efforts to optimize the stencil-based Jacobi iteration, including but not restricted to the use of array padding, loop unrolling, NUMA-aware allocation [7], and a

combination of temporal and spatial blocking techniques [15]. In this paper, we discuss stencils in a more general context and give more complex examples of how they appear in practice. The goal is to provide computer scientists with information that enables new research in code optimization techniques that will benefit applications that use stencils.

2. OPERATOR REPRESENTATIONS ON STRUCTURED GRIDS

The general workflow in the simulation of PDE-based models consists of three major steps:

1. Construct a suitable computational mesh for the given problem domain.
2. Approximate the PDE with a discretization method to produce a system of equations.
3. Solve the system of equations.

In many applications we obtain a *linear* system of equations, where the linear operator can be mathematically described as a sparse matrix. Even for nonlinear problems, it is necessary to construct linearized operators, which then again form sparse matrices. This motivates the need for linear system solvers. To solve these systems, iterative solvers such as conjugate gradient and GMRES are often used [17], which consist of matrix-vector multiplications and basic vector operations. Their convergence is generally accelerated with suitable preconditioners.

Some of the most efficient preconditioners are multigrid methods. Multigrid methods [6] achieve their efficiency through a combination of smoothing and coarse-grid correction. The smoother is generally a simple iterative method such as Jacobi or Gauß-Seidel that is effective at reducing high-frequency error in a small number of iterations. The remaining errors are then eliminated on coarser grids and the resulting smaller linear systems during the coarse grid correction step. The algorithm requires suitable restriction and prolongation (or interpolation) operators to move between grid levels. The two-grid method proceeds by performing a few smoothing steps on the fine system $A\mathbf{u} = \mathbf{f}$, restricting the residual $\mathbf{r} = \mathbf{f} - A\tilde{\mathbf{u}}$, where $\tilde{\mathbf{u}}$ denotes an approximation to the solution $\mathbf{u}$, and then solving the coarse system $A_c\mathbf{e}_c = R\mathbf{r}$, where A_c is the coarse system matrix and R the restriction operator. The solution $\mathbf{e}_c$ is then interpolated to the fine level via a prolongation operator P and added to the approximate solution, $\tilde{\mathbf{u}} \leftarrow \tilde{\mathbf{u}} + P\mathbf{e}_c$, possibly followed by some smoothing steps. To obtain a multigrid method, this process is applied recursively to the coarse level. The method consists of various matrix-vector multiplications, which can also be viewed as stencil applications. Note that interpolation and restriction operators are rectangular matrices.

In many applications, the operator of the original system exhibits a well defined structure, and this structure can be exploited to store the linear operator more efficiently than in a general sparse matrix format. One of these approaches is the *stencil representation* and another is the *element-matrix representation*. The two approaches are closely related, and the latter could even be interpreted as a kind of *stencil* itself when the grid is structured. In both cases similar optimization techniques can be applied.

In the remainder of this section, we define the notion of stencils and grids and how they relate to sparse matrices through a simple one-dimensional example. We also describe finite element representations on structured grids and their relationship to stencils and matrices. These concepts are explicitly used in both *hypre* [11] and DUNE [2, 1], but they have a long history and are useful in a much broader context. First, we discuss sparse matrices in general.

If we consider a linear operator mapping from $\mathbb{R}^N$ to $\mathbb{R}^M$, this operator can be represented as an $M \times N$ matrix A and the application of the operator is a simple matrix-vector product. In PDE applications, the arising linear operators typically exhibit a sparse structure, meaning that most of the matrix entries are zero. There is a wide range of sparse matrix formats that store only the nonzero entries and the sparsity pattern. Perhaps the most well known format is the CRS (compressed row storage) format, also referred to as CSR (compressed sparse row). Storing all matrix information is definitely the most flexible approach, but not necessarily the most efficient. As we are interested in approaches that avoid the storage of the complete matrix, we do not discuss the various sparse matrix options in detail.

2.1. Stencil Representation

In the general setting, we define stencils on grids, where here the term *grid* denotes a set of discrete points in $\mathbb{R}^d$. Assuming for the moment that we are in $\mathbb{R}^2$, then each point in the grid Ω^g is associated with a unique ordered pair of integers (i, j) and labeled (x_i, y_j). That is, we can associate the grid with a set of indices,

$$\mathcal{G} = \{(i, j) : (x_i, y_j) \in \Omega^g\}. \tag{1}$$

Assuming some ordering of the grid points, we now define vectors. If $\mathbf{u}$ is such a vector, its s^{th} component is labeled $u_{i,j}$, where (x_i, y_j) is the s^{th} point on the grid. Note that since the set $\mathcal{G}$ represents a logical uniform version of the physical grid Ω^g, we use the terms "grid" and "point" in both cases, and provide clarification when necessary.

To simplify the setting, consider the one-dimensional fine and coarse grids defined respectively as the sets of indices $\mathcal{G} = \{1, 2, \ldots, 7\}$ and $\mathcal{G}_c = \{2, 4, 6\}$ with geometric representation

$$\begin{array}{ccccccccc} \cdot & & \times & & \cdot & & \times & & \cdot \\ 1 & & 2 & & 3 & & 4 & & 5 & & 6 & & 7 \end{array} \tag{2}$$

Vectors are related to these grids through their component indices as follows: $\mathbf{u} = (u_1, \ldots, u_7)^T$ and $\mathbf{u}_c = (u_2, u_4, u_6)^T$. Now suppose the matrix A is given by

$$A = \begin{pmatrix} C_1 & E_1 & & & & & & \\ W_2 & C_2 & E_2 & & & & & \\ & W_3 & C_3 & E_3 & & & & \\ & & W_4 & C_4 & E_4 & & & \\ & & & W_5 & C_5 & E_5 & & \\ & & & & W_6 & C_6 & E_6 & \\ & & & & & W_7 & C_7 & \end{pmatrix}. \tag{3}$$

Consider the *matrix-vector* multiply $A\mathbf{u}$. The operation of row i of A on the vector $\mathbf{u}$ is

$$(A\mathbf{u})_i = W_i u_{i-1} + C_i u_i + E_i u_{i+1}. \tag{4}$$

Now represent A (row-by-row) with the stencil

$$A \sim \begin{bmatrix} W_i & C_i & E_i \end{bmatrix}. \tag{5}$$

We see that if (5) is placed over the grid $\mathcal{G}$ of (2) and centered at point i, it describes the operation in (4). That is, a stencil operation on a grid has a one-to-one correspondence with a matrix row operation. Note that if $i = 1$ or $i = 7$, the coefficients W_1, E_7 and the variables u_0, u_8 are not defined. Thus, to avoid listing boundary stencils separately, we assume that if a stencil coefficient reaches outside of the grid, the coefficient is zero. Similarly, we may assume that $u_i = 0$ for all $i \notin \mathcal{G}$. For periodic problems, we can cause the grid in (2) to wrap around on itself by declaring that grid point 0 is the same as grid point 7 (and similarly that grid points 8 and 1 are the same). Hence, a nonzero stencil coefficient at the boundary wraps around to the other side of the domain. This has the effect of adding a nonzero coefficient with value W_1 in row 1, column 7 in (3) (and similarly the nonzero E_7 in row 7, column 1). It does not change how the coefficients are stored, however. Finally, note that if any stencil coefficient is constant across the grid, e.g. $C_i = 2$ for all i, the constant value only needs to be stored once, leading to significant memory savings, while the remaining coefficients vary over the grid.

As mentioned in the previous section, stencils can also represent rectangular matrices such as the interpolation and restriction operators in multigrid solvers [6]. For example, if we assume that

$C_i = 1$, the interpolation operator that is generally used with A in (3) is given by the matrix

$$P = \begin{pmatrix} -E_1 & & & \\ 1 & & & \\ -W_3 & -E_3 & & \\ & 1 & & \\ & -W_5 & -E_5 & \\ & & 1 & \\ & & -W_7 & \end{pmatrix}. \tag{6}$$

We see that (6) has two different types of row operations: one for even numbered rows and one for odd numbered rows. More precisely, row operations for (6) are defined by

$$(P\mathbf{u})_i = \begin{cases} -W_i u_{i-1} - E_i u_{i+1} & i \in \mathcal{G} \setminus \mathcal{G}_c, \\ u_i & i \in \mathcal{G}_c. \end{cases} \tag{7}$$

Denoting the sets $\mathcal{G} \setminus \mathcal{G}_c$ and $\mathcal{G}_c$ by the letters f and c respectively, then (6) can be represented by the stencil

$$P \sim \begin{bmatrix} -W_i & 1 & -E_i \end{bmatrix}_c = \begin{bmatrix} -W_i & \cdot & -E_i \end{bmatrix}_c^f \oplus \begin{bmatrix} \cdot & 1 & \cdot \end{bmatrix}_c^c, \tag{8}$$

where $\oplus$ denotes direct sum (the range spaces of the two stencils are orthogonal subspaces of the range space of P). If $i \in \mathcal{G} \setminus \mathcal{G}_c$, we obtain the result $(P\mathbf{u})_i$ by applying the stencil with superscript f to $\mathcal{G}$ centered at point i. Likewise, if $i \in \mathcal{G}_c$, we obtain the result $(P\mathbf{u})_i$ by applying the stencil with superscript c to $\mathcal{G}$ centered at i. Equation (8) only operates on the vector components $\{u_2, u_4, u_6\}$. Note that (6) can also be expressed using a "column" stencil. Stencils of this type have traditionally been represented in the multigrid literature with the outward bracket notation given by

$$P \sim \big] \begin{matrix} -E_{i-1} \\ 1 \\ -W_{i+1} \end{matrix} \big[. \tag{9}$$

In general, a stencil representation of a matrix for a one dimensional problem is defined as follows:

$$\begin{bmatrix} \cdots & X_i^{-1} & X_i^0 & X_i^{+1} \cdots \end{bmatrix}_{\textbf{dom}}^{\textbf{ran}}, \tag{10}$$

where **dom** indicates the domain grid to which (10) is applied, **ran** signifies the range of (10), and the domain and range are related through a common index space of integer values. A stencil-based representation of a matrix can now be written as

$$A = \sum_{p \in \mathcal{G}_{\text{ran}}} \mathbf{q}_p \, \mathbf{s}_p^T Q_p^T,$$

where $\mathbf{s}_p$ is a local vector of stencil coefficients for point p (recall that these coefficients also correspond to nonzeros in some row of the matrix), and $\mathbf{q}_p$ and Q_p are mappings from the local numbering to the respective row number and column numbers of the global matrix. To be more specific, assume that $\mathcal{G}_{\text{ran}}$ is a range grid with n grid points, and p is the m-th point in the grid. Also assume that the matrix is square for simplicity. Define $\mathbf{q}_p \in \mathbb{R}^n$ as the vector whose m-th coefficient is 1 and all other coefficients are 0, and $\mathbf{s}_p \in \mathbb{R}^k$ as the vector of coefficients for the k-point stencil. The $n \times k$ matrix Q_p maps the local stencil for p to its global column numbers, i.e., if the i-th coefficent in $\mathbf{s}_p$ points to the j-th point in the grid, the j-th coefficient in the i-th column of Q_p is 1 while all of its other coefficients are zero. With this representation, we can rewrite matrix-vector multiplication in terms of local operations on local vectors $\mathbf{u}_p \in \mathbb{R}^k$ as follows:

$$A\mathbf{u} = \sum_{p \in \mathcal{G}_{\text{ran}}} \mathbf{q}_p(\mathbf{s}_p^T(Q_p^T\mathbf{u})) = \sum_{p \in \mathcal{G}_{\text{ran}}} \mathbf{q}_p(\mathbf{s}_p^T\mathbf{u}_p).$$

2.2. Element-Matrix Representation

The element-matrix representation is the usual approach in finite element codes to assemble the global matrix, but it can just as well be used to avoid a global assembly completely. Following the 1D example above, consider the element matrix

$$A_\alpha = \frac{1}{2}\begin{pmatrix} C_\alpha & E_\alpha \\ W_{\alpha+1} & C_{\alpha+1} \end{pmatrix}, \tag{11}$$

defined on the element $[x_\alpha, x_{\alpha+1}]$ where $\alpha \in \mathcal{T}(\Omega) = \{1, 2, \ldots, 6\}$. The matrix A in (3), as well as finite element matrices in general, can then be decomposed into local matrix contributions A_α:

$$A = \sum_{\alpha \in \mathcal{T}(\Omega)} Q_\alpha A_\alpha Q_\alpha^T ,$$

where $\mathcal{T}$ denotes the computational mesh, α is an element in this mesh, and Q_α maps from element local numbering to global numbering. These are local matrices describing the contributions on a single element α. If we consider a vertex-based discretization (i.e., Q^1 ansatz functions) on a quadrilateral mesh, this means that the matrix A_α is a 4×4 matrix.

This representation allows us to rewrite matrix-vector products in terms of local contributions

$$A\mathbf{u} = \sum_{\alpha \in \mathcal{T}(\Omega)} Q_\alpha (A_\alpha (Q_\alpha^T \mathbf{u})) = \sum_{\alpha \in \mathcal{T}(\Omega)} Q_\alpha (A_\alpha \mathbf{u}_\alpha)$$

and thus enables a flexible on-the-fly representation of the linear operator. *Note:* the actual performance of this approach strongly depends on the mesh data structures and the amount of work on each element. As illustrated in [13], using an element-matrix representation for higher order methods can achieve close-to-peak Flop performance.

For multigrid solvers, the necessary prolongation and restriction operators can easily be represented using element matrices. In this case the global matrix is a rectangular matrix. The local matrices are however (generally) still square matrices, since the local contributions on a single mesh element represent the coupling of the shape-functions on the fine mesh with the shape-functions on the coarse mesh; in both spaces the number of basis-functions with local support is the same, 4 for Q^1-FEM on a quadrilateral mesh.

The application of the prolongation to a coarse mesh vector can be formulated as

$$P\mathbf{u} = \sum_{\alpha \in \mathcal{T}^c(\Omega)} \sum_{\alpha' \in \mathcal{C}(\alpha)} Q_{\alpha'} (P_\alpha^{\alpha'} (Q_\alpha^T \mathbf{u}))$$

where $\mathcal{C}(\alpha)$ denotes the set of child-elements of α and $P_{\alpha'}^{\alpha}$ is the local prolongation operator mapping from the coarse element α to the fine element α'.

2.3. Performance Optimization for Stencil-based Codes

The so-called Jacobi iteration mentioned in the introduction is a simple stencil-based computation that has been studied extensively in the context of performance optimization. The iteration studied in the literature is not always strictly equivalent to the Jacobi method used to solve linear systems, but it exhibits the primary computational pattern of importance (so it is Jacobi-like). From a mathematical point of view, one iteration of Jacobi for solving the linear system $C\mathbf{u} = \mathbf{f}$ is given by $\mathbf{u}^k = \mathbf{u}^{k-1} + D^{-1}(\mathbf{f} - C\mathbf{u}^{k-1})$, where D is the diagonal matrix containing the diagonal of C, and $\mathbf{u}^k$ is the k-th iterate to approximate the solution $\mathbf{u}$. If we let $A = I - D^{-1}C$, then $\mathbf{u}^k = A\mathbf{u}^{k-1} + D^{-1}\mathbf{f}$, and it is easy to see that the main computational component of the Jacobi method is simply a matrix-vector multiply, which can be written as in Algorithm 1.

Algorithm 1 Jacobi: matrix-multiply form

for $k = 1$ **to** K **do**
  $\mathbf{u}^k = A\mathbf{u}^{k-1}$
end for

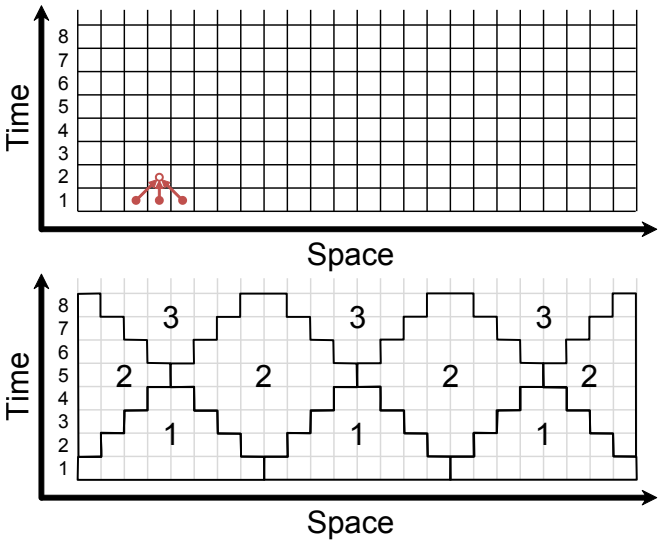

Figure 1. Jacobi algorithm with 3-point stencil in 1D (top) and temporal blocking optimization using diamond tiling (bottom). The data dependency between time levels (iterations) is shown with red arrows. Diamond-shaped tiles with the same number are independent of each other and can be computed in parallel. Tile sizes are chosen based on various hardware characteristics such as cache size.

Using the example of the 1D spatial grid and 3-point stencil in equations (3)–(5), it is easy to see that the algorithm can be written as a stencil application in Algorithm 2.

Algorithm 2 Jacobi: 3-point stencil in 1D

for $k = 1$ **to** K **do**
  for $i = 1$ **to** N **do**
    $u_i^k = W_i \cdot u_{i-1}^{k-1} + C_i \cdot u_i^{k-1} + E_i \cdot u_{i+1}^{k-1}$
  end for
end for

In the algorithm, space is referenced by index i and time (or iteration) by index k. The values W_i, C_i, and E_i are coefficients that may or may not vary spatially. For example, C_i may be constant as in the stencil (8).

In general, the matrix-vector multiply in Algorithm 1 need not involve a stencil operation at all, as in the case of unstructured-grid codes. In this paper, we are primarily interested in cases where either all or part of the multiply can be represented as a stencil operation. In particular, we give examples of higher-dimensional stencils in Section 3 for a variety of applications, and in Section 4 we discuss the use of stencils in the software libraries *hypre* and DUNE.

In the context of performance optimization, the Jacobi algorithm in 2 is the most commonly studied form of the iteration, though it has also been studied extensively for the 5-point stencil in 2D and the 7-point stencil in 3D. For simplicity, we focus here on the 1D case. A graphical illustration of Algorithm 2 is shown in Figure 1 (top). Many optimization techniques have been developed for this algorithm, too many to be listed here. However we will mention a few. One of the earliest approaches is the wavefront technique introduced in [14]. A variety of optimization techniques, such as array padding, multilevel blocking, loop unrolling and reordering, and more are considered for the 7-point stencil in [7]. Another more recent temporal blocking approach is the diamond tiling technique depicted in Figure 1 (bottom) and described in some detail in [15]. The latter paper also discusses the 3D spatial setting.

One advantage of the Jacobi example above is that it is easy to understand and also provides many optimization opportunities. In practice, however, Jacobi is often not the best solver, especially

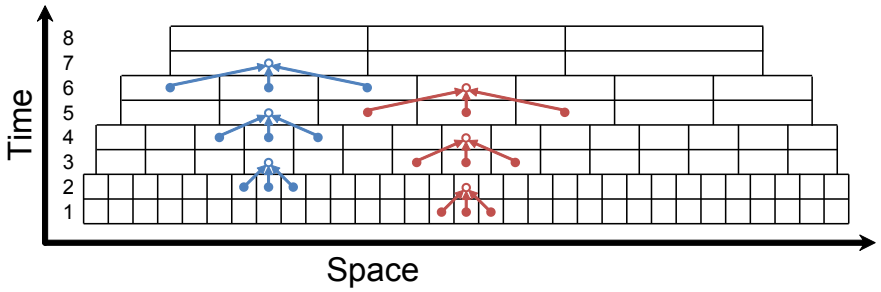

Figure 2. Multigrid algorithm in 1D. Data dependencies within grid levels are shown with red arrows (square matrix operations) and data dependencies between grid levels are shown with blue arrows (rectangular restriction-matrix operations). This represents only the first half of the so-called multigrid V-cycle. The second half of the cycle is the inverse of this diagram; it starts at the top and proceeds down.

for large-scale PDE simulations. One of the fastest solvers in these settings is multigrid [6]. As in Algorithm 1, multigrid can also be described as a sequence of matrix-vector multiplies. However, the multiplies involve both square and rectangular matrices. In the simple 1D setting, this leads to the picture shown in Figure 2. Because of the wide data dependency graph, temporal blocking is more difficult. Note that it is easy to increase the number of square matrix-vector operations on each grid level to increase optimization opportunities [8], but if this does not lead to a sufficient decrease in iterations, the extra computations generally produce little gain in overall solver speed.

3. EXAMPLES OF STENCIL REPRESENTATIONS

In this section, we consider three examples to illustrate more complex stencil situations. We begin with a simple 2D Laplace example, then discuss a Stokes problem and a more involved discretization of diffusion problems. The examples are all presented on grids (meshes) that contain large Cartesian components consisting of rectangular *cells* with data located around the cells at mesh *entities* such as cell centers, vertices (nodes), faces, and edges (in 3D). See Figure 3 for a 2D illustration.

3.1. 2D Laplace stencil

In the first example, we consider the Laplace equation in 2D, a standard test problem for elliptic PDEs. For a given domain Ω with boundary $\Gamma = \Gamma_D \cup \Gamma_N \cup \Gamma_R$ and a given source function f, we want to find u that satisfies the following equations:

$$-\Delta u = f \quad \text{on } \Omega \tag{12a}$$
$$u = g \quad \text{on } \Gamma_D \tag{12b}$$
$$\partial_n u = j \quad \text{on } \Gamma_N \tag{12c}$$
$$u + \alpha \partial_n u = k \quad \text{on } \Gamma_R\ . \tag{12d}$$

We can distinguish between three basic types of boundary conditions, each leading to a distinct stencil on the boundary: Dirichlet conditions on Γ_D impose a fixed value on the boundary; Neumann conditions on Γ_N impose a given flux through the boundary; and Robin conditions on Γ_R are a mixture of both. We present two stencils produced by two common discretization choices. Note that we omit scaling factors, since focus here is the stencil shape. Scaling factors can be included in the right hand side or can vary depending on the discretization scheme used to generate the stencil.

For a first order finite element discretization on a structured quadrilateral mesh, we obtain the well known 9-point stencil,

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}. \tag{13}$$

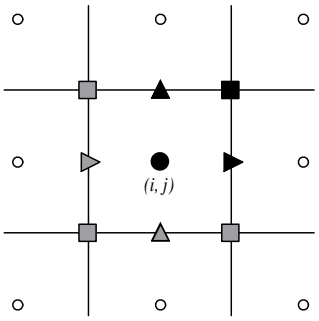

Figure 3. Cell-centered, x-face, y-face and nodal data points on a cell.

This example relates neighboring vertices. Another common discretization is the cell-centered finite volume approach. Here the unknowns are associated with cells and connected via faces to produce the classical 5-point stencil,

$$\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}. \tag{14}$$

Note that this stencil can also be generated using a finite difference discretization on a node-centered grid or finite elements on a square triangulated mesh.

These two examples demonstrate the need for code that can manage different types of grid entities, but this is actually easy to accomplish here. Because each stencil involves either vertices or cells (not both), we can produce a dual mesh by shifting the original cell mesh. On the dual mesh, the cells become vertices, so the two entity types can be treated as if they were the same. This approach is not sufficient in general, however. Depending on the exact type of discretization (e.g., higher-order finite element methods) and the type of model, unknowns can also be associated with faces or edges (in 3D) or a mixture of these entities. The next two sections provide examples.

3.2. 2D Stokes problem

A well-known problem is the Stokes problem. It is important in many engineering applications, yet it is one of the simplest models that employs a mixture of grid entity types. The physical problem is given by the following equation and an additional constraint, which ensures incompressibility:

$$-\nu\Delta\vec{v} + \nabla p = \vec{f} \tag{15a}$$

$$\nabla \cdot \vec{v} = 0 \tag{15b}$$

Here we have to discretize three physical quantities, the two velocity components v_x, v_y and the pressure p. A wide range of mathematical formulations for these equations are available and the actual choice depends on the particular problem being solved.

In the context of this paper, one issue for the Stokes problem is that we are not totally free to adapt the data layout to best fit the hardware. On one hand the data layout, i.e. the numbering of unknowns, has an immediate impact on the memory access pattern and on the overall performance of the algorithm. On the other hand, the discretization has to satisfy certain properties to ensure existence and uniqueness of the solution, namely, it must satisfy the discrete inf-sup condition known as the LBB condition [5]. A consequence of this property is that the discretization must have fewer degrees of freedom for the pressure unknown than for either of the velocity components and thus the memory layout for pressure and velocity must differ.

A common choice for the finite element discretization is the Taylor-Hood element [18], which assumes piecewise (bi-)quadratic functions for the velocity components and piecewise (bi-)linears for the pressure. The degrees of freedom for this formulation are oriented as in Figure 4, i.e., two velocity unknowns v_x, v_y at each vertex, face, and cell, and one pressure unknown p at each vertex.

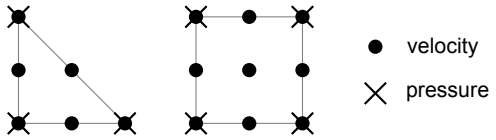

Figure 4. Layout of unknowns for the Taylor-Hood Stokes finite element on triangles and quadrilaterals.

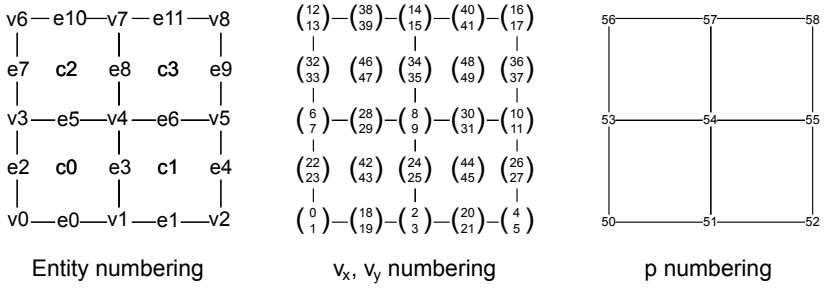

Figure 5. Example of a mapping from entity numbering (left) to a global numbering of the degrees of freedom (center, right). This particular numbering of velocity and pressure unknowns (v_x, v_y, p) is currently implemented in DUNE.

The associated linear system takes the form

$$\begin{pmatrix} A & B \\ -B^T & 0 \end{pmatrix} \begin{pmatrix} \mathbf{v} \\ \mathbf{p} \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ 0 \end{pmatrix} , \tag{16}$$

where A is a quadratic matrix representing a vector-valued Laplace operator, and B is a rectangular matrix representing a discrete gradient operator.

An efficient approach for solving the linear system is the so called Schur-complement approach. Here the equations are rewritten so that the pressure satisfies the system

$$S\mathbf{p} = B^T A^{-1} \mathbf{f} \, , \tag{17}$$

where $S = B^T A^{-1} B$ is called the Schur complement. The matrix S is never computed explicitly, as it is dense. Instead, S is applied by calling the individual operators. Note that applying A^{-1} means we have to solve a linear system. Since A, B, and B^T can be represented by stencils, it is possible to employ fast stencil-based techniques to achieve a high algorithmic intensity, even for this more challenging problem. *Remark:* A similar data layout is obtained when considering a finite difference discretization on a staggered grid.

To globally store the data, a common approach is to block the velocity components per entity. Two numberings are employed: first all entities are numbered from 0 to $N_v - 1$, then the vertices are numbered lexicographically from 0 to $N_p - 1$. The total number of unknowns is $2N_v + N_p$. The velocity components (v_x, v_y) at a particular entity $i \in [0, N_v)$ are stored at positions $2i$ and $2i + 1$, and the pressure at vertex $j \in [0, N_p)$ is stored at position $2N_v + j$. In DUNE, both numberings are constructed from numberings per entity type as described in Section 4.1.3 and the resulting numbering for the 2D Stokes problem is depicted in Figure 5.

Implementing this discretization in terms of stencils requires nine different stencils as illustrated in Figure 6: four to compute the velocity-velocity operator A; four for the velocity-pressure operator $-B^T$; and one for the pressure-velocity operator B. In reality, only the five stencils for A and B are needed, since the operation of $-B^T$ can be implemented in terms of B.

3.3. 2D Local Support-Operator Diffusion Scheme

A cell-centered discretization scheme for arbitrary quadrilateral meshes is introduced in [16] for the diffusion problem

$$\frac{\partial u}{\partial t} - \nabla \cdot D \nabla u = f \tag{18}$$

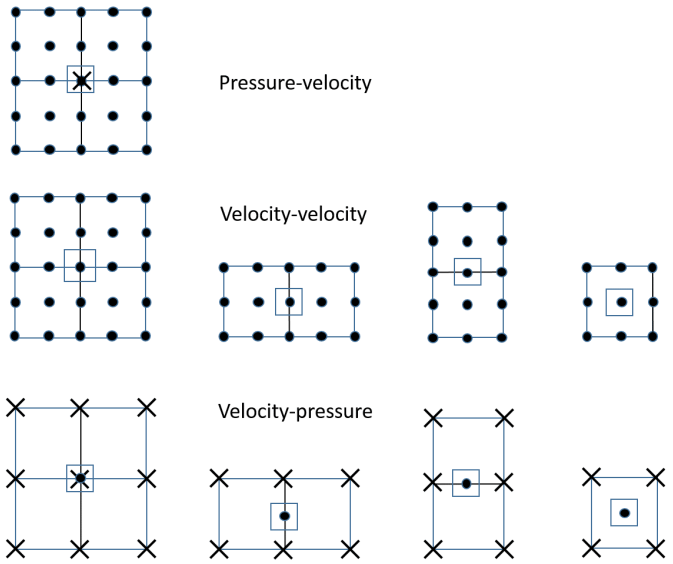

Figure 6. Stencils for the Stokes problem. Center points (enclosed in squares) are connected to all other points (the connections are omitted). The velocity points (black dots) represent two velocity components v_x and v_y. The stencils (ordered left to right and top to bottom) have 51, 50, 30, 30, 18, 10, 7, 7, and 5 points.

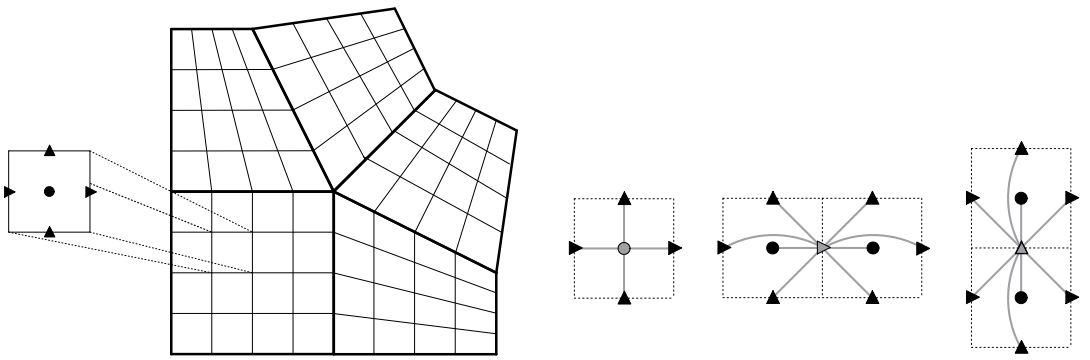

Figure 7. A block-structured grid with cell-centered and face-centered variables (left) and the stencils generated by the local support-operator diffusion scheme (right). The stencil for the u equations is centered at cells and the two stencils for the F equations are centered at faces.

on a logically Cartesian 2D grid. The scheme introduces the flux variable $F = -D\nabla u$, and associates variable u with cell centers and F with cell faces as shown in Figure 7. The discrete equations can be described via three stencils, also depicted in the figure. Note that this finite difference formulation is equivalent to a mixed finite element formulation and to mimetic finite differences.

One way to implement this scheme is to block the unknowns by entity type. Assigning numbers to the types cell (1), x-face (2), and y-face (3), this approach leads to the block 3×3 matrix

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}. \tag{19}$$

Each submatrix A_{ij} contains stencil coefficients that couple unknown type i to j, and each can be described in terms of (sub)grids and (sub)stencils. For $i = j$ the submatrix is square, A_{11} is a diagonal matrix, and A_{22} and A_{33} have 3 nonzeros per row. For $i \neq j$ the matrices are rectangular

with 2 to 4 nonzeros per row. Since A is a sparse matrix, the scheme is called "local", whereas the traditional support-operator method yields dense matrices.

Note that this discretization scheme is of second-order accuracy even on non-smooth meshes. It also translates naturally to block-structured grids as in Figure 7. In particular, it extends straightforwardly across the five structured grids in the figure, even around the corner point in the intersection of the grids. Grids of this type can be handled by the semi-structured interface in *hypre*, which is discussed in Section 4.2.2.

4. IMPLEMENTATION DETAILS OF OPERATOR CONCEPTS

In this section we compare the implementation concepts of DUNE and *hypre*. The DUNE library focuses on generality and aims mainly at solving PDEs on unstructured meshes, while *hypre* is mainly concerned with linear algebra and linear solvers. Traditionally DUNE uses classical matrix formats, like CSR, or the more recently adopted SELL-C-σ format [12], an efficient sparse matrix format for CPUs and accelerators. All of these formats have in common that they explicitly store only non-zero entries. The *hypre* library offers a rich choice of optimized matrix formats for semi-structured data, where stencil-like representations are applicable and the memory footprint can be reduced dramatically. While the DUNE approach is more flexible, it is also more expensive.

4.1. Implementation in DUNE

DUNE allows for a range of different numerical methods, but in the following we describe the usual approach as it is used in the DUNE/PDELab finite element package. A core feature of DUNE is the grid interface [2, 1], which allows one to describe a very general class of meshes and provide specialized implementations under a common interface. For example, DUNE can describe unstructured, locally refined, simplicial grids, as well as structured hexahedral grids. The latter class of grids is the typical structure that is considered in stencil codes.

In order to support a broad class of meshes, DUNE/PDELab uses an element-based representation of the operator. This follows the element matrix representation described in Section 2. With this representation, it is possible to either assemble the global matrix A or immediately apply the operator without a full assembly. In the latter case the element matrices are usually computed on-the-fly and have to be recomputed for every cell. Depending on the a-priori information, the developer can avoid certain re-computations, e.g. evaluation of all shape-functions at all quadrature points, but such optimisations can not be performed automatically by the framework. Symmetry and regularity of the grid can, to some extent, be exploited via the DUNE Grid Interface, as will be discussed next.

4.1.1. The DUNE *Grid Interface* allows modeling of structured and unstructured grids in a unified fashion. Using generic C++ programming techniques, DUNE provides a general interface to all of these implementations. Similarly to the idea of STL algorithms, it is possible to write algorithms based on these interfaces without knowing the actual implementation. For DUNE, a wide range of grid implementations exist, ranging from simple structured grids to parallel and locally adaptive grids, to special purpose implementations like surface meshes of network grids (see Figure 8).

In DUNE/PDELab such algorithms are provided to implement finite element discretizations on arbitrary DUNE grids. In order to implement the element matrix formulation, the first abstraction needed is a loop over all grid elements. Using the iterator concept, all cells are ordered linearly and an iterator allows one to loop through the elements in this linear (implementation dependent) ordering. Now on each element the local matrix A_E has to be computed and must be mapped back to the global numbering. The mapping from local to global numbering uses two main features of the DUNE interfaces. For each basis function with element local support, we can associate a local degree of freedom and thus a row in the matrix A_E. These local basis functions are associated with subentities of the cell, e.g., the fourth vertex of the cell or the second face of the cell. Together with

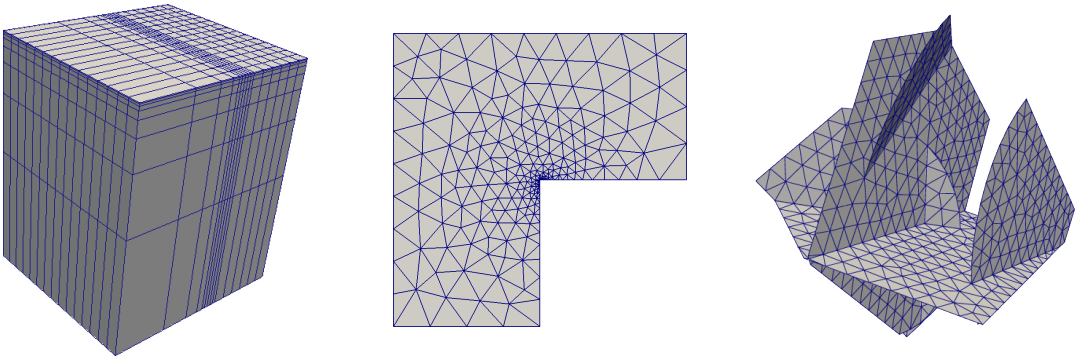

Figure 8. Examples for possible grids covered by the DUNE grid interface specification: tensor structured grids (left), fully unstructured and locally adaptive grids (middle), embedded surface grid (right) [§].

a (per process) global consecutive numbering of all entities, it is possible to construct a consecutive global index for all degrees of freedom and to compute the global index for a given local unknown.

4.1.2. Structured Grids are one possible implementation of the general DUNE grid interface. The structured grid implementation in DUNE (called YaspGrid) offers tensor-product grids, i.e., non-equidistant structured grids in arbitrary space dimensions. On one hand, the grid provides topological information that can be used to compute the indices, while on the other hand, it provides geometric information that is necessary to implement a general finite element formulation. Due to the well defined structure of the grid, it is possible to simplify many of the geometry computations. If the grid is equidistant, all cells are only translations of the first cell and thus most of the geometric information does not change throughout the grid.

While it is possible to avoid recomputing this information, it is difficult to further use the fact that the information does not change. In the extreme case, the local matrices A_E would not change throughout the grid, but in the current interfaces, it is not possible to carry this information to the higher level components of DUNE/PDELab. However, this is only a minor issue, because the case where A_E is actually constant for all cells is an academic extreme, and for nearly all applications some kind of locally varying information enters the model. This information might be some non-linearity like in the Navier-Stokes equations, local parameters like in porous media applications, or locally varying geometric information like in tensor-product meshes.

4.1.3. Local-Global Index mapping is constructed using a set of consecutive indices for each type of entity in the mesh. Consider the case of the finite element Taylor-Hood discretization. For Q^2 shape functions we have one unknown per vertex, face, edge and cell, and for Q^1 we only have an unknown at each vertex. For the velocity we have three Q^2 spaces, as we have vector-valued data. As suggested in Section 3.2, the storage layout can simply be computed from these indices by grouping unknowns per entity type. The resulting solution vector $\mathbf{u}$ would be given as

$$\mathbf{u} = [\mathbf{v}^v, \mathbf{v}^e, \mathbf{v}^f, \mathbf{v}^c, \mathbf{p}] \quad \text{with} \quad \mathbf{v}^\tau = [(v_x^0, v_y^0, z_z^0), (v_x^1, v_y^1, z_z^1), \dots], \quad \mathbf{p} = [p^0, p^1, \dots],$$

where the superscript $\tau \in \{v, e, f, c\}$ denotes the entity type (vertex, edge, face, cell). This index mapping approach enables the generation of a wide range of different storage patterns for all kinds of models. The flexibility also allows to some extent the ability to increase cache reuse.

4.1.4. On-the-fly operators are supported in DUNE to avoid storing the full sparse matrix. Linear solvers such as the conjugate gradient method can be written in this way, because they work

[§]Picture from `http://www.dune-project.org/modules/dune-foamgrid/`

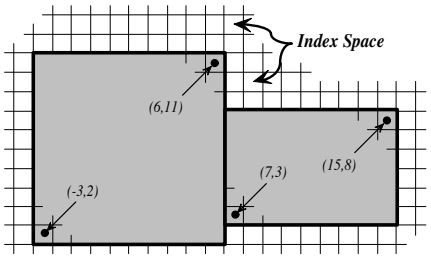

Figure 9. A grid consisting of two boxes in *hypre*'s structured interface.

purely on an operator-based interface, that is, they only require the operation $A\mathbf{u}$, but not the individual entries of the matrix. Such an operator interface can be implemented completely *on the fly*, using the element-matrix representation, discussed in Section 2.2. The same holds true for simple preconditioners like the Jacobi preconditioner or for geometric multigrid, where the prolongation and restriction operators can also be represented as the sum over local matrices (see 2.2). Depending on the discretization it might require additional local information, like the vertex-cell relation, or the knowledge of the diagonal elements, which can be pre-computed and stored cheaply. In these cases, the DUNE/PDELab assembler performs a grid loop and immediately evaluates the application of the operator and not the matrix entries. This is similar to what is done in classic stencil codes, but instead of a classical stencil, it relies on the concept of element contributions. The advantage of this approach is that it also works for unstructured grids and advanced numerical schemes, but it comes at the price of an increase in overhead. This overhead has basically two sources: 1) the grid loop can be relatively expensive, in particular in the case of fully unstructured grids; 2) in the general case the local matrix operator cannot make use of a-priori knowledge, like constant coefficients, or affine geometries, and thus the entries must be reevaluated in each cell. The amount of overhead depends strongly on the particular scheme and on assumptions put into the implementation. Higher-order methods reduce the overhead, by increasing the arithmetic intensity.

4.2. Implementation in hypre

The *hypre* library offers a set of optimized matrix formats, which try to avoid storing repetitive information. For example, on structured meshes the matrix pattern does not need to be stored, and for operators that are generated from constant stencils such as the 9-point stencil (13) or the 5-point stencil (14) the row entries are known a priori. Therefore *hypre* offers several interfaces: a structured, a semi-structured, a finite element, and a linear-algebraic interface. Since the focus of this paper is stencils, we only consider the structured and semi-structured interfaces, which are based on stencils. In this section we describe the implementation of matrices and vectors in the structured and semi-structured interfaces and some planned extensions to them.

4.2.1. The Structured Interface is based on grids and stencils. The grids in the structured interface are defined on an index space. The index space consists of a collection of d-tuples for a space of dimension d, e.g., doubles (i, j) in 2D. The StructGrid is a collection of non-overlapping boxes defined on the index space. A box is a collection of cell-centered indices defined by its lower corner, i.e., the d-tuple with the smallest values, and its upper corner, i.e., the d-tuple with the largest values. Figure 9 illustrates a grid.

A StructStencil is a collection of indices, representing a relative offset from some point in the grid. For example a standard 2D 5-point stencil would be represented as

$$\begin{bmatrix} & (0,1) & \\ (-1,0) & (0,0) & (1,0) \\ & (0,-1) & \end{bmatrix}.$$

The matrix in the structured interface, the StructMatrix, is defined by a grid and a stencil. The data structure of the StructMatrix contains information on the grid and the stencil in addition to the

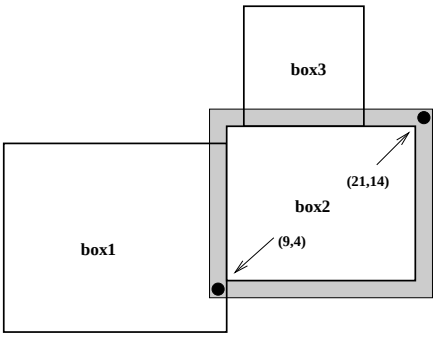

Figure 10. A box with ghost layer connected to two boxes in *hypre*'s structured interface.

data, which is an array of doubles defining the coupling coefficients of the matrix. The data structure of the corresponding vector, the StructVector, is similar except it does not have a stencil.

For both the vector and the matrix, all values of the data array associated with a given box of the grid are stored contiguously. In the case of the matrix where the data contains the values for different stencil entries, the data for a particular stencil entry are also stored contiguously. From a matrix point of view, this is equivalent to storing the diagonals (or a part of a diagonal if the grid consists of more than one box) of a matrix contiguously. For both a vector and a matrix, the boxes for the data array may be larger than the actual grid boxes. This allows the data to have ghost cells for better efficiency and to avoid the need for special stencils at the box boundaries. Some of these ghost cells can overlap with other grid boxes on the same processor or a neighboring processor. Updates in the ghost cells therefore require either copying data from other boxes or communicating the data from a different processor. We do not discuss the parallel aspects further, since this is not the focus of the paper. Figure 10 shows a box with its ghost layer connected to two other boxes.

Note that for certain problems and/or computer architectures it might make sense to change the data storage. For example, interleaving stencil entries could possibly improve cache use for certain problems. We do not investigate this further here since this goes beyond the scope of the paper.

In the current *hypre* implementation, the StructMatrix has only one grid, which is used both as range and domain, forcing the matrix to be square. However, work is in progress to redefine the StructMatrix data structure to allow for different domain and range grids, and hence allow for rectangular matrices as well. Rectangular matrices are important operators for multigrid solvers, which require restriction and interpolation operators to move between increasingly coarser grids. In the context of multigrid, an interpolation operator has a domain grid that is generally a coarsened version of the range grid, and vice versa for a restriction operator. To support this, the new StructMatrix data structure also contains strides, i.e., d-tuples that define the coarsening factor for each dimension. For example, a stride of $(2, 1)$ on a two-dimensional grid of size $n \times n$ would generate a grid of size $n/2 \times n$. There is another important type of rectangular matrix that occurs in the context of the semi-structured interface and it is covered in Section 4.2.2.

4.2.2. The Semi-Structured Interface allows the use of more complex grids, including block structured grids (illustrated in Figure 7), overset grids, and adaptive mesh refinement grids (as shown in Figure 11). These grids are generated by gluing together various structured parts using a graph and suitable stencils at the interfaces. While the structured interface only considered the cell-centered variable type, here it is possible to define additional variable types such as nodal, edge centered, and face centered. Figure 3 illustrates cell centered (circles), nodal (squares), x-face and y-face centered (triangles) variables. The index (i, j) is used to reference the variables in black. The grey variables are referenced by other indices. Figure 7 illustrates several stencils between different variable types and Figure 12 illustrates how they are represented in *hypre*.

The semi-structured grid is composed of a number of structured grid parts each with their own index spaces. Each part consists of boxes and variables. The unknowns in the linear system are characterized by their part number, their variable type, and an index that identifies the cell on the

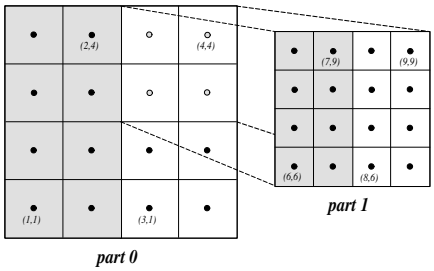

Figure 11. An adaptive mesh refinement grid in *hypre*'s semi-structured interface.

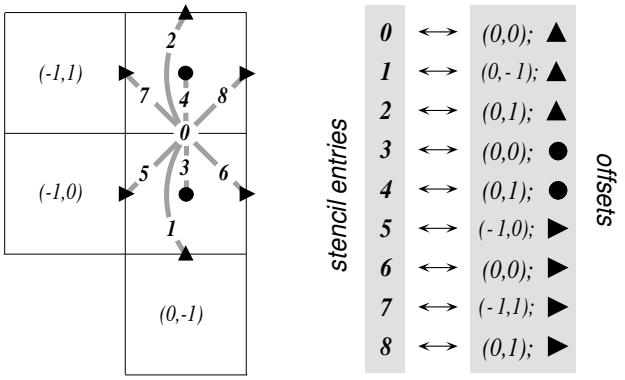

Figure 12. Assignments of labels and geometries to the y-face stencils in Figure 7.

part. The non-zero pattern of the semi-structured matrix, the SStructMatrix, is described through a graph, which consists of two types of couplings: stencil and non-stencil. The stencil couplings are described by stencils similar to the structured interface. The non-stencil couplings define specific couplings between particular unknowns, e.g., couplings between different parts, but also couplings within a part that only belong to specific cells and do not belong to the stencil. The SStructMatrix can then be defined as a sum of a structured matrix part S and an unstructured part U:

$$A = S + U. \tag{20}$$

The U matrix is completely unstructured and stored as a ParCSRMatrix, which is a parallel sparse matrix format based on the well known serial CSR format. The S matrix consists of a collection of StructMatrices. Currently, only stencil couplings between the same variable types are stored in S, whereas couplings between different variable types are included in U. Hence, for an example case of a single part with three different variable types, the SStructMatrix currently has the form

$$\begin{pmatrix} S_{11} & U_{12} & U_{13} \\ U_{21} & S_{22} & U_{23} \\ U_{31} & U_{32} & S_{33} \end{pmatrix}. \tag{21}$$

Even though the U_{ij} blocks are actually structured matrices, they cannot be expressed in terms of StructMatrices, since they are rectangular and the structured interface currently only allows square structured matrices, as mentioned in Section 4.2.1. The SStructVector also consists of an unstructured and a structured part, but is of course less complex.

To get a better understanding of the structure of U_{ij} one needs to take a look at the grids one obtains for different variable types in the same part and how they relate to a cell-centered grid. Figure 13 shows a box consisting of 3×3 cells with cell-centered, nodal, x-face and y-face centered variables. Since the structured grid is based on cell-centered variables and since we would like to use already existing functionality, a box for a new variable type is converted to a cell-centered box,

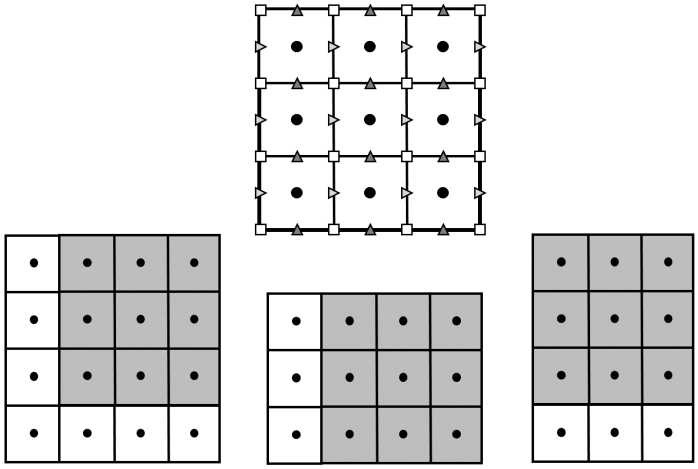

Figure 13. Converting a box (top) to cell-centered type boxes for nodal (bottom left), y-face (bottom center) and x-face (bottom right) variables.

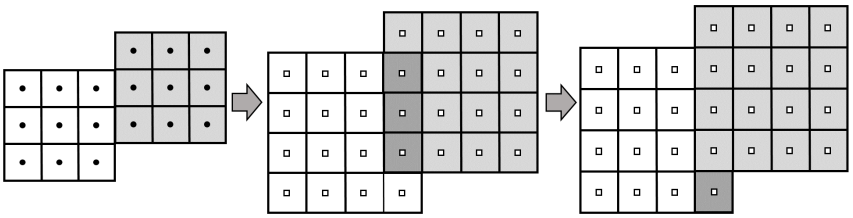

Figure 14. Converting a cell-centered grid with two boxes to a "nodal" cell-centered grid.

leading to a larger box as illustrated in the lower half of Figure 13. The grey shaded part shows how the original cell-centered box relates to the new box.

While this conversion is fairly simple for one box, it becomes more complicated for a collection of boxes. Figure 14 shows the conversion from a more complex cell-centered grid with two boxes to a nodal grid. Using the approach demonstrated for one box and applying it to the individual boxes leads to two overlapping boxes as illustrated in the center of Figure 14. Since overlapping boxes are not allowed, the grid needs to be redefined to a grid without overlap. Employing the approach used in *hypre* to deal with this situation leads to a grid with three boxes as depicted on the right of Figure 14. Generally it is necessary to redefine grids in this way when building matrices S_{ii} that correspond to non-cell centered variable types.

Future plans for *hypre* include replacing the matrices U_{ij} by structured matrices S_{ij}. As already mentioned in Section 4.2.1 this requires both a range and a domain grid. If we consider the grid with two boxes in Figure 14 and assume that i is cell-centered and j nodal, the range grid would be the left grid in Figure 14 and the domain grid the right grid. If one overlays the nodal grid with the cell-centered grid (boxes with thick black dashed lines) as illustrated in Figure 15, it becomes clear that these boxes do not line up with each other. This makes it very difficult to define matrix operations so they can be implemented in an efficient way. To avoid this difficulty boxes for both grids need to correspond to each other. While it is possible to deal with a box of one variable type that is embedded in a corresponding box of another variable type, a box is not allowed to overlap several boxes as is the case for the left cell-centered box in Figure 15. To avoid this issue, it would be necessary to generate a new set of boxes that fulfills these conditions, which here would lead

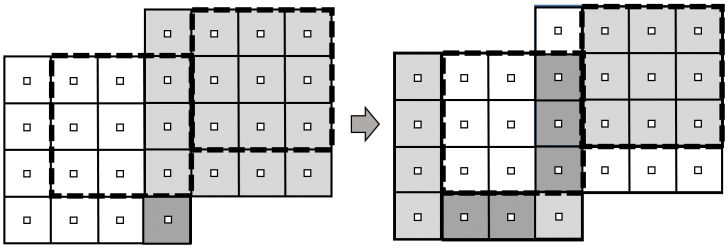

Figure 15. Demonstrating the difficulty to convert a more complex grid to a different variable type and matching up boxes.

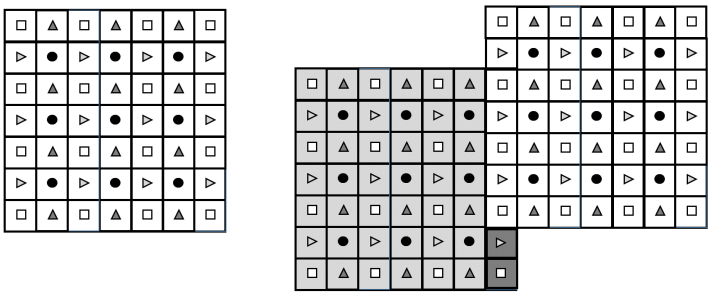

Figure 16. Finer grid with various variable types.

to eight boxes from the original two (see Figure 14). This approach becomes increasingly more complicated for grids with more boxes and variable types.

A better way to deal with this situation is to convert the original grid into a "cell-centered" grid of about double the size in each direction that contains all variable types as illustrated in Figure 16, and then access the individual variable types using strides. In that situation, one could use the current strategy to eliminate overlapping boxes and would get significantly fewer boxes overall. In the example with two boxes, this would produce only three boxes for all four variable types as in Figure 16, compared to eight boxes for two variable types in the previous approach. Implementing this approach in *hypre* is an area of future work.

5. CONCLUSIONS

In the context of performance optimization for stencil-based computations, the so-called Jacobi iteration has been studied extensively and helped to produce many useful optimization techniques. In contrast, scientific simulation codes generally involve more complex computational patterns, yet in many cases these patterns still exhibit structure that can potentially be exploited. In this paper, we discuss the use of stencils in more general settings and give examples of how they are used in PDE applications. In particular, we describe the relationship between stencils, matrices, solvers, and discretizations of PDEs. We also discuss data layout approaches for these problems and give examples of how stencil-based problems are managed in *hypre* and DUNE. The goal is to help create a bridge between the computer science and computational mathematics communities by providing information that enables new research in code optimization techniques that will benefit PDE applications that use stencils.

REFERENCES

1. P. BASTIAN, M. BLATT, A. DEDNER, C. ENGWER, R. KLÖFKORN, R. KORNHUBER, M. OHLBERGER, AND O. SANDER, *A generic grid interface for parallel and adaptive scientific computing. Part II: Implementation and tests in DUNE*, Computing, 82 (2008), pp. 121–138.
2. P. BASTIAN, M. BLATT, A. DEDNER, C. ENGWER, R. KLÖFKORN, M. OHLBERGER, AND O. SANDER, *A generic grid interface for parallel and adaptive scientific computing. Part I: Abstract framework*, Computing, 82 (2008), pp. 103–119.
3. B. BERGEN, T. GRADL, F. HÜLSEMANN, AND U. RÜDE, *A massively parallel multigrid method for finite elements*, Computing in Science and Engineering, 8 (2006), pp. 56–62. Special issue on Multigrid Computing.
4. *BLAS (Basic Linear Algebra Subprograms)*. http://www.netlib.org/blas/.
5. F. BREZZI AND M. FORTIN, *Mixed and hybrid finite element methods*, vol. 15 of Springer Series in Computational Mathematics, Springer Science & Business Media, 2012.
6. W. BRIGGS, V. HENSON, AND S. MCCORMICK, *A Multigrid Tutorial*, SIAM, Philadelphia, 2000. Second edition.
7. K. DATTA, M. MURPHY, V. VOLKOV, S. WILLIAMS, J. CARTER, L. OLIKER, D. PATTERSON, J. SHALF, AND K. YELICK, *Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures*, in Supercomputing 2008, Austin, TX, November 2008.
8. C. C. DOUGLAS, J. HU, M. KOWARSCHIK, U. RÜDE, AND C. WEISS, *Cache optimization for structured and unstructured grid multigrid*, Electronic Transactions on Numerical Analysis, 10 (2000), pp. 21–40.
9. B. GMEINER, T. GRADL, H. KÖSTLER, AND U. RÜDE, *Highly parallel geometric multigrid algorithm for hierarchical hybrid grids*, in NIC Symposium 2012, K. Binder, G. Münster, and M. Kremer, eds., no. 45 in NIC Series, FZ Jülich, 2012, pp. 323–330.
10. B. GMEINER, H. KÖSTLER, M. STÜRMER, AND U. RÜDE, *Parallel multigrid on hierarchical hybrid grids: a performance study on current high performance computing clusters*, Concurrency and Computation: Practice and Experience, 26 (2014), pp. 217–240.
11. *hypre: High performance preconditioners*. http://www.llnl.gov/CASC/hypre/.
12. M. KREUTZER, G. HAGER, G. WELLEIN, H. FEHSKE, AND A. R. BISHOP, *A unified sparse matrix data format for modern processors with wide SIMD units*, SIAM Journal on Scientific Computing, 36 (2014), pp. C401–C423.
13. M. KRONBICHLER AND K. KORMANN, *A generic interface for parallel cell-based finite element operator application*, Computers & Fluids, 63 (2012), pp. 135–147.
14. L. LAMPORT, *The parallel execution of do loops*, Communications of the ACM, 17 (1974), pp. 83–93.
15. T. MALAS, G. HAGER, H. LTAIEF, H. STENGEL, G. WELLEIN, AND D. KEYES, *Multicore-optimized wavefront diamond blocking for optimizing stencil updates*, SIAM Journal on Scientific Computing, 37 (2015), pp. C439–C464.
16. J. MOREL, R. M. ROBERTS, AND M. J. SHASHKOV, *A local support-operators diffusion discretization scheme for quadrilateral r-z meshes*, Journal of Computational Physics, 144 (1998), pp. 17–51.
17. Y. SAAD, *Iterative methods for sparse linear systems*, SIAM, 2003.
18. C. TAYLOR AND P. HOOD, *A numerical solution of the Navier-Stokes equations using the finite element technique*, Computers & Fluids, 1 (1973), pp. 73–100.