A calculus for stencils on arbitrary grids with applications to parallel PDE solution

Guntram Berti[1]

Abstract

The local data dependency pattern or stencil of a numerical algorithm is a structural property which is important for parallel computations. We present an algebraic notation for stencils on unstructured grids, derive some basic properties of stencils, and introduce two algorithms for constructing grid overlaps based on stencils. Finally, we show how these results lead to a more general and reusable approach to parallel PDE solution.

1 Introduction and Motivation

For a numerical mathematician, discretization algorithms are of interest mostly from the point of view of their approximation properties. Here, we look at such algorithms from a different angle: We investigate some of their *structural* aspects, using techniques from combinatorics and topology.

The motivation for this approach is found in the parallel numerical solution of partial differential equations. In order to meet the demand for computational power, using affordable parallel hardware like PC clusters has become a viable option for PDE solver software.

Although the parallelization of such software in general follows the conceptually simple paradigm of geometric data partitioning, there are serious technical obstacles to obtaining correct and efficient parallel software. The difficulties stem in part from the need of duplicating information in order to ensure efficient data access.

The exact extent of the duplication or overlap depends on the data dependency patterns (stencils) of the numerical algorithms used, and may vary considerably among different applications. Whereas this problem has found sufficient coverage in the case of structured grids [4, 5], the case of unstructured grids still lacked a general solution. Exploiting the common structure of discretization algorithm, we present a simple algebraic description of their

[1]Institut für Mathematik, BTU Cottbus, Germany

stencils, which can be used to create the necessary overlap automatically and efficiently.

The layout of this paper is as follows: To set the stage, we first introduce some aspects of parallel PDE solution. In section 3, we introduce a notation for stencils on arbitrary grids, and present some fundamental properties of stencils. Section 4 deals with a sequential and a distributed algorithm for calculating grid overlaps depending on a stencil. Finally, we discuss some issues related to a general and reusable implementation of these algorithms in section 5.

2 Some Aspects of Distributed PDE solution

The paradigm of choice for parallel PDE solution is geometric data partitioning or domain decomposition[2]: The computational domain, represented by a grid, is partitioned into parts of approximately equal size and distributed accordingly. To ensure data locality, each part has to be enlarged by a grid overlap. As the grid data on the overlap is duplicated on several parts, special action has to be taken to reestablish consistency of copies at well-defined moments.

Most of the technical details can be handled by data structures specialized to represent distributed overlapping grids. When constructing such a distributed overlapping grid, we start, at least conceptually, with a single global grid $\widehat{\mathcal{G}}$, which is partitioned by some grid partitioning algorithm, see e. g. [6]. Appropriate overlaps have to be determined and added to the local parts, leading to *local overlapping grids.*

In each local grid, the grid elements (like vertices or cells) can be partitioned into disjoint ranges, classified as *private*, *exposed* (to other parts), *shared* (with other parts), and *copied* (from other parts). This allows to restrict the computation to the logically owned part of the grid, and to overlap computation and communication (*compute-and-send-ahead*). Also, the synchronization of grid data can be implemented in a general manner on top of these overlap ranges, see [1] for details.

The crucial part — which served as motivation and starting point for the development of the stencil calculus — is the determination of the overlap, because information on (and *only* on) the *structure* of the concrete numer-

[2]This is meant in a general sense, not Schwarz DD.

```
for all Cells c ∈ G^d do
  flux(c) = 0;
  for all Neighbor cells c' of c do
    flux(c) += numflux(c,c',U)
```

(a) The skeleton of a first-order Finite Volume algorithm

```
for all Vertices v ∈ G^0 do
  avg(v) = 0, vol = 0
  for all cells c incident to v do
    avg(v) += volume(c)*U(c)
    vol += volume(c)
  avg(v) = avg(v)/vol
```

(b) Averaging cell values on vertices

Figure 1: Two simple algorithms, with stencils (C, F, C) and (V, C)

ical algorithms must be used. So, higher-order approximation algorithms typically need a larger overlap than low-order ones do.

3 Stencils of Numerical Algorithms

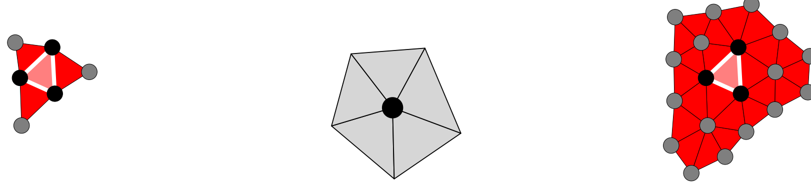
Let us look at some archetypical examples of numerical discretizations for PDEs to get an idea of how their stencils can be described in a simple, yet exact way.

The archetype of a first-order Finite Volume scheme is shown in figure 1(a). Its data dependency pattern can be described as going from cells (C) over incident facets (F) to incident cells (C), written compactly as (C, F, C); or $(d, d-1, d)$, where d is the dimension of the grid.

A more complicated example is shown in figure 1(b). Here, the state of the solution is averaged on vertices, using the state of incident cells. Hence, the stencil is (V, C), or $(0, d)$. If this algorithm is used together with a recovery approach, replacing the flux calculation in figure 1(a), the combined stencil becomes (C, F, C, V, C), or $(d, d-1, d, 0, d)$.

Definition 1 (incidence sequence). An *incidence sequence* for a d-dimensional grid is a sequence I of numbers $(a_0, \ldots, a_n)$ with $0 \leq a_i \leq d$ and $a_i \neq a_{i+1}$. The sequence I is *cell-based*, if $I = (d, d_1, d, \ldots, d_k, d)$, where d is the dimension of the grid. (All stencils shown here are cell-based.)

Now, let us assume we are given an algorithm A with stencil described by a sequence I, and a partition P of a grid $\mathcal{G}$. The overlap needed by A when



(a) Stencil (C, F, C) (b) Stencil (V, C) (c) Stencil (C, F, C, V, C)

Figure 2: Stencil 2(c) is composed by chaining stencils 2(a) and 2(b) and results when combining algorithms 1(a) and 1(b)

executed on P is the set of all cells (and elements of lower dimension incident to them) which can be reached from the cells of P (the *germs*) "following the sequence I":

Definition 2 (incidence layers and hulls). Let $\mathcal{K} \subset \mathcal{G}$ be an initial set (*germ*) of elements of dimension i. The *incidence layer* $\mathcal{L}_{(i,j)}(\mathcal{K})$ is defined by

$$\mathcal{L}_{(i,j)}(\mathcal{K}) := \bigcup_{e \in \mathcal{K}^i} \mathcal{I}_j(e) = \bigcup_{e \in \mathcal{K}^i} \{f \in \mathcal{G}^j \mid f \text{ incident to } e\} \tag{1}$$

Let $I = (a_0, a_1, \ldots, a_n)$ be an incidence sequence over a grid $\mathcal{G}$. The *hull* $\mathcal{H} = \mathcal{H}_I(\mathcal{K})$ and the layers $\mathcal{L}_I^{(k)}(\mathcal{K})$ generated by I and a germ $\mathcal{K} \subset G^{a_0}$ are then defined recursively by

$$\mathcal{L}_I^{(0)}(\mathcal{K}) := \mathcal{K} \tag{2}$$

$$\mathcal{L}_I^{(k)}(\mathcal{K}) := \mathcal{L}_{(a_{k-1}, a_k)}(\mathcal{L}_I^{(k-1)}(\mathcal{K})) \setminus \bigcup_{j=0}^{k-1} (\mathcal{L}_I^{(k-1)}(\mathcal{K})) \tag{3}$$

$$\mathcal{H}_I(\mathcal{K}) := \bigcup_{k=0}^{n} \mathcal{L}_I^{(k)}(\mathcal{K}) \tag{4}$$

See figure 4 for an example. For sake of simplicity, we will henceforth only consider cell-based sequences.

It is clear that the stencil $(d, 0, d)$ will generate a larger hull than $(d, d - 1, d)$. Also, $(d, 0, d, 0, d)$ generates a larger hull than $(d, 0, d)$. We may express

this fact by setting $(d,0,d,0,d) > (d,0,d) > (d,d-1,d)$. Generalizing this ordering, we arrive at a partial order for stencils:

Definition 3 (partial order on stencils). A stencil $I = (d, d_1, d, \ldots, d_n, d)$ *dominates* a stencil $J = (d, c_1, d, \ldots, c_k, d)$, in symbols $I \geq J$, if there exists $1 \leq i_1 < \ldots < i_k \leq n$ such that

$$c_r \geq d_{i_r} \quad \text{for} \quad 1 \leq r \leq k$$

The expected result then is that a greater stencil generates a larger hull. This holds indeed for sufficiently well-behaved grids, but is false in general:

Theorem 1 (Hull monotonicity of cell-based stencils). Let

$$I = (d, d_1, d, d_2, \ldots, d_n, d)$$

be a cell-based stencil, and $\mathcal{G}$ a manifold (with boundary) grid. Then

(i) For all cell sets $A, B \subset \mathcal{G}^d$

$$A \subset B \Rightarrow \mathcal{H}_I(A) \subset \mathcal{H}_I(B) \qquad \text{(germ monotonicity)} \tag{5}$$

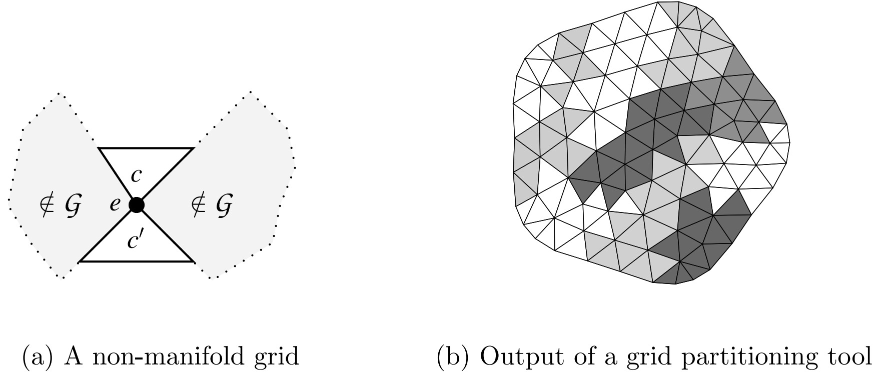
(ii) If J is another cell-based stencil, then

$$J \leq I \Rightarrow \mathcal{H}_J(A) \subseteq \mathcal{H}_I(A) \qquad \text{(stencil monotonicity)} \tag{6}$$

This result allows to compare stencils of different algorithms, and to select a common super-stencil for overlap generation. At first glance, non-manifold grids like in figure 3(a) appear not to be relevant in the context of numerical PDE solution. Yet it is often necessary to restrict hull operations to some subgrid. If such a subgrid is a partition produced by a standard partitioning tool (cf. fig. 3(b)), non-manifold situations can in general not be excluded.

4 Algorithms for Calculating Stencil Hulls

The main purpose for introducing an algebraic notation for stencils on unstructured grids was to enable automatic calculation of the hulls generated by stencils. This section presents two algorithms for doing so: The first one works in a sequential setting and operates on a germ set knowing the whole



(a) A non-manifold grid (b) Output of a grid partitioning tool

Figure 3: Non-manifold grid parts may occur in practice.

(global) grid. The second one is a distributed version, working on the local part of a distributed grid, initially knowing only about how shared elements relate to corresponding elements of direct (i.e. linked by a common facet) neighbors. We will see that the average case complexity of both algorithms is optimal, provided some decent conditions are satisfied by the grid, and the by partitioning in the distributed case.

The algorithms are *dimension independent*, achieved by judicious reference to codimension instead of to dimension whenever appropriate. Both algorithms require that the grid representation supports the necessary incidence queries: If the stencil contains for instance a sub-section like $(0, d)$, there must be a possibility to get all cells incident to a vertex. Also, in order to achieve the complexity results mentioned above, we assume that in the example the time required to access all cells incident to a vertex is proportional to the number of those cells.

The formulation of the sequential hull calculation algorithm INCIDENCE-HULL is given in table 4. In line 1, it marks all grid elements as non-visited in constant time. Also, in line 8 and 9, the visited-state of an element is queried and changed. Here too, we require complexity $O(1)$ (for the average case). These requirements can be satisfied by taking a hash-table for the visited map. If instead we take a balanced tree, we get worst and average case complexity $O(\log n)$ in lines 8 and 9, where n is the number of already visited elements, that is, $n = O(|\mathcal{H}|)$. The following complexity result assumes a hash-table like data structure.

```
IN: stencil I = (d_0, ..., d_n)
IN: germ 𝒦
OUT: levels ℒ_I^(0)(𝒦), ..., ℒ_I^(n)(𝒦)
 1: visited ←false   (by default: O(1))
 2: ℒ_I^(0) = 𝒦
 3: for all e ∈ ℒ_I^(0) do
 4:   visited(e) ←true
 5: for k = 1, ..., n do
 6:   for all e ∈ ℒ_I^(k−1) do
 7:     for all f incident to e, dim(f) = d_k do
 8:       if not visited(f_e) then
 9:         visited(f_e) ←true
10:         ℒ_I^(k) ←{f_e} ∪ ℒ_I^(k)
```

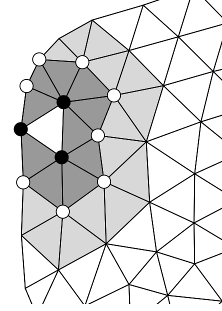


Figure 4: Right: The IncidenceHull algorithm, left: Layers of (C, V, C, V, C) seq.

Theorem 2. (Complexity of IncidenceHull) Let, for each element e of a grid $\mathcal{G}$, the number of elements of any dimension incident to e be bounded by a constant M. Let I be a stencil, and $K \subset \mathcal{G}$ be a germ set. Then the algorithm IncidenceHull calculates the hull $\mathcal{H} = \mathcal{H}_I(K) \subset \mathcal{G}$ in expected time $O(|\mathcal{H}|)$.

The constant M in the theorem is for practical grids reasonably small; for two-dimensional grids, the average number of cells incident to a vertex is 6.

The algorithm DistributedHull for the distributed case is given in table 5. Referring to the picture, we see the local part $\mathcal{G}_i^{(0)}$ is all the algorithm knows initially (plus the correspondences $\mathcal{S}_{ij}$ of boundary elements to those of the four direct neighbors). In the next step, partial hulls are copied from those direct neighbors. In the second step, also information from indirect neighbors (i. e. those sharing only a vertex with $\mathcal{G}_i^{(0)}$) is obtained via the direct neighbors.

In this simple case, we are finished in step 2. However, for more complex stencils and unstructured grids, we could need more steps:

IN: Distributed grid $\mathcal{G}_i^{(0)} = P_i, 1 \le i \le n$
IN: boundary correspondence
$\mathcal{S}_{ij} : \partial P_i \mapsto \partial P_j, 1 \le i, j \le n$
$k \leftarrow 0$
repeat
for all parts P_i **do** *(parallel)*
for all direct neighbors P_j of P_i **do**
calculate hull $\mathcal{H}_{ij} = \mathcal{H}_I(\mathcal{S}_{ij})$ on $\mathcal{G}_i^{(k)}$
transfer all new elements of $\mathcal{H}_{ij}$ to P_j
for all direct neighbors P_j of P_i **do**
get new elements of $\mathcal{H}_{ji}$ from P_j
for all $e \in \mathcal{H}_{ji}$ **do**
if e is still unknown **then**
$l \leftarrow$ owner of e, $\mathcal{C}_{il} \leftarrow \{e\} \cup \mathcal{C}_{il}$
$\mathcal{G}_i^{(k+1)} \leftarrow \mathcal{G}_i^{(k)} \cup \{$new elements$\}$
$k \leftarrow k + 1$
until no new elements have been added

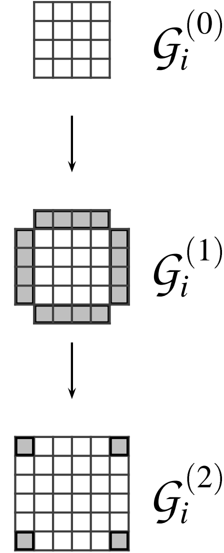


Figure 5: TheDistributedHull algorithm

Theorem 3. Algorithm DistributedHull obtains the copied ranges $\mathcal{C}_{ik}$ for a stencil $I = (d, d_1, \ldots, d_r, d)$ in at most

$$\sum_{j=1}^{r} \max_{\mathfrak{e} \in \mathfrak{G}^{d_j}} \operatorname{diam}(\operatorname{st}(\mathfrak{e}))$$

steps. Here $\mathfrak{G}$ is the quotient grid, and with $\operatorname{diam}(\operatorname{st}(\mathfrak{e}))$ we mean the maximal distance of two cells in $\operatorname{st}(\mathfrak{e})$, when passing to another cell is allowed only through a facet.

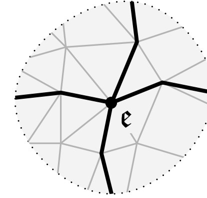


Figure 6: Quotient grid around $\mathfrak{e}$, $\operatorname{diam}(\operatorname{st}(\mathfrak{e})) = 2$

Typically, the value of $\operatorname{diam}(\operatorname{st}(\mathfrak{e}))$ is bounded by a small number (2 for 2D Cartesian quotient, 1 for a typical vertex of a generic (unstructured) partitioning). For a fixed stencil length, the average case complexity of hull calculation is again $O(\mathcal{H})$.

5 Implementation Issues & Discussion

In order to use the concepts presented before in an application, it might seem that a lot of coding effort on top of the given grid data structures is necessary. To avoid repeated work for each application, it would be ideal to provide an implementation which is independent of the underlying grid data structures. A framework allowing exactly this independence has been developed in [1], using a paradigm known as *generic programming* [8]. Algorithms and other components — such as distributed grids and overlap ranges — are based generically on an interface (kernel) describing grid functionality in an abstract way, and do *not* refer directly to the concrete grid representation itself. Thus, the implementation can be reused across a wide variety of grid representations, after an initial effort of creating a layer implementing the kernel interface.

The actual realization of this approach uses the C++ template features and follows the lines of the C++ Standard Template Library [7, 9]. Therefore, the resulting code is quite efficient and in some cases achieves performance equal to direct (non-generic) implementations. Software is available at [2].

The concepts presented have been successfully applied to parallelize applications for the solution of the instationary Euler equations and the stationary incompressible Navier-Stokes equations, using explicit and implicit schemes, respectively. Another parallelization effort is under way.

For the parallelization of a given numerical application, One will first create the abstract interface layer as mentioned above. Next, one determines the stencils of the algorithms used by the application. The generation of distributed overlapping grids can be left completely to the generic components discussed above.

The program structures needing some adaptation are, above all, global loops and global reduction operations. These typically bear a data-parallel meaning analogous to "for all cells of the grid do ... ", and have to be restricted to local ranges. The calculated grid data have to be synchronized afterwards, an operation completely hidden by the underlying distributed overlapping grids.

There has been much research work devoted to frameworks for parallel computing in the context of structured grids, see e. g. [4, 5]. The situation is different for unstructured grids, where no comparable work is known to the author. A approach based on distributed graphs has been presented in [3];

however, due to its generality, it cannot offer the same level of support and efficiency for grid-based applications.

Distributed PDE solution has been the starting point for this work. Another potential application are cache-sensitive implementation which also need to know the exact data dependencies of algorithms. Also, some of the ad-hoc notations found in the literature could be unified, eliminating the need for complicated (and potentially ambiguous) drawings.

References

[1] Berti, G. *Generic software components for Scientific Computing*. PhD thesis, BTU Cottbus, Germany, 2000.

[2] Berti, G. GrAL – Grid Algorithms Library. http://www.math.tu-cottbus.de/~berti/gral, 2001.

[3] Birken, K. Semi-automatic parallelisation of dynamic, graph-based applications. In D'Hollander, E., Joubert, G., Peters, F., and Trottenberg, U., editors, *Parallel Computing: Fundamentals, Applications and New Directions*. Elsevier Science, 1998.

[4] Brown, D. L., Chesshire, G. S., Henshaw, W. D., and Quinlan, D. J. OVERTURE: An object-oriented software system for solving partial differential equations in serial and parallel environments. In *8th SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis*, Mar. 1997.

[5] Cummings, J. C., Crotinger, J. A., Haney, S. W., Humphrey, W. F., Karmesin, S. R., Reynders, J. V., Smith, S. A., and Williams, T. J. Rapid application development and enhanced code interoperabily using the POOMA framework. In Henderson, M. E., Anderson, C. R., and Lyons, S. L., editors, *Proc. of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*. SIAM, Oct. 1998.

[6] Karypis, G. METIS home page. http://www-users.cs.umn.edu/~karypis/metis/, 1999.

[7] Lee, M. and Stepanov, A. A. The standard template library. Technical report, Hewlett-Packard Laboratories, Feb. 1995.

[8] Musser, D. R. and Stepanov, A. A. Generic programming. In Gianni, P., editor, *Symbolic and algebraic computation: International Symposium ISSAC '88, Rome, Italy, July 4–8, 1988: proceedings*, number 358 in LNCS, pages 13–25. Springer, 1989.

[9] SGI Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl.