Concepts for parallel numerical solution of PDEs

Guntram Berti

Institut für Mathematik, BTU Cottbus
Universitätsplatz 3-4
03044 Cottbus, Germany
`berti@math.tu-cottbus.de`

ABSTRACT *The parallelization of numerical codes on unstructured grids still is a demanding programming task, which is often performed in an application-specific manner. However, many of the algorithms used in this field exhibit strong structural similarities. A concept exploiting the latter is developed in this paper. Furthermore, it will be shown how the abstract concepts can be implemented in a reusable generic way, thus easing the actual work to be done for building parallel PDE software.*

Keywords: parallel scientific computing, distributed grid data structures

1. Introduction

For applications in the field of numerical PDE solution, the increased computational power of parallel hardware offers a promising way of surpassing limitations of resources. However, parallelization of numerical code can be a demanding task, especially when performed from scratch.

Taking advantage of existing tools for parallel computing is therefore a sensible option. For applications operating on regularly structured data (e. g. Cartesian grids) a good deal of tools are available. For unstructured grids, on the other hand, there is still a lack of sufficient tool support, see [Bir98] for a different approach and an overview.

We propose a generic, data-structure based solution, which makes only minor modifications to existing code necessary. This is no automatic parallelization concept; the act of data exchange remains under the control of the application program.

Our approach emphasizes *logical* grid distribution; it is applicable for distributed as well as for shared memory, and for multi-block type grids.

The organization of the paper is as follows: First, a short overview over the strategy of geometric partitioning is given. Then the concepts of distributed overlapping grids are introduced, and some aspects of their implementation are discussed. Finally, we review some implementation-related issues.

2. Geometric Partitioning for parallel PDE solution

Algorithms for grid-based numerical PDE solution typically work locally on data with a predictable pattern of access, with work roughly proportional to grid size. Many of these algorithms do not depend on the order grid elements are processed (this is not true e. g. for some linear solvers used for convection-dominated problems).

Therefore, *geometric partitioning*, or *domain decomposition*, is an adequate parallelization strategy [Zom96]: Equal-sized parts of the computational grid are assigned to different processes. These parts have an overlap which acts like a cache in order to avoid direct access to remote data; its size depends on the *client algorithms* used by the application. Computations in parallel (*owner-computes* rule) alternate with synchronizing the locally copied information in the overlap with the neighbor processes.

In order to implement domain decomposition, several concrete problems have to be solved. A given grid has to be partitioned into equal parts; an appropriate overlap has to be derived from the algorithms and to be added to these parts. We must maintain some correspondence relation between grid regions duplicated in the overlap, and provide a means to ensure consistency of duplicated data on the grid.

These question have to be tackled in the basic case of *static* distribution. If the grid distribution is *dynamic*, there are the additional difficulties of incremental load re-balancing and migration of grid-related data structures.

We concentrate here on the static case, because it is required for every application. Also, it covers most of the parallel application-level code as well as the actual communications load, so it is the key to both parallel performance and ease-of-use. Migrating a distributed grid data structure definitely is not a trivial task, but can be made largely transparent to the application-level code. This problem will be discussed in [Ber99].

We do not treat *partitioning* of grids in this paper. It is a well-investigated problem ([Els97]) and is considered a preprocessing step here.

3. Concepts for distributed grids

Manageable data structures for distributed grids are the key to parallel implementations of PDE solvers. They should both hide any detail of no interest to the application programmer and at the same time provide the high-level constructs needed to express the logical nature of data distribution. In particular, a simple yet expressive way of dealing with the logical parts of grid overlap according to their ownership semantics proves essential.

The following three sections are concerned respectively with the inner structure of the overlap, maintaining consistency of data *on* the grid (distributed grid functions), and finally the construction of the overlap itself, driven by the *stencils* of client algorithms.

3.1. *Distributed overlapping grids — DOGs*

Our point of departure is a simple 'sequential' grid, henceforth called *local grid*. For parallelization of existing code, it is crucial that the representation of this grid does not need to be changed. On top of this local grid sits an *overlapping grid*, which manages overlap-related information. It also has access to a coarse *partition grid*, the cells of which correspond to the local grids, and establishes the link to neighboring local grids.

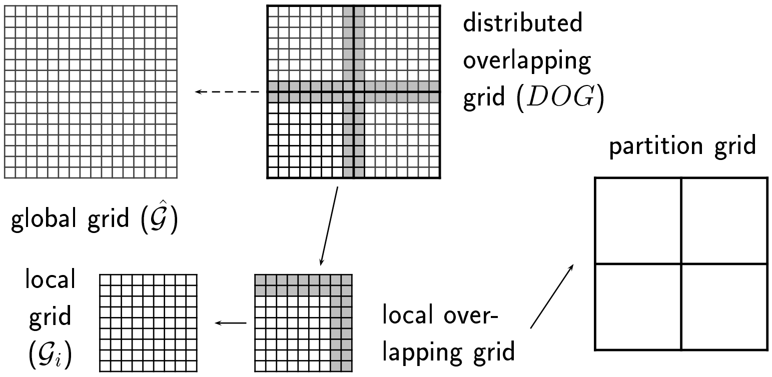


Figure 1: components of a distributed grid

Thus far, the grid entities have a local meaning only. Now, we add as an additional layer a *distributed grid*, which stands for the ensemble of all local grids. If there is single global memory, there needs to be only one such entity that contains the array of local overlapping grids. In the case of a physically distributed environment, this layer is represented by a distributed grid for each physical memory.

The distinction between overlapping and distributed grid is essential for the meaning of global reduction operations. It also leads to a better factorization of functionality, such that an overlapping grid implementation can be reused for many distribution scenarios.

The concepts of overlap structure will be introduced here in an informal manner; the reader interested in a formal treatment is referred to [Ber99].

As mentioned above, each local grid $\mathcal{G}_i$ contains a certain overlap $\mathcal{O}_i$ which can be thought of as a 'halo' surrounding it. The local grid parts are tied together by means of identification mappings between their overlaps, defining the global grid $\hat{\mathcal{G}}$ as the union of the $\mathcal{G}_i$ with the proper identifications.

On each local overlap, we impose *ownership relations* by distinguishing between parts $\mathcal{E}_{ij}$ that *belong* to $\mathcal{G}_i$ (and are copied in $\mathcal{G}_j$), ranges $\mathcal{S}_{ij}$ that are *shared* between $\mathcal{G}_i$ and $\mathcal{G}_j$, and parts $\mathcal{C}_{ij}$ that are *copied* from $\mathcal{G}_j$, cf. fig. 2.

These *bilateral ranges* support the efficient data-exchange among neighbors. They are complemented by *total ranges* $\mathcal{E}_i, \mathcal{S}_i$ and $\mathcal{C}_i$ given by union over all neighbors j. These are central for guiding the parallel execution of algorithms. For completeness sake, we mention also *private* ranges $\mathcal{P}_i$ not duplicated elsewhere, *imported* ranges $(\mathcal{S}_i \cup \mathcal{C}_i)$ and *exported* ranges $\mathcal{E}_i \cup \mathcal{S}_i$.

3.2. *Distributed grid functions – DGFs*

Static communication patterns exchange only data *on* the grid, a notion captured by the concept of *distributed grid functions*. As before, we distinguish between local ('sequential'), overlapping, distributed and global grid functions, corresponding to the type of grid they are thought to 'live on'.

For a proper definition of the notions of consistency, an overlapping grid function F has to be considered jointly with an algorithm $\mathfrak{F}$ responsible for calculating its values, a *write-range* W where F is locally computed, and a *read-range* R, where F is locally accessed.

What about the class of algorithms $\mathfrak{F}$ we consider here? First, according to the SPMD paradigm, $\mathfrak{F}$ is thought to be a function on the *global* grid computing a global grid function $\hat{F}$. This can only be achieved by computing locally the F_i if $\mathfrak{F}$ is *data-parallel*: It does not depend on the order of computation, i. e. not on the values of F itself. If this not the case (e. g. Gauss-Seidel iteration), we obtain another algorithm $\mathfrak{F}^P$ which is a block-version of the original $\mathfrak{F}$. This may or may not be tolerable, and must be decided by the numerical scientist.

Assuming now $\mathfrak{F}$ is 'sufficiently data-parallel', we call a tuple $(F, \mathfrak{F}, W, R)$ *locally up-to-date* if $F(e) = \mathfrak{F}(e)$ for all $e \in W$; it is *locally consistent* if also $F(e) = \mathfrak{F}(\tilde{e})$ for all $e \in R$ with remotely owned or shared counterparts $\tilde{e}$. A distributed grid function $(F_i, \mathfrak{F}, W_i, R_i)_{1 \leq i \leq N}$ is said to be globally consistent if each $(F_i, \mathfrak{F}, W_i, R_i)$ is locally consistent. Then the global grid function $\hat{F}$ is well defined and (ideally) identical to the outcome on a global sequential grid. The act called *synchronization* of establishing global consistency is a central service offered by a distributed grid function.

If *DGF*s are used for a parallel program, we have first to decide about appropriate ranges W and R. In most cases, $W = \mathcal{P}_i \cup \mathcal{E}_i \cup \mathcal{S}_i$ is adequate; the range R depends on the set of algorithms using F, so taking R equal to the whole local grid is always safe but may be wasteful.

Second, we have to control *when* a synchronization should occur. Again, this is safe when all local computations are done; we can overlap computation and communication (*compute-and-send-ahead*) by first calculating on exported ranges, initializing data exchange, calculating the rest, and finishing data exchange.

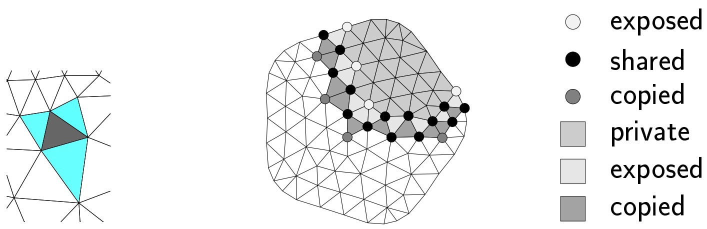
The task performed by *DGFs* may be a bit more involved. So it is sometimes useful to calculate results only partially on the shared parts of W, e. g. the assemblation of FEM stiffness matrices. Then synchronization means *reduction* (typically addition) on shared ranges. It may also be useful to work with data that is only locally up-to-date, and it depends on the sequence of algebraic operations, when a consistent state has to be established, see [Bas94].

3.3. *Incidence-based stencils and overlap generation*

The efficiency of the approach strongly depends on the size of the *local support* or *stencil* of $\mathfrak{F}$, which is for each grid element e the minimal neighborhood $U(e)$ such that the evaluation of $\mathfrak{F}(e)$ uses only information defined on $U(e)$.

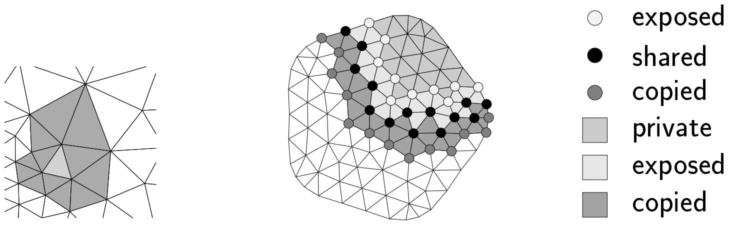
Two questions now arise: how can the stencils of algorithms on unstructured grids be described, and how can one produce the correct overlap from that and a partitioned grid?

Attacking the first problem, we confine ourselves to stencils expressible in terms of the grids *incidence relations*. To formalize this idea, we introduce *incidence sequences* $I = (d_0, \ldots, d_n)$ with $0 \leq d_i \leq d = \dim(\mathcal{G})$, meaning "go from elements of dim. d_0 to incident elements of dim. d_1 to ... ". Given a set A (or *germ*) of elements of dimension d_0, we may define the *hull* $\mathcal{H}_I(A)$ generated by I on A by taking the union of all elements visited by following the path given by I, starting from the elements of A.



(a) stencil $(d, d-1, d)$ (b) corresponding hulls

Figure 2: Simple stencil and overlap generated by MARK-HULL



(a) stencil $(d, 1, d)$ (b) corresponding hulls

Figure 3: Another stencil and overlap generated for one grid part

Consider the following simple flux calculation example:

for all Cells $c \in \mathcal{G}$ **do**
  for all Cells n adjacent to c **do**
    fluxsum(c) $\leftarrow$fluxsum(c) + flux(U(c),U(n))

The stencil of this algorithm is given by $(d, d-1, d)$, which means that for a given cell, each neighbor cell is accessed, via the corresponding facet, as depicted by figure 2(a). If some kind of second-order recovery is added, involving interpolation on vertices, a more complicated stencil results (figure 3).

Knowing this local stencil, it is now possible to derive an appropriate overlap, given a partitioned grid. In essence, this reduces to the calculation of the hull generated by the stencil, taking the boundary of the grid partitions as germs. An algorithm MARK-HULL using space and time proportional to the size of the generated hull has been developed and implemented, but cannot be reproduced here due to space limitations, see [Ber99] for a discussion. For the flux-algorithm stencil above, this results in an overlap as in figure 2(b).

In a typical application, there are several stencils involved, and thus different hulls may need to be computed. The total overlap of the grid is then composed by the union of partial overlaps imposed by different algorithms.

4. Generic implementation and practical experience

The concepts presented so far are useful in their own right. However, we would like to offer an implementation which is usable without change for a large variety of concrete grid data structures. In order to achieve such a grid-independent realization, it proves necessary to adopt a more abstract approach to grid data structures.

In [BB99] is shown how an abstract grid interface enables a generic programming style, producing implementations that do not rely on a particular grid representation. See [Ber99] for a detailed description of these concepts.

We just remark that our approach makes a clear separation between combinatorial grid, geometric embedding of the grid and grid functions that allow to store data on grid elements. This corresponds well to the requirements of the distributed grid data structures, as it mainly depends on the combinatorial properties, and in the case of a static distribution, grid functions are the main actors.

Most importantly, incidence information is captured by *iterators*, which essentially answer queries like "give me all vertices incident to this cell, one after another", making them a suitable basis for the MARK-HULL algorithm mentioned above.

On top of this grid abstraction, data structures for representing distributed grids, overlaps and grid ranges, as well as the algorithms for generating them from incidence-type stencils, have been developed in a fully generic fashion. Thus, the implementation may be directly used with any grid that satisfies the abstract interface.

In practice, it proves rather straightforward to adapt existing grid implementations to this interface, or more precisely, to provide a separate implementation for it. According to our experiences, this requires only one or two days of work, which could probably still further reduced. Once this has been done, the generic implementation of distributed grids can be used 'off the shelf', fig. 4. It remains to insert the appropriate calls to synchronization routines and global reduction operations. Most of the code may continue to use the 'native' grid data structures.

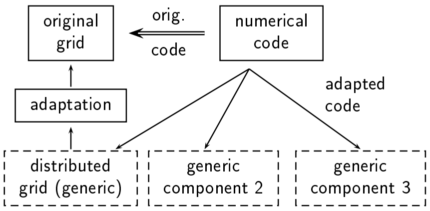


Figure 4: using generic components with existing codes

It should be noted, that as a result of this adaptation, not only the distributed grid components may be used, but all generic algorithms and data components developed on top of the interface are immediately available.

The techniques discussed so far have been successfully used to implement distributed solvers for the 2D Euler equations for one- and two-component systems. These solvers use a fluctuation-splitting algorithm on unstructured grids with a second-order recovery step, and an explicit time integration. The mathematics are described in [SBB97].

The implementation of this solver is based on a C++ -realization of the generic grid interface mentioned above. As such, the parallelization of the code essentially involved substitution of grid functions with their distributed counterparts, and addition of synchronization calls. Also, the maximal stencil of the algorithms used has been determined by simple pencil-and-paper inspection. Due to a parameterization of the solver over the actual grid type, the same numerical code can be used both for the parallel and the sequential version.

Currently, the parallelization of a 2D Navier-Stokes solver with implicit time integration and a two-level grid is underway. This code has been developed using its own grid data structure; thus an adaption to the generic interface had to be done.

5. Conclusion

We have presented a novel approach to the development of parallel applications involving grid-based algorithms of data-parallel type, such as used in solvers for partial differential equations. In particular, the concepts allow the programmer to concentrate on the abstract aspects of grid distribution. It offers automatic generation of grid overlap based on the stencil of the client algorithms, and is therefore directly applicable to a broad class of applications. Furthermore, a generic implementation of these concepts make it easily usable.

6. Bibliography

[Bas94] Bastian, P. *Parallele adaptive Mehrgitterverfahren.* PhD thesis, Universität Heidelberg, 1994.

[BB99] Bader, G. and Berti, G. Design principles of reusable software components for the numerical solution of PDE problems. In Hackbusch, W. and Wittum, G., editors, *Concepts of Numerical Software.* Vieweg Verlag, 1999. To appear.

[Ber99] Berti, G. *Generic software components for Scientific Computing.* PhD thesis, BTU Cottbus, 1999. In preparation.

[Bir98] Birken, K. *Ein Modell zur effizienten Parallelisierung von Algorithmen auf komplexen, dynamischen Datenstrukturen.* PhD thesis, Universität Stuttgart, October 1998.

[Els97] Elsner, U. Graph partitioning – a survey. Technical Report SFB393/97-27, Technische Universität Chemnitz, December 1997.

[SBB97] Schenk, K., Bader, G., and Berti, G. Analysis and approximation of multicomponent gas mixtures. In Feistauer, M., Kozel, K., and Rannacher, R., editors, *Proceedings of the 3rd Summer Conference Numerical Modelling in Continuum Mechanics*, Prague, 1997.

[Zom96] Zomaya, A. Y., editor. *Parallel and distributed computing handbook*, chapter Scientific Computation. McGraw Hill, 1996.