Generic Programming Support for Mesh Partitioning Based Parallelization

Guntram Berti

C&C Research Laboratories, NEC Europe Ltd.
Rathausallee 10, 53757 St. Augustin, Germany
berti@ccrl-nece.de

Abstract. We present a domain-specific, library based approach for parallelizing mesh-based data-parallel applications (like numerical PDE solution) using the domain partitioning paradigm. Concepts are presented to formalize the notions of domain and mesh partitioning. Generic programming is used to implement reusable software components that encapsulate the common core of domain partitioning and make it available to application programmers. A novel aspect of our approach is its independence of the underlying sequential structured or unstructured mesh data structures, which makes reuse of most pre-existing sequential application code possible.

1 Introduction

Despite continuously increasing power of computer hardware, parallel computing is — and will remain — the only viable option for many scientific simulation challenges. The proper use of parallel architectures poses a number of problems to the programmer, who in general has to be more explicit about the underlying hardware then he wishes. This constitutes a significant barrier to the construction of efficient parallel scientific applications.

Although research for general-purpose automatic parallelization support has been pursued for decades now, success has been limited. In our opinion, this stems from the (unavoidable) diversity of different parallelization patterns (see e.g. the work in [1]) which withstands a uniform treatment.

On the other hand, when we focus on a *specific* domain, like parallel PDE solution, or more generally, structured or unstructured mesh-based "data-parallel" applications, the situation changes. Analyzing typical parallel applications in this area, we can see that they all exhibit strong structural similarities, as they all use the same parallelization paradigm called domain partitioning.

Hence, there is hope of providing some general support for parallelizing such applications, and even to encapsulate the repetitive task entirely into reusable components, which is what we propose here.

The essential feature of our approach is that it is purely library based and works in a bottom-up way: The parallel application programmer gets supplied high-level constructs to express data distribution, but retains full control over parallel execution. No extra tools like special pre-processors or compilers are needed. Most of the existing code and data structures can be reused, the "parallel" enhancements are simply "wrapped around" it, with a very low memory overhead which is close to what a manual parallelization would need. This "wrapping" is enabled by leveraging generic programming techniques in C++, which allow to abstract from concrete data representation issues.

The reuse of original sequential data structures is a key difference to the top-down approach followed by parallel frameworks for PDE solution, like Overture [2] or AMROC [3], which concentrate on (hierarchies of) structured grids. Here, an application programmer has to reuse the (data) structures provided by the framework. This makes programming more comfortable for new developments, but also more restrictive.

The dynamic distributed data (DDD) library by Birken [4] in contrast has distributed graphs as underlying abstraction and is therefore more general than our approach. Lacking the more specialized notion of distributed meshes, it cannot directly offer specific support for them.

An approach related to our one is the GRIDS library [5], which uses scripts and a special pre-processor to generate parallel Fortran code. Other approaches centered on unstructured meshes like DIME [6] or PMO [7] are more like frameworks in that they require reuse of their native data structures.

The organization of this paper is as follows: First, we describe the domain partitioning paradigm (Sec. 2.1) and then general concepts for distributed meshes in a general way (Secs. 2.3 – 2.6). Next, their implementation using generic programming is discussed in Sec. 3. Finally, we discuss further research issues.

2 Concepts for Distributed Meshes

2.1 The Domain Partitioning Paradigm

Domain partitioning (also known as geometric partitioning) is the parallelization paradigm of choice for mesh-based applications for PDE solution, for instance using finite elements, finite volume or finite differences. It exploits the fact that the data dependencies of these spatial discretizations (*stencils*) are merely local. The computational domain (i.e. the mesh representing it) is divided into pieces of roughly the same size, and the discretization algorithms are applied to each piece in parallel. In order to guarantee efficient access to the needed information (*data locality*), some data has to be replicated on different processes by adding an overlap region (or halo) to each mesh part. Data associated to the overlap mesh region must be updated regularly in order to provide a consistent view of the global state. In general, it is most efficient to perform these updates altogether to minimize communication costs.

The domain partitioning approach implicitly assumes *data-parallel* algorithms, in which case it can produce results identical with the sequential case (except effects due to different orders of floating point operations). While the assumption of data-parallelity is often satisfied, for instance finite element stiffness matrix assembly or finite volume flux balance computation, it is violated in some important cases, most notably the solution of linear equation systems which introduce a global data dependency. In this case, an application of the paradigm produces a modified algorithm, which may or may not be appropriate (Jacobian iteration is about the only broadly used data-parallel algorithm for solving linear systems, which explains its low efficiency). More often, one uses algorithms specialized to domain partitioning, like interface iteration using Schur complements [8]. Nonetheless, the concepts like overlap, ownership and generalized stencils we are going to develop are useful for this more general situation, too.

Now, while the general ideas are common to virtually all implementations of parallel PDE solution, individual applications differ in the details:

– How large is the overlap?
– Which data has to be exchanged when?
– How is the data transferred?
– What kind of data structures are used for grids and grid functions?

In the following sections, we will see how to formalize these intuitive concepts. Further on, we will show how generic components can parameterize the associated choices, in order to provide the application programmer with exactly those high-level abstractions needed for implementing data-parallel algorithms on arbitrary grids. As the parallelization is driven by mesh distribution, distributed data structures for grids and grid functions are the essential components which can encapsulate most of the technical details.

In this paper, we will not discuss issues related to the partitioning itself, see e.g. [9] for a survey.

2.2 Some Grid-related Terminology

In order to understand the following sections, we some introduce some terminology. By a (combinatorial) grid (or equivalently, mesh) $\mathcal{G}$ of dimension d we understand a tuple $(\mathcal{G}^0, \dots, \mathcal{G}^d)$ of element ranges of dimensions $k = 0$ (vertices) to $k = d$ (cells), together with an incidence partial ordering. Two elements e_1, e_2 are *incident*, if e_1 one is contained in the topological closure of e_2, that is, e_1 is on the boundary of e_2.

A grid function F on $\mathcal{G}$ is a mapping $F : \mathcal{G}^k \mapsto \mathcal{T}$ for some set $\mathcal{T}$. Grid functions (often called fields) are a means of storing data on grid elements, and play a crucial role in parallelization as this data has to be kept in sync on different parts.

2.3 Distributed Overlapping Grids

We can take two complementary views on distributed grids (Fig. 1). The *global point of view* sees a global grid, partitioned into local *parts* (with overlap). We use the term parts here to emphasize the independence of the concepts of the physical distribution aspects. The *local point of view* starts from local parts, equipped with correspondence maps between overlaps of neighboring parts. This correspondence gives rise to a (virtual) global grid. Thus, the local view in general corresponds more closely to the actual physical situation, where the global grid does not actually exist. It represents also a good compromise between a more idealized global view and the actual physical distribution, which leads to a simple yet powerful and efficient programming model.

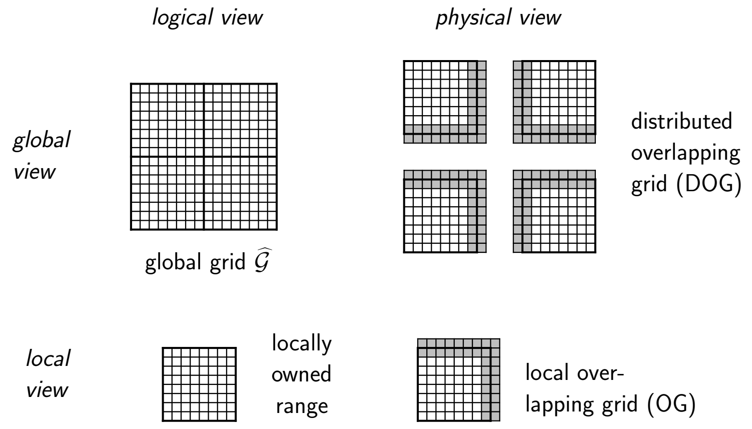
logical view
physical view
global view
global grid $\widehat{\mathcal{G}}$
distributed overlapping grid (DOG)
local view
locally owned range
local overlapping grid (OG)

Fig. 1. Components for representing a distributed grid. The global grid $\widehat{\mathcal{G}}$ is in general not physically represented. The local grid $\mathcal{G}_i$ underlies the overlapping grid OG.

The basic data structure of each local part is a *local grid* $\mathcal{G}_i$, represented by an arbitrary sequential grid data structure typically provided by the sequential application, which contains an appropriate amount of overlap $\mathcal{O}_i$. A *local overlapping grid* divides this overlap into a system of subranges, making them available to the application programmer, thus allowing him to control the scope of local computation (see Figs. 16 and 18). The top layer, *distributed overlapping grids*, complements local overlapping grids with system of correspondence mappings Φ between overlap portions of neighboring parts.

More formally, let $\mathcal{G}_i, 1 \leq i \leq N$ be a family of local grids. An *overlap structure* on $(\mathcal{G}_i)_{1 \leq i \leq N}$ is a system of grid isomorphisms on *bilateral overlaps* $\mathcal{O}_{ij}$

$$\Phi_{ij} : \mathcal{O}_{ij} \subset \mathcal{G}_i \mapsto \mathcal{O}_{ji} \subset \mathcal{G}_j$$

satisfying

$$\Phi_{ij}(\mathcal{O}_{ij}) = \mathcal{O}_{ji}, \qquad \Phi_{ij}^{-1} = \Phi_{ji}, \qquad \text{(symmetry)} \tag{1}$$

and

$$\Phi_{ij} \circ \Phi_{jk} = \Phi_{ik} \qquad \text{on} \quad \mathcal{O}_{ij} \cap \Phi_{ji}(\mathcal{O}_{jk}) \qquad \text{(transitivity)} \tag{2}$$

for $1 \leq i, j, k \leq N$. The local overlap of part i is given by the union $\mathcal{O}_i = \bigcup_{j=1}^{N} \mathcal{O}_{ij}$.

The functions Φ_{ij} give rise to a *equivalence relation* between elements of the local grids, with $e_i \sim e_j$ iff $e_j = \Phi_{ij}(e_i)$. The equivalence classes $\hat{e}$ form the elements of the global grid $\widehat{\mathcal{G}}$, which can thus be reconstructed from the local grids and the correspondence Φ.

2.4 Ownership on Overlaps

The overlap $\mathcal{O}_i$ of part $\mathcal{G}_i$ is now further divided into subranges reflecting ownership: We distinguish between *exposed* ranges $\mathcal{E}_{ij}$, which are owned by part i, *shared* ranges $\mathcal{S}_{ij}$, which belong to both part i and part j, and *copied* ranges $\mathcal{C}_{ij}$, that belong to part j. The following symmetry relations must hold between these ranges:

$$\begin{aligned}\mathcal{E}_{ij} &= \Phi_{ji}(\mathcal{C}_{ji})\\ \mathcal{S}_{ij} &= \Phi_{ji}(\mathcal{S}_{ji})\\ \mathcal{C}_{ij} &= \Phi_{ji}(\mathcal{E}_{ji})\\ \mathcal{O}_{ij} &= \mathcal{E}_{ij} \cup \mathcal{S}_{ij} \cup \mathcal{C}_{ij}\end{aligned}$$

and $\mathcal{E}_{ij}, \mathcal{S}_{ij}, \mathcal{C}_{ij}$ are pairwise disjoint. See figure 4 for the general picture.

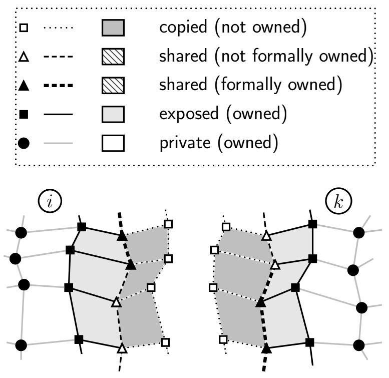

Fig. 2. Typical 1-cell overlap example, without shared cells.

For the classical Schwarz domain decomposition technique, there are no shared ranges, and the overlaps are disconnected (Fig. 6). For the so-called non-overlapping domain decomposition, we have only shared ranges (Fig. 5).

These bilateral ranges are useful primary for data exchange between different parts. For deciding where calculation has to take place, and where not, so-called *total ranges* are needed, which are roughly the unions of the bilateral ranges:

$$\mathcal{P}_i = \mathcal{G}_i \setminus \bigcup_{j=1}^{n} \mathcal{O}_{ij} \quad \textbf{p}\text{rivate}$$

$$\mathcal{C}_i = \bigcup_{j=1}^{n} \mathcal{C}_{ij} \quad \textbf{c}\text{opied}$$

$$\mathcal{S}_i = \bigcup_{j=1}^{n} \mathcal{S}_{ij} \setminus \mathcal{C}_i \quad \textbf{s}\text{hared}$$

$$\mathcal{E}_i = \bigcup_{j=1}^{n} \mathcal{E}_{ij} \setminus (\mathcal{S}_i \cup \mathcal{C}_i) \quad \textbf{e}\text{xported}$$

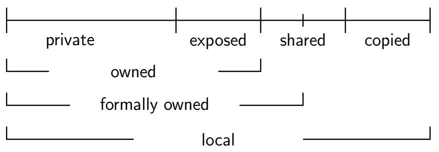

Fig. 3. Logically layered structure of overlap ranges

On exposed elements in $\mathcal{E}_i$, the local part is exclusively responsible for performing any calculation. On the shared range, several parts will in general contribute to the calculation, or calculations are done redundantly on each part. On copied elements, typically no local calculations are performed. Besides these elementary ranges, we often need a *formally owned* range $\mathcal{F}_i \subset \mathcal{P}_i \cup \mathcal{E}_i \cup \mathcal{S}_i$ (Fig. 3) which further split the shared elements in an arbitrary way, such that $\mathcal{F}_j \cap \Phi_{ij}(F_i) = \emptyset$ for $i \neq j$. This is useful e.g. for implementing global reduction operations.

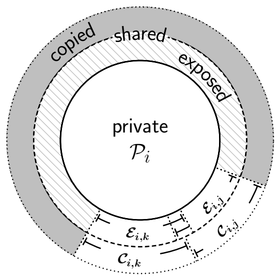

Fig. 4. Generic overlap configuration

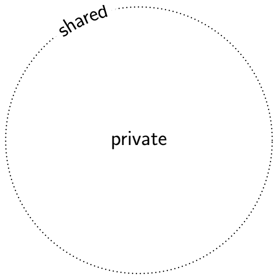

Fig. 5. Configuration for "non-overlapping" domain decomposition

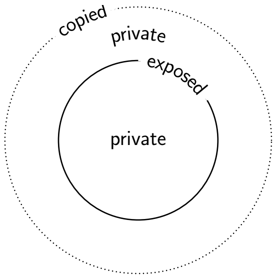

Fig. 6. Configuration for Schwarz domain decomposition

2.5 Distributed Overlapping Grid Functions

Data *on* the grid is given by *grid functions* in the sequential case, and by *distributed grid functions* (DGFs) in the distributed case. Because a DGF is multiply defined for overlap elements, we must define its *consistency state.*

With an overlapping grid function F (the local representative of a DGF), we associate a write range W a read range R, and a function f defining the algorithm used to define the values of F. Typical values for W is the owned range, and for R the local range (cf. Fig. 3).

We now assume f to be a function

$$f : \mathcal{G} \times S_f \mapsto \mathcal{T}$$

which depends on a state S_f of variables, for example other grid functions. If $F \notin S_f$, we call the loop

```
**for all** $e \in W$ **do**
    $F(e) \leftarrow f(e, S_f)$
```

a *data-parallel loop*, and f a data-parallel algorithm for F and S_f. For example, if F is a vector on grid vertices, and f is the local Jacobi relaxation on a vertex, then the loop is data parallel. In contrast, for the Gauss-Seidel relaxation, the loop is *not* data-parallel, the corrected values are immediately used.

It is now straightforward to define consistency on the tuple (F, f, W, R), given a distributed grid $(\mathcal{G}_i)_{1 \leq i \leq N}$ with correspondence relation Φ: F is *locally con-*

sistent on a range $L \subset W$, iff $F(e) = f(e) \quad \forall e \in L$. A distributed grid function $(F_i, f, W_i, R_i)_{1 \le i \le N}$ is *globally consistent*, iff each (F_i, f, W_i, R_i) is locally consistent on W_i and for each pair $e_i \in R_i, e_k \in R_k$ with $\hat{e}_i = \hat{e}_k$ we have $F_i(e_i) = F_k(e_k)$. In other words,

$$F_i = F_k \circ \Phi_{ik} \quad \text{on} \quad R_i \cap \mathcal{O}_{ik}$$

The action of making a DGF globally consistent is called *synchronization.*

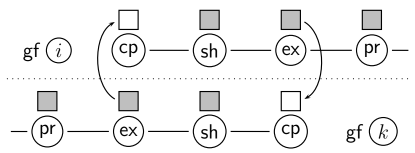

Fig. 7. Redundant calculation on shared ranges

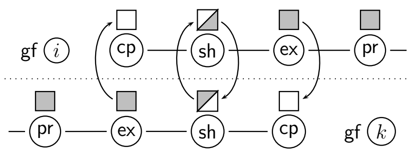

Fig. 8. Partial calculation on shared ranges

On shared elements, two different strategies of computation are possible: Either redundant (Fig. 7), in which case each part calculates its results on the shared range locally, or partial (Fig. 8), in which case the results from all sharing parts have to be combined afterwards. For example, when computing a finite element stiffness matrix in a non-overlapping configuration like in Fig. 5, the rows belonging to shared vertices have to be added if a consistent representation of the matrix is needed.

The concepts presented in this section are surprisingly powerful for implementing parallel grid-based algorithms. However, in order to provide a complete solution of the parallelization problem, there must be a means of generating the data structures like overlap ranges defining the distributed grid. This task is tackled in the following section.

2.6 Stencils and Overlap Generation

In finite difference terminology, the term *stencil* is used to denote the local data dependency of an algorithm (discretization), and typically described by (the non-zero pattern of) a matrix. If we want to generalize stencils to unstructured meshes, we must abandon the matrix notion.

Looking at the algorithm in figure 9, we see that this algorithm contains a global loop over all cells, and for each cell c, every neighbor cell is accessed, i.e. every cell sharing a facet with c. The stencil of this algorithm is shown graphically in figure 10. It can be described algebraically by the sequence $(d, d-1, d)$ (where d is the dimension of the mesh), meaning "start from grid elements of dimension

```
for all Cells $c \in \mathcal{G}^d$ do
  flux($c$) = 0;
  for all Neighbor cells $c'$ of $c$ do
    flux($c$) += numflux($c$,$c'$,U)
```

Fig. 9. A simple finite volume flux balance

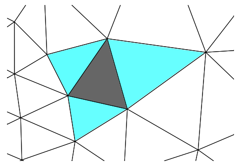

Fig. 10. its associated stencil $(2, 1, 2)$, or $(d, d-1, d)$ dimension-independently

d, go over incident elements of dimension $d-1$ (the facets) to incident elements of dimension d (neighbor cells)".

If we analyze each algorithm in a program this way, we arrive at a number of different stencils. These can be used to find the necessary overlap for a partitioned mesh.

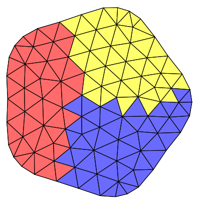

Fig. 11. A tiny mesh, partitionend into 3 parts.

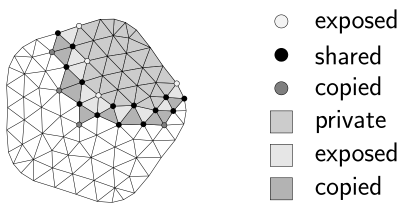

Fig. 12. Overlap for one part generated from the stencil of Fig. 10

3 Generic Components for Distributed Meshes

Our aim is to develop software components which implement the concepts presented before in a way which is as general as possible, while being easily adaptable to the specific application at hand, and inducing as few changes as possible in existing application code.

To achieve this aim, we extend the functionality of the sequential program layer by layer by the additional functionality needed for a parallel application.

A given sequential grid data structure forms the core layer. All local, application-dependent, grid-related algorithms continue to be based solely on this data structure and thus do not need to be changed. Any global, distribution-dependent functionality is provided by an additional layer – distributed grids. This layer itself is configurable in order to be adaptable to different physical distribution contexts and communication libraries.

This layering leads to a clean separation of algorithmic from distribution aspects, and is crucial for the a-posteriori parallelization of existing applications, where algorithmic code is based on a specific sequential grid data structure.

The technical basis for this data structure independent approach is given by a *generic programming* technique, which we discuss very briefly in the next section. The solution we are going present is based on C++ and is applicable directly only to applications also written in C++, however, providing interfaces to C and Fortran would be possible.

3.1 Generic programming

Generic programming aims at separating data representation details from higher level components like algorithm implementations. Thus, a new level of generality and reusability is achieved. One of the most prominent examples for generic programming is the C++ Standard Template Library (STL) [10] part of the C++ standard library, which deals with sequences (containers) and algorithms on sequences.

```
IN: $U : \mathcal{G}^d \mapsto \mathbb{R}^p$
OUT: avg : $\mathcal{G}^0 \mapsto \mathbb{R}^p$
  for all Vertices $v \in \mathcal{G}^0$ 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
```

Fig. 13. Algorithm for vertex averaging of cell values

```
grid_function<Cell,  state> U;          // IN
grid_function<Vertex,state> avg(grid,0);// OUT

for(VertexIterator v(grid); v; ++v) {
  double vol = 0;
  for(CellOnVertexIterator cv(*v); cv; ++cv) {
    avg[*v] += geom.volume(*cv) * U(*cv);
    vol_sum += geom.volume(*cv);
  }
  avg[*v] /= vol_sum;
}
```

Fig. 14. Generic implementation of the vertex averaging

An approach similar in spirit suitable for meshes has been investigated by the author within the open source Grid Algorithms Library GrAL [11]. With GrAL, it is possible to develop algorithms and derived data structures which are independent of the underlying mesh data structures, a feature which is crucial for implementing the concepts presented before in a truly reusable and general way.

The concepts of GrAL cannot be discussed here in detail, the interested reader is referred to [12] or [13]. Here, we restrict ourselves to an example. Looking at figure 14, we see a generic implementation using the GrAL interface of the algorithm of Fig. 13, which volume-averages cell values on vertices.

The generic class template `grid_function` allows to store any type of data on any grid element type (e.g. cell or vertex). With a `VertexIterator`, we can iterate through all vertices of a grid (or a subrange!); a `CellOnVertexIterator` allows to access all cells incident to a vertex.

We notice that this code is independent of the mesh dimension, type (structured/unstructured) and representation. It only requires the two iterator types mentioned before to be defined for the actual grid type. In a similar way, the data structures and algorithms for distributed grids discussed below can be implemented, and "wrapped around" a user supplied sequential grid data structure. To enable this wrapping, a user has to provide a description of his grid data structure in terms of a GrAL-compliant interface.

3.2 Components for Distributed Grids and Grid Functions

In the following, we discuss several generic software components which can be used to enhance a given sequential grid data structure with support for distributed programming. In order to keep the exposition short, we do not give syntactic details – the reader is referred to the reference implementation available in GrAL. A high-level view on the components relationship in an UML-like notation is given in Fig. 15.

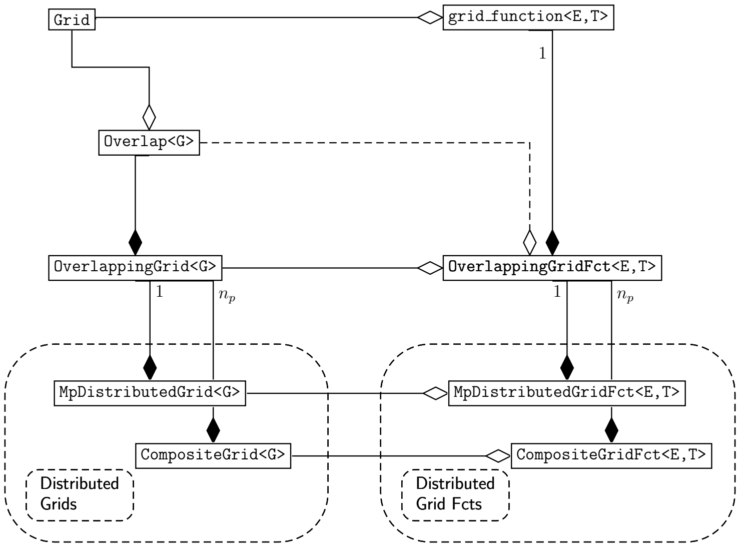

Fig. 15. Components for distributed overlapping grids and their relationships. Overlapping grid functions may contain their own overlaps (dashed line).

Overlap ranges A data structure for representing the overlap of a distributed grid must provide efficient access to any segment (copied, shared etc., see Fig. 3) of both total and bilateral ranges, for any grid element type needed (cells, edges, vertices etc.). As the segments can be combined by simple unions of adjacent elementary segments (i.e. private, exposed, shared, and copied, Fig. 3), it is straightforward to derive an efficient representation for both total and bilateral overlaps for each element dimension by using arrays of *element handles* (a sort of minimal element representation, typically an integer, defined in GrAL).

Using the overlap data structure (`Overlap<Grid>` in Fig. 15) combining these layered ranges, a programmer of a parallel application can then access total ranges, for instance all copied cells, all formally owned vertices etc., for constraining iteration in a parallel loop. Also, the distributed grid functions can access bilateral ranges for communication. Bilateral ranges are used to implicitely represent the correspondence mappings Φ_{ij} by a consistent ordering or their elements.

For the total overlap, it is in principle possible to manage with $O(1)$ memory overhead if we can sort the mesh elements in a way consistent with the elementary segments.

Overlap ranges are contained in overlapping grids, and optimized (i.e. smaller) overlap ranges may also be part of overlapping grid functions.

Distributed overlapping grids We distinguish between basic *overlapping grids* (OGs) with local semantics, and *distributed overlapping grids* (DOGs) with global semantics, representing the complete grid. Depending on the physical nature of distribution, there are different versions of distributed overlapping grids, for example message passing DOGs for distributed memory architectures, which typically contain a single OG, or composite grids, which are a set of OGs living in a single global memory. Such composite grids can be used for shared-memory parallelization using threads, to avoid cache conflicts and false sharing, and to test a parallel application sequentially.

Overlapping grids are used to factor out common functionality that is independent of the physical distribution. They consist of a user-supplied mesh data structure, enhanced with a GrAL-compliant interface, and an overlap range data structure like that described before.

Distributed overlapping grid functions For grid functions, the same distinction between local and global points of view as in the case of grids leads to *overlapping grid functions* (OGFs) on the one hand, and *distributed grid functions* (DGFs) on the other hand, where the latter group has further ramifications depending on the underlying DOG type, for instance message passing distributed grid function or composite grid function.

An overlapping grid function contains a user-defined, GrAL-compliant sequential grid function, has a reference to an overlapping grid, and optionally might contain an own overlap which is a subset of the overlap provided by the OG. Thus, an overlapping grid function which is used only on formally owned cells can avoid superfluous data communication.

A distributed grid function adds communication handlers to an OGF, which encapsulate the protocol for copying data, for instance using MPI or simple memory copy. It provides methods `begin_synchronize()` and `end_synchronize()` for bracketing the updating process, and thus allows for asynchronous communication and overlapping of calculation and communication.

A DGF is responsible for performing synchronizations in the right order for correctness, for instance, shared ranges must be updated before copied/exposed ranges if partial calculation is active.

Global reduction operations are defined on DGFs, parameterized by a binary reduction operator (for instance sum or maximum). These operation rely on formally owned ranges. In some cases, the binary reduction operator may require additional information, in which case the user may want to perform an explict local reduction by hand (cf. CFL-number computation in Fig. 17). A global reduction on the scalar values is then performed by a generic routine `global_reduce()`.

3.3 Overlap Generation

A cornerstone of the whole approach is the ability to generate the overlap with correct extent automatically, based on a stencil, and starting from a partitioned grid. For doing so, the concept of the *hull* generated by a stencil and a set of germ elements is crucial. Intuitively, the hull is defined by starting at the germ elements and successively adding incident elements according to the stencil. This is described in more detail in [14].

Overlaps can now be computed either from a partitioned global grid, or from an already distributed grid containing the correspondence information for shared elements. In general, a user only has to provide a partitioning and the stencil, all subsequent steps can be handled automatically.

3.4 Application Examples

Below, we present some simple but typical examples for parallel algorithms using the components presented before. In Fig. 16, we see the parallelized version of the sequential vertex avering of Fig. 13. We observe that the basic parallelized algorithm is almost identical to the sequential version, only the loop bounds and the final synchronization operation have to be added. A similar observation applies to the explicit reduction loop in Fig. 17.

If we want to optimize it by overlapping computation and communication, the straightforward solution is to duplicate the loop (Fig. 18), which is ugly. A better possibility is to use iteration over different ownership ranges, as indicated by figure 19 (not yet implemented in GrAL). Yet another possibility would be to implement a sort of `for_each_cell()` template function, which takes the loop body as an argument. This has the advantage of making parallelity more explicit in the interface, while allowing an arbitrary ordering of computation behind the scenes. While it is in principle possible to implement such a loop template in C++ by writing a specialized class executing the loop body (local algorithm), it turns out to be clumsy, and to require substantial changes to program organization. Unlike functional languages, C++ does not support closures or unnamed

```
IN: U: $\mathcal{G}^d \mapsto \mathbb{R}^p$
OUT: flux: $\mathcal{G}^d \mapsto \mathbb{R}^p$
  for all Cells $c \in$ owned$(\mathcal{G})^d$ do
    flux($c$) = 0;
    for all Neighbor cells $c'$ of $c$ do
      flux($c$) += numflux($c$,$c'$,U)
  synchronize(flux)
```

Fig. 16. Simple parallel version of Algorithm 9

```
IN: U: $\mathcal{G}^d \mapsto \mathbb{R}^p$
OUT: maxcfl: maximal CFL value
  for all Cells $c \in$ formally owned$(\mathcal{G})^d$ do
    maxcfl $\longleftarrow$max(maxcfl, cfl(c,U));
  maxcfl $\longleftarrow$global_reduce(maxcfl, max);
```

Fig. 17. Parallel reduction code: Explicit loop

"lambda functions". Approaches like the boost lambda library [15] are probably only of limited help here.

```
IN: U: $\mathcal{G}^d \mapsto \mathbb{R}^p$
OUT: flux: $\mathcal{G}^d \mapsto \mathbb{R}^p$
  for all Cells $c \in$ exposed$(\mathcal{G})^d$ do
    flux($c$) = 0;
    for all Neighbor cells $c'$ of $c$ do
      flux($c$) += numflux($c$,$c'$,U)
  begin_synchronize(flux)
  for all Cells $c \in$ private$(\mathcal{G})^d$ do
    flux($c$) = 0;
    for all Neighbor cells $c'$ of $c$ do
      flux($c$) += numflux($c$,$c'$,U)
  end_synchronize(flux)
```

Fig. 18. Parallel version of Algorithm 9, overlapping communication and computation. Note the unwanted duplication of the loop body.

```
IN: U: $\mathcal{G}^d \mapsto \mathbb{R}^p$
OUT: flux: $\mathcal{G}^d \mapsto \mathbb{R}^p$

  for all Rge. $R \in \{\mathsf{exp}(\mathcal{G})^d, \mathsf{prv}(\mathcal{G})^d\}$ do
    for all Cells $c \in R$ do
      flux($c$) = 0;
      for all Neighbor cells $c'$ of $c$ do
        flux($c$) += numflux($c$,$c'$,U)
    begin_synchronize(flux, R)


  end_synchronize(flux)
```

Fig. 19. Parallel version of Algorithm 9, using range iteration to overlap computation and communication. Here, only one loop body is needed.

The components described here have been used to develop a generic two-dimension solver for one- and two-components Euler solver which is described in [16]. This application uses the same generic code for sequential and parallel executables, and achieves typical parallel efficiency for explicit discretizations. Also, an application for solving the two-dimensional, incompressible, stationary Navier-Stokes equations using a SIMPLE pressure correction method has been parallelized *a posteriori*, i.e. using the original data structures. Only a few dozens of lines of the original code had to be changed, plus the creation of a GrAL adaptation layer for the original mesh data structure, which could be developed and tested completely separatly. The straight forward parallelization of the nested iteration of the SIMPLE algorithm worked correctly, but is not very efficient. Here, algorithms better suited to the distributed case have to be applied.

4 Conclusion

We have presented a set of concepts formalizing the notions of domain and mesh partitioning based parallelization. Building on that, we have shown how generic, reusable software components can encapsulate many of the related technical issues, thus making high-level parallel programming possible in the context of mesh-based, data-parallel applications. A key feature of these components is their non-intrusiveness: They allow to continue reusing existing sequential application code and data structures to a great extent, thus radically cutting down the parallelization work.

A necessary step for applying the generic components is the definition of a GrAL-complying interface for the mesh data structures of the sequential application. Albeit in principle not difficult, it can be a substantial barrier. GrAL already contains support for such adaptation, but it may be worthwhile to go a step further and create almost ready-made solutions for typical cases.

A useful enhancement to distributed grid functions would be to allow separate overlap ranges per grid functions, which could be organized in an overlap data base to save memory. Also, iteration over ownership ranges (Fig. 19) will be useful.

The concepts and components described here are for static mesh distribution. While for the concepts, nothing would change for dynamic mesh distribution, components for dynamic distributed grids would be highly useful. As for the automatic overlap generation, the complicated machinery could probably completely hidden behind a simple interface.

We plan to base the parallelization of an octree based mesh generator on top of the parallel components described here. The somewhat more irregular algorithmic patterns of a mesh generator may result in additions to the components, such as distributed partial grid functions.

References

1. Massingill, B.L., Mattson, T.G., Sander, B.A.: Patterns for parallel application programs. In: Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP 1999). (1999)
2. Brown, D.L., Chesshire, G.S., Henshaw, W.D., 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. (1997)
3. Deiterding, R.: AMROC – blockstructured Adaptive Mesh Refinement in object-oriented C++. `http://www.math.tu-cottbus.de/~deiter/amroc` (2002)
4. Birken, K.: Semi-automatic parallelisation of dynamic, graph-based applications. In D'Hollander, E., Joubert, G., Peters, F., Trottenberg, U., eds.: Parallel Computing: Fundamentals, Applications and New Directions, Elsevier Science (1998)
5. Geuder, U., Härdtner, M., Reuter, A., Wörner, B., Zink, R.: GRIDS — a parallel programming system for grid-based algorithms. The Computer Journal **36** (1993) 702–711

6. Fox, G.C., Williams, R.D., Messina, P.C.: Parallel Computing Works! Morgan Kaufmann Publishers (1994)
7. Terascale LLC: Parallel Mesh Object (PMO) home page (2002)
8. Kuznetsov, S., Lo, G.C., Saad, Y.: Parallel solution of general sparse linear systems. Technical Report UMSI 97/98, Minnesota Supercomputer Institute, University of Minnesota (1997)
9. Elsner, U.: Graph partitioning – a survey. Technical Report SFB393/97-27, Technische Universität Chemnitz (1997)
10. Lee, M., Stepanov, A.A.: The standard template library. Technical report, Hewlett-Packard Laboratories (1995)
11. Berti, G.: GrAL – the Grid Algorithms Library. `http://www.math.tu-cottbus.de/~berti/gral` (2001)
12. Berti, G.: GrAL – the Grid Algorithms Library. In Sloot, P.M.A., Tan, C.J.K., Dongarra, J.J., Hoekstra, A.G., eds.: Proceedings of ICCS 2002, part 3. Volume 2331 of LNCS., Springer (2002) 745–754
13. Berti, G.: Generic software components for Scientific Computing. PhD thesis, Faculty of mathematics, computer science, and natural science, BTU Cottbus, Germany (2000)
14. Berti, G.: A calculus for stencils on arbitrary grids with applications to parallel pde solution. In Sonar, T., Thomas, I., eds.: Proceedings of the GAMM Workshop on Discrete Modelling and discrete Algorithms in Continuum Mechanics, Logos Verlag Berlin (2001) 37–46
15. Järvi, J., Powell, G.: The lambda library: Lambda abstraction in C++. In: Proceedings Second Workshop on C++ Template Programming, Tampa Bay, OOPSLA2001 (2001)
16. Schenk, K., Bader, G., Berti, G.: Analysis and approximation of multicomponent gas mixtures. In Feistauer, M., Kozel, K., Rannacher, R., eds.: Proceedings of the 3rd Summer Conference Numerical Modelling in Continuum Mechanics, Prague (1997)