Overlapping Domain Decomposition Methods

X. Cai[1,2]

[1] Simula Research Laboratory
[2] Department of Informatics, University of Oslo

Abstract. Overlapping domain decomposition methods are efficient and flexible. It is also important that such methods are inherently suitable for parallel computing. In this chapter, we will first explain the mathematical formulation and algorithmic composition of the overlapping domain decomposition methods. Afterwards, we will focus on a generic implementation framework and its applications within Diffpack.

1 Introduction

The present chapter concerns a special class of numerical methods for solving partial differential equations (PDEs), where the methods of concern are based on a physical decomposition of a global solution domain. The global solution to a PDE is then sought by solving the smaller subdomain problems collaboratively and "patching together" the subdomain solutions. These numerical methods are therefore termed as *domain decomposition* (DD) methods. The DD methods have established themselves as very efficient PDE solution methods, see e.g. [3,8]. Although sequential DD methods already have superior efficiency compared with many other numerical methods, their most distinguished advantage is the straightforward applicability for parallel computing. Other advantages include easy handling of global solution domains of irregular shape and the possibility of using different numerical techniques in different subdomains, e.g., special treatment of singularities.

In particular, we will concentrate on one special group of DD methods, namely iterative DD methods using *overlapping* subdomains. The overlapping DD methods have a simple algorithmic structure, because there is no need to solve special interface problems between neighboring subdomain. This feature differs overlapping DD methods from non-overlapping DD methods, see e.g. [3,8]. Roughly speaking, overlapping DD methods operate by an iterative procedure, where the PDE is repeatedly solved within every subdomain. For each subdomain, the artificial internal boundary condition (see Section 2) is provided by its neighboring subdomains. The convergence of the solution on these internal boundaries ensures the convergence of the solution in the entire solution domain.

The rest of the chapter is organized as follows. We start with a brief mathematical description of the overlapping DD methods. The mathematical description is followed by a simple coding example. Then, we continue with some important issues and a discussion of the algorithmic structure of the overlapping DD methods. Thereafter, we present a generic framework

where the components of the overlapping DD methods are implemented as standardized and extensible C++ classes. We remark that Sections 2-6 do not specifically distinguish between sequential and parallel DD methods, because most of the parallel DD methods have the same mathematical and algorithmic structure as their sequential counterparts. We therefore postpone until Section 7 the presentation of some particular issues that are only applicable to parallel DD methods. Finally we provide some guidance on how to use the generic framework for easy and flexible implementation of overlapping DD methods in Diffpack.

2 The Mathematical Formulations

In this section, we introduce the mathematical formulations of the overlapping DD methods. The mathematics will be presented at a very brief level, so interested readers are referred to the existing literature, e.g. [3,8,10], for more detailed theories.

2.1 The Classical Alternating Schwarz Method

The basic mathematical idea of overlapping DD methods can be demonstrated by the very first DD method: the classical *alternating Schwarz method*, see [7]. This method was devised to solve a Poisson equation in a specially shaped domain $\Omega = \Omega_1 \cup \Omega_2$, i.e., the union of a circle and a rectangle, as depicted in Figure 1. More specifically, the boundary-value problem reads

$$\begin{aligned} -\nabla^2 u &= f \quad \text{in } \Omega = \Omega_1 \cup \Omega_2, \\ u &= g \quad \text{on } \partial\Omega. \end{aligned}$$

The part of the subdomain boundary $\partial\Omega_i$, which is not part of the global physical boundary $\partial\Omega$, is referred to as the artificial *internal boundary*. In Figure 1, we see that Γ_1 is the artificial internal boundary of subdomain Ω_1, and Γ_2 is the artificial internal boundary of subdomain Ω_2.

In order to utilize analytical solution methods for solving the Poisson equation on a circle and a rectangle separately, Schwarz proposed the following *iterative* procedure for finding the approximate solution in the entire domain Ω. Let u_i^n denote the approximate solution in subdomain Ω_i, and f_i denote the restriction of f to Ω_i. Starting with an initial guess u^0, we iterate for $n = 1, 2, \ldots$ to find better and better approximate solutions u^1, u^2, and so on. During each iteration, we first solve the Poisson equation restricted to the circle Ω_1, using the previous iteration's solution from Ω_2 on the artificial internal boundary Γ_1:

$$\begin{aligned} -\nabla^2 u_1^n &= f_1 \quad \text{in } \Omega_1, \\ u_1^n &= g \quad \text{on } \partial\Omega_1 \backslash \Gamma_1, \\ u_1^n &= u_2^{n-1}|_{\Gamma_1} \quad \text{on } \Gamma_1. \end{aligned}$$

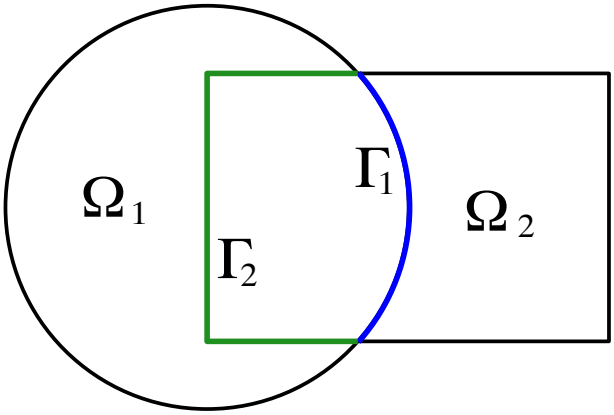

Fig. 1. Solution domain for the classical alternating Schwarz method.

Then, we solve the Poisson equation within the rectangle Ω_2, using the latest solution u_1^n on the artificial internal boundary Γ_2:

$$\begin{aligned} -\nabla^2 u_2^n &= f_2 \quad \text{in } \Omega_2, \\ u_2^n &= g \quad \text{on } \partial\Omega_2 \backslash \Gamma_2, \\ u_2^n &= u_1^n|_{\Gamma_2} \quad \text{on } \Gamma_2. \end{aligned}$$

The two local Poisson equations in Ω_1 and Ω_2 are coupled together in the following way: the artificial Dirichlet condition on the internal boundary Γ_1 of subdomain Ω_1 is provided by subdomain Ω_2 in form of $u_2^{n-1}|_{\Gamma_1}$, and vice versa. It is clear that $u_2^{n-1}|_{\Gamma_1}$ and $u_1^n|_{\Gamma_2}$ may change from iteration to iteration, while converging towards the true solution. Therefore, in each Schwarz iteration, the two Poisson equations need to update the artificial Dirichlet conditions on Γ_1 and Γ_2 by exchanging some data. Note also that the classical alternating Schwarz method is *sequential* by nature, meaning that the two Poisson solves within each iteration must be carried out in a predetermined sequence, first in Ω_1 then in Ω_2. Of course, the above alternating Schwarz method can equally well choose the rectangle as Ω_1 and the circle as Ω_2, without any noticeable effects on the convergence.

2.2 The Multiplicative Schwarz Method

We now extend the classical alternating Schwarz method to more than two subdomains. To this end, assume that we want to solve a linear elliptic PDE of the form:

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

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

where L is some linear operator.

In order to use a "divide-and-conquer" strategy, we decompose the global solution domain Ω into a set of P subdomains $\{\Omega_i\}_{i=1}^P$, such that $\Omega = \cup_{i=1}^P \Omega_i$. As before, we denote by Γ_i the internal boundary of subdomain number i, i.e., the part of $\partial\Omega_i$ not belonging to the physical global boundary $\partial\Omega$. In addition, we denote by $\mathcal{N}_i$ the index set of neighboring subdomains for subdomain number i, such that $j \in \mathcal{N}_i \Rightarrow \Omega_i \cap \Omega_j \neq \emptyset$. We require that there is explicit overlap between each pair of neighboring subdomains. (The case of non-overlapping DD methods is beyond the scope of this chapter.) In other words, every point on Γ_i must also lie in the *interior* of at least one neighboring subdomain Ω_j, $j \in \mathcal{N}_i$.

When the set of overlapping subdomains is ready, we run an iterative solution procedure that starts with an initial guess $\{u_i^0\}_{i=1}^P$. The work of iteration number n consists of P sub-steps that must be carried out *in sequence* $i = 1, 2, \ldots, P$. For sub-step number i, the work consists in solving the PDE restricted to subdomain Ω_i:

$$L_i u_i^n = f_i \quad \text{in } \Omega_i, \tag{3}$$

$$u_i^n = g \quad \text{on } \partial\Omega_i \backslash \Gamma_i, \tag{4}$$

$$u_i^n = \tilde{g}^* \quad \text{on } \Gamma_i. \tag{5}$$

Here, in a rigorous mathematical formulation, L_i in (3) means the restriction of L onto Ω_i. However, for most cases the L_i and L operators have exactly the same form. The right-hand side term f_i in (3) arises from restricting f onto Ω_i. The notation $\tilde{g}^*$ in (5) means an artificial Dirichlet condition on Γ_i. The artificial Dirichlet condition is updated by "receiving" the *latest* solution on Γ_i from the neighboring subdomains. Different points on Γ_i may use solution from different neighboring subdomains. More precisely, for every point x that lies on Γ_i, we suppose it is also an interior point to the neighboring subdomains: $\Omega_{j_1}, \Omega_{j_2}, \ldots, \Omega_{j_m}$, where $j_1 < j_2 < \ldots < j_m$ with $j_k \in \mathcal{N}_i$ for $k = 1, \ldots, m$. Then, the latest approximate solution at x from a particular subdomain, number $j_{\hat{k}}$, will be used. That is, $u_{j_{\hat{k}}}^n(x)$ is "received" as the artificial Dirichlet condition on x. Here, $j_{\hat{k}}$ should be the maximal index among those indices satisfying $j_k < i$. In case of $i < j_1$, $u_{j_m}^{n-1}(x)$ will be used as the artificial Dirichlet condition on x. The derived method is the so-called *multiplicative Schwarz method*.

Example: Multiplicative Schwarz on the Unit Square. Suppose we want to partition Ω, the unit square $(x, y) \in [0, 1] \times [0, 1]$, into four overlapping subdomains. This can be achieved by first dividing Ω into four non-overlapping subdomains, each covering one quarter of the unit square. This is shown by the dashed lines in Figure 2. Then, each subdomain extends itself in both x- and y-direction by Δ, resulting in four overlapping subdomains Ω_1, Ω_2, Ω_3, and Ω_4, bounded by the solid lines in Figure 2. The artificial internal boundary Γ_i of each subdomain can be divided into three parts, two of them border

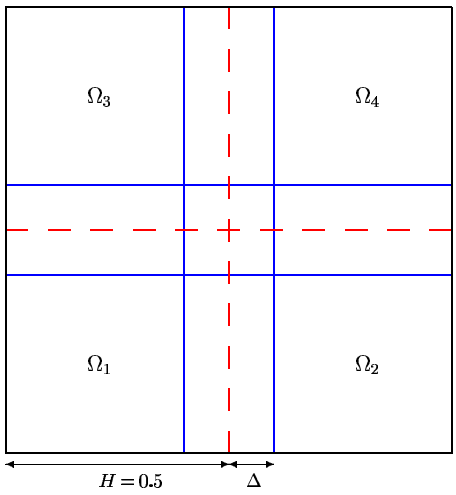

Fig. 2. An example of partitioning the unit square into four overlapping subdomains.

with only one neighboring subdomain, while the third part borders with all the three neighboring subdomains. Let us take Ω_1 for instance. The first part of Γ_1: $x = 0.5 + \Delta$, $0 \leq y \leq 0.5 - \Delta$ lies in the interior of Ω_2, so the artificial Dirichlet condition on this part is updated by "receiving" the latest solution from Ω_2. The second part of Γ_1: $0 \leq x \leq 0.5 - \Delta$, $y = 0.5 + \Delta$ lies in the interior of Ω_3. The artificial Dirichlet condition on this part is thus updated by "receiving" the latest solution from Ω_3. The remaining part of Γ_1, i.e., $0.5 - \Delta \leq x \leq 0.5 + \Delta$, $y = 0.5 + \Delta$, and $x = 0.5 + \Delta$, $0.5 - \Delta \leq y \leq 0.5 + \Delta$, lies in the interior of all the three neighbors. The artificial Dirichlet condition on this part, however, needs to be updated by "receiving" the latest solution from Ω_4. This is because subdomain number 4 is the last to carry out its subdomain solve in a previous multiplicative Schwarz iteration.

Like the classical alternating Schwarz method, the multiplicative Schwarz method is also sequential by nature. This is because in each iteration the P sub-steps for solving the PDE restricted onto Ω_i, $i = 1, 2, \ldots, P$, must be carried out in a fixed order. More precisely, simultaneous solution of (3)-(5) on more than one subdomain is not allowed due to (5).

In order for the multiplicative Schwarz method to be able to run on a multi-processor platform, it is necessary to modify the method by using a multi-coloring scheme. More specifically, we use a small number of different colors and assign each subdomain with one color. The result of multi-coloring is that any pair of neighboring subdomains always have different colors. In a modified multiplicative Schwarz method, all the subdomains that have the same color can carry out the subdomain solves simultaneously. Therefore, multiple processors are allowed to work simultaneously in subdomains with

the same color. It should be noted that the modified multiplicative Schwarz method is not so flexible as the additive Schwarz method (see below) when it comes to parallel implementation. The convergence speed of the modified multiplicative Schwarz method becomes slower compared with that of the standard multiplicative Schwarz method.

2.3 The Additive Schwarz Method

Another variant of overlapping DD methods, which inherently promotes parallel computing, is called *additive Schwarz method*. The difference between the multiplicative Schwarz method and the additive counterpart lies in the way how the artificial Dirichlet condition is updated on Γ_i. For the additive Schwarz method, we use

$$u_i^n = \tilde{g}^{n-1} \quad \text{on } \Gamma_i,$$

instead of (5). This means that the artificial Dirichlet condition on Γ_i is updated in a Jacobi fashion, using solutions from all the relevant neighboring subdomains from iteration number $n-1$. Therefore, the subdomain solves in the additive Schwarz method can be carried out completely independently, thus making the method inherently parallel. In more precise terms, for every point x that lies on Γ_i, we suppose it is also an interior point of the neighboring subdomains $\Omega_{j_1}, \Omega_{j_2}, \ldots, \Omega_{j_m}$, where $j_1 < j_2 < \ldots < j_m$ with $j_k \in \mathcal{N}_i$ for $k = 1, \ldots, m$. Then, the average value

$$\frac{1}{m} \sum_{k=1}^{m} u_{j_k}^{n-1}(x)$$

will be used as the artificial Dirichlet condition on $x \in \Gamma_i$. Although the additive Schwarz method suits well for parallel computing, it should be noted that its convergence property is inferior to that of the multiplicative Schwarz method. In case of convergence, the additive Schwarz method uses roughly twice as many iterations as that of the standard multiplicative Schwarz method. This is not surprising when the Schwarz methods are compared with their linear system solver analogues; multiplicative Schwarz is a block Gauss-Seidel approach, whereas additive Schwarz is a block Jacobi approach.

2.4 Formulation in the Residual Form

It is often more convenient to work with an equivalent formulation of the overlapping DD methods. More precisely, Equation (3) can be rewritten using a residual form:

$$u_i^n = u_i^{n-1} + L_i^{-1}(f_i - L_i u_i^{n-1}). \tag{6}$$

The above residual form is, e.g., essential for introducing the so-called coarse grid corrections (see Section 4.4). Moreover, by using a restriction

operator R_i from Ω to Ω_i, whose matrix representation consists of only ones and zeros, we can rewrite, e.g., the additive Schwarz method more compactly as

$$u^n = u^{n-1} + \sum_{i=1}^{P} R_i^T L_i^{-1} R_i (f - L u^{n-1}), \tag{7}$$

where R_i^T is an interpolation operator associated with R_i. To understand the effect of R_i, suppose we want to solve a scalar PDE on a global grid with M grid points. Let subgrid number i have M_i points, where point number j has index I_j^i in the global grid. Then, the corresponding matrix for R_i is of dimension $M_i \times M$. On row number j, there is only one entry of "1" at column number I_j^i, the other entries are "0". The matrix form of R_i^T is simply the transpose of that of R_i. However, in a practical implementation of an overlapping DD method, we can avoid physically constructing the global matrices and vectors. It is sufficient to work directly with the subdomain matrices/vectors, which together constitute their global counterparts. Thus, we do not need to know I_j^i explicitly, only the correspondence between neighboring subdomains in terms of the shared grid points is necessary. In other words, the R_i and R_i^T operators are only necessary for the correctness of the mathematical formulation, they are not explicitly used in an implementation where all the global vectors are distributed as subdomain vectors.

Consequently, iteration number $n+1$ of the multiplicative Schwarz method can be expressed in the residual form, consisting of P sequential sub-steps:

$$u^{n+\frac{1}{P}} = u^n + R_1^T L_1^{-1} R_1 (f - L u^n), \tag{8}$$

$$u^{n+\frac{2}{P}} = u^{n+\frac{1}{P}} + R_2^T L_2^{-1} R_2 (f - L u^{n+\frac{1}{P}}), \tag{9}$$

$$\cdots$$

$$u^{n+1} = u^{n+\frac{P-1}{P}} + R_P^T L_P^{-1} R_P (f - L u^{n+\frac{P-1}{P}}). \tag{10}$$

2.5 Overlapping DD as Preconditioner

So far, we have discussed the overlapping DD methods in the operator form using L_i, R_i and R_i^T. When we discretize (1)-(2), by e.g. the finite element method (FEM), a global system of linear equations will arise in the following matrix-vector form:

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{11}$$

If the above global linear system is large, it is beneficial to apply an iterative method. Moreover, preconditioning is often necessary for achieving fast convergence. That is, we solve

$$\mathbf{B}\mathbf{A}\mathbf{x} = \mathbf{B}\mathbf{b}$$

instead of (11), where $\mathbf{B}$ is some preconditioner expressed in the matrix form. The preconditioner $\mathbf{B}$ should be close to $\mathbf{A}^{-1}$, so that the condition number

of **BA** becomes much smaller than that of **A**. An optimal situation will be that the condition number of **BA** is independent of h, the grid size. Then the number of iterations needed by a global Krylov subspace method for obtaining a desired convergence will remain a constant, independent of h. In fact, for certain PDEs, one Schwarz iteration with coarse grid correction (see Section 4.4) can work as such an optimal preconditioner. Using overlapping DD methods as preconditioners in such a way actually results in more robust convergence than using them as stand-alone solvers.

In the operator form, the result of applying preconditioner B can be expressed by

$$v = Bw.$$

Typically, w is a residual vector given by $w = b - Ax$ and v is the resulting preconditioned correction vector.

We note that a Schwarz preconditioner, either additive or multiplicative, is no other than one Schwarz iteration with zero initial guess. More precisely, an additive Schwarz preconditioner produces v by

$$v = \sum_{i=1}^{P} R_i^T L_i^{-1} R_i w. \tag{12}$$

For a multiplicative Schwarz preconditioner, it is difficult to express it in a single and compact formula as above. But its preconditioned vector v can be obtained as u^{n+1}, if we set $f = w$ and $u^n = 0$ in (8)-(10). It is also worth noting that a multiplicative Schwarz preconditioner has better convergence property than its additive counterpart.

3 A 1D Example

In order to help the reader to associate the mathematical algorithm of the additive Schwarz method with concrete C++ programming, we present in the following a simple Diffpack program that applies the additive Schwarz method with two subdomains for solving a 1D Poisson equation: $-u''(x) = f$, where f is some constant.

More specifically, we cover the 1D solution domain $\Omega = [0, 1]$ with a global uniform lattice grid having n points, whose indices are from 1 to n. In addition, we introduce two indices a and b, where $b < a$, such that points with indices $1, \ldots, a$ form the left subdomain and points with indices $b, \ldots, n$ form the right subdomain. Note the condition $b < a$ ensures that the two subdomains are overlapping.

In the following Diffpack program, where the simulator class has name `DD4Poisson`, the data structure for the subdomain solvers consists of

- two `GridLattice` objects: `grid_1` and `grid_2` for the left and right subdomain, respectively,

- two `FieldLattice` objects containing the subdomain solutions from the current Schwarz iteration: `u_1` and `u_2`,
- two `FieldLattice` objects containing the subdomain solutions from the previous Schwarz iteration: `u_1_prev` and `u_2_prev`,
- two pairs of `MatTri(real)`/`ArrayGen(real)` objects: `A_1` and `b_1` for Ω_1 and `A_2` and `b_2` for Ω_2. They contain the subdomain linear systems.

The Diffpack program starts with preparing the subdomain grids and subdomain linear systems. Then, it creates the data structure for a global `GridLattice` object `grid` and a global `FieldLattice` object `u`. Thereafter, the program proceeds with a `setUpMatAndVec` function for filling up the values of the subdomain linear systems:

```
void DD4Poisson:: setUpMatAndVec()
{
  // set up matrices and vectors

  int i;
  const real h = grid->Delta(1);

  A_1.fill(0.0);
  A_2.fill(0.0);
  b_1.fill(0.0);
  b_2.fill(0.0);

  // left domain  (domain 1):
  A_1(1,0) = 1;
  b_1(1) = 0;
  for (i = 2; i < a; i++) {
    A_1(i,-1) = 1;  A_1(i,0) = -2;  A_1(i,1) = 1;
    b_1(i) =  -f*h*h;
  }
  A_1(a,0) = 1;
  // b_1(a) is updated for each iteration, in solveProblem()

  A_1.factLU();

  // right domain (domain 2):
  A_2(1,0) = 1;
  // b_2(1) is updated for each iteration, in solveProblem()
  for (i = 2; i < n-b+1; i++) {
    A_2(i,-1) = 1;  A_2(i,0) = -2;  A_2(i,1) = 1;
    b_2(i) = - f*h*h;
  }
  A_2(n-b+1,0) = 1;
  b_2(n-b+1) = 0;

  A_2.factLU();
}
```

The main computational loop of the additive Schwarz method looks as follows:

```
for (int j = 2; j <= no_iter; j++) {
```

```
  // local solution in each subdomain, and update the global domain
  b_1(a) = u_2_prev.values()(a);
  // the input rhs vector is changed during the solution process
  // we send a copy of the b_1 vector to the function
  rhs_tmp = b_1;
  A_1.forwBackLU(rhs_tmp,u_1.values());  // Gaussian elimination
  b_2(1) = u_1_prev.values()(b);
  rhs_tmp = b_2;
  A_2.forwBackLU(rhs_tmp,u_2.values());  // Gaussian elimination

  updateValues();
}
```

In the above loop, the subdomain matrices `A_1` and `A_2` remain constant, so the LU factorization is carried out once and for all inside `setUpMatAndVec`, whereas only two calls of `forwBackLU` need to be invoked in each Schwarz iteration. Furthermore, the change of the right-hand side vectors `b_1` and `b_2` from iteration to iteration is small, i.e., only the last entry of `b_1` and the first entry of `b_2` need to be updated. The work of function `updateValues` is to store the current subdomain solutions for the next Schwarz iteration, while also updating the global solution field `u`. We mention that the points of the right subdomain grid, `grid_2`, have indices $b, \ldots, n$. This greatly simplifies the procedure of mapping the subdomain nodal values from `u_2` to `u`, as is shown below.

```
void DD4Poisson:: updateValues()
{
  int i;

  // remember the subdomain solutions for the next iteration
  u_1_prev.values() = u_1.values();
  u_2_prev.values() = u_2.values();

  // update global domain values
  for (i = 1; i <= b; i++)
    { u.values()(i) = u_1.values()(i); }

  for (i = b+1; i < a; i++)
    { u.values()(i) = (u_1.values()(i) + u_2.values()(i))/2; }

  for (i = a; i <= n; i++)
    { u.values()(i) = u_2.values()(i); }
}
```

4 Some Important Issues

4.1 Domain Partitioning

The starting point of an overlapping DD method is a set of overlapping subdomains $\{\Omega_i\}_{i=1}^P$. There is, however, no standard way of generating $\{\Omega_i\}_{i=1}^P$

from a global domain Ω. One commonly used approach is to start with a global fine grid $\mathcal{T}$ that covers Ω and consider the graph that corresponds to $\mathcal{T}$. By a graph we mean a collection of nodes, where some nodes are connected by edges. Note that nodes and edges in a graph are not the same as the grid points and element sides of $\mathcal{T}$. For the graph that arises from a finite element grid $\mathcal{T}$, each element is represented by a node, an edge exists between two nodes if the two corresponding elements are neighbors; see e.g. [4,9]. That is, the graph is a simplified topological representation of the grid, without considering the size and location of the elements. We can then use some graph partitioning algorithm to produce non-overlapping subdomains, such that every element belongs to a unique subdomain. Thereafter, a post-processing procedure is necessary to enlarge the subdomains and thereby ensure that every grid point on Γ_i eventually also lies in the interior of at least one neighboring subdomain.

4.2 Subdomain Discretizations in Matrix Form

When the set of overlapping subdomains $\{\Omega_i\}_{i=1}^P$ and the associated subgrids $\mathcal{T}_i$ are ready, the next step for a DD method is to carry out discretization on each subgrid $\mathcal{T}_i$. An important observation about overlapping DD methods is that the subproblems arise from restricting the original PDE(s) onto the subdomains. Discretization on $\mathcal{T}_i$ can therefore be done straightforwardly, without having to first construct the global stiffness matrix $\mathbf{A}$ for Ω and then "cut out" the portion corresponding to $\mathbf{A}_i$.

Let us denote respectively by $\mathbf{A}_i^0$ and $\mathbf{f}_i^0$ the resulting local stiffness matrix and right-hand vector, without enforcing the artificial Dirichlet condition on Γ_i. Then, to incorporate the artificial Dirichlet condition, we need to remove all the non-zero entries on the rows of $\mathbf{A}_i^0$ that correspond to the grid points on Γ_i. In addition, the main diagonal entries for these rows are set to one. The modified subdomain matrix $\mathbf{A}_i$ remains unchanged during all the DD iterations. During DD iteration number i, the right-hand side vector $\mathbf{f}_i^n$ is obtained by modifying $\mathbf{f}_i^0$ on the entries that correspond to the grid points on Γ_i. Those entries will use values that are provided by the neighboring subdomains.

From now on we will use the matrix representations $\mathbf{A}_i$, $\mathbf{A}_i^{-1}$, $\mathbf{R}_i$, $\mathbf{R}_i^T$ for L_i, L_i^{-1}, R_i, R_i^T, respectively.

4.3 Inexact Subdomain Solver

From Section 2 we can see that the main computation of an overlapping DD method happens in the subdomain solves (3)-(5). Quite often, the size of the subgrids may still be large, therefore preventing an exact subdomain solution in form of $\mathbf{A}_i^{-1}$. This means that we have to resort to some iterative solution method also for the subdomain problems. That is, we use an inexact subdomain solver $\tilde{\mathbf{A}}_i^{-1}$, which approximates the inverse of $\mathbf{A}_i$ in some sense.

However, the subdomain problems have to be solved quite accurately when overlapping DD methods are used as stand-alone iterative solvers. When an overlapping DD method is used as a preconditioner, it is sufficient to solve the subdomain problems only to a certain degree of accuracy. In the latter case, it is standard to use in the subdomain solves an iterative Krylov solver with a loose convergence criterion, or even one multigrid V-cycle [5]. This may greatly enhance the overall efficiency of the overlapping DD methods.

4.4 Coarse Grid Corrections

One weakness with the Schwarz methods from Section 2 is that convergence deteriorates as the number of subdomains increases, especially when an additive Schwarz method is used as a stand-alone iterative solver. This is due to the fact that each subdomain problem is restricted to Ω_i. The global information propagation, in form of data exchange within the overlapping regions, happens only between immediate neighboring subdomains during one DD iteration. That is, the information of a change at one end of Ω needs several DD iterations to be propagated to the other end of Ω.

To incorporate a more rapid global information propagation mechanism for faster convergence, we may use a coarse global grid $\mathcal{T}_H$ that covers the whole domain Ω. Then an additional global coarse grid problem is solved in every DD iteration. Such overlapping DD methods with *coarse grid corrections* can be viewed as a variant of two-level multigrid methods, where the collaborative subdomain solves work as a special multigrid smoother, see e.g. [10]. For example, the additive Schwarz preconditioner equipped with a coarse grid solver can be expressed as:

$$\mathbf{v} = \sum_{i=0}^{P} \mathbf{R}_i^T \tilde{\mathbf{A}}_i^{-1} \mathbf{R}_i \mathbf{w}, \tag{13}$$

where we denote by $\tilde{\mathbf{A}}_0^{-1}$ a solver for the global coarse grid problem, $\mathbf{R}_0$ and $\mathbf{R}_0^T$ represent the restriction and interpolation matrices describing the mapping between $\mathcal{T}$ and $\mathcal{T}_H$. In the later implementation, the matrices $\mathbf{R}_0$ $\mathbf{R}_0^T$ are not formed explicitly. Their operations are achieved equivalently by using a series of mapping matrices between $\mathcal{T}_i$ and $\mathcal{T}_H$. We also note that (13) allows inexact subdomain solvers.

4.5 Linear System Solvers at Different Levels

It is worth noticing that overlapping DD methods involve linear system administration on at least two levels (three if coarse grid corrections are used). First of all, we need a *global* administration of the solution of the global linear system, e.g., calculating the global residual, updating the artificial Dirichlet conditions, and monitoring the convergence of the global solution. Secondly, we need a separate *local* administration for handling the local linear system

restricted to each subdomain. In case an iterative subdomain linear system solver is chosen, it is normally sufficient to use a less strict subdomain stopping criterion than that used for monitoring the convergence of the global DD iterations. An important observation is that although linear solvers exist at both global and subdomain levels, there is no absolute need for constructing physically the global matrices and vectors. We can actually adopt a *distributed data storage* scheme, where we only store for each subdomain its subdomain matrix and vector. These subdomain matrices and vectors, which are simply parts of their global counterparts, can together give a *logical* representation of the global matrix and vector. More specifically, all the global linear algebra operations can be achieved by running local linear algebra operations on the subdomains, together with necessary data exchange between the subdomains. This allows us to avoid physical storage of the global matrices and vectors, thus making the transition from a sequential overlapping DD code to a parallel code quite straightforward.

5 Components of Overlapping DD Methods

Before moving on to the implementation of the overlapping DD methods, we find it important to view these methods at a higher abstraction level. This is for gaining insight into the different components that form the algorithmic structure.

We have pointed out in Section 4.1 that the starting point of an overlapping DD method is a set of overlapping subdomains. Based on this overlapping subdomain set, the basic building block of an overlapping DD method is its subdomain solvers. The subdomain solvers should be flexible enough to handle different shapes of subdomains and mixture of physical and artificial boundary conditions. They also need to be numerically efficient, because the overall efficiency of an overlapping DD method depends mainly on the efficiency of the subdomain solvers.

Due to the necessity of updating the artificial Dirichlet condition (5) during each DD iteration, data must be exchanged between neighboring subdomains. In order to relieve the subdomain solvers of the work of directly handling the data exchange, there is need for a so-called "data exchanger". This component has an overview of which grid points of $\mathcal{T}_i$ that lie on Γ_i, and knows which neighboring subdomain should provide which part of its solution along Γ_i. For every pair of neighboring subdomains, the task is to construct data exchange vectors by retrieving grid point values from corresponding locations of the local solution vector and then make the exchange.

In order to coordinate the subdomain solves and the necessary data exchanges, it is essential to have a global administration component. The tasks of this component include controlling the progress of DD iterations, monitoring the global convergence behavior, starting the subdomain solves, and invoking data exchange via the data exchange component.

As a summary, an overlapping DD method should have the following main components:

- Subgrid preparation component;
- Subdomain solver component;
- Data exchange component;
- Global administration component.

6 A Generic Implementation Framework

If there already exists a global solver for a PDE, then it should, in principle, also work as a subdomain solver in an overlapping DD method. This is due to the fact that the subdomain problems of an overlapping DD method are of the same type as the original global PDE problem. Of course, small modifications and/or extensions of the original global solver are necessary, such as for incorporating the artificial Dirichlet boundary condition (5).

Object-oriented programming is an ideal tool for code extensions. The objective of this section is thus to present a generic object-oriented implementation framework, where an existing Diffpack PDE solver can be easily extended for incorporating an overlapping DD method. The implementation framework is used to simplify the coding effort and promote a structured design.

6.1 The Simulator-Parallel Approach

The traditional way of programming overlapping DD methods is to implement from scratch all the involved algebra-level operations, like matrix-vector product, residual vector calculation etc. We use however a so-called *simulator-parallel* approach, which was first implemented in a generic fashion in [1] for implementing parallel DD solvers, but applies equally well to sequential DD solvers. This approach takes an existing PDE simulator as the starting point. The simulator is re-used as the subdomain solver component, after minor code modifications and/or extensions. Each subdomain is then assigned with such a subdomain solver. The computation coordination of the subdomain solvers is left to a global administrator, which is implemented at a high abstraction level close to the mathematical formulation of the overlapping DD methods.

The simulator-parallel approach can be used to program any overlapping DD code, which is achievable by the traditional linear-algebra level programming approach. This is because any object-oriented sequential simulator, which has data structure for local matrices/vectors, a numerical discretization scheme, and a linear algebra toolbox, is capable of carrying out all the operations needed in the subdomain solves. An advantage of the simulator-parallel approach is that each subdomain simulator "sees" the physical properties of the involved PDEs, which may be used to speed up the solution of

the subproblems. Furthermore, a subdomain simulator may have independent choice of its own solution method, preconditioner, stopping criterion etc. Most importantly, developing DD solvers in this way strongly promotes code reuse, because most of the global administration and the related data exchange between subdomains can be extracted from specific applications, thereby forming a generic library.

6.2 Overall Design of the Implementation Framework

As we have pointed out in Section 5, the subdomain solvers are only one of the components that form the algorithmic structure of an overlapping DD method. The other components are normally independent of specific PDEs, thus enabling us to build a generic implementation framework where components are realized as standardized and yet extensible objects.

The implementation framework is made up of three parts. In addition to the subdomain solvers, the other two parts are a communication part and a global administrator. The communication part contains a subgrid preparation component and a data exchange component, as mentioned in Section 5. In this way, the data exchange component has direct access to the information about how the overlapping subgrids are formed. The information will thus be used to carry out data exchanges between neighboring subdomains.

When using the framework for implementing an overlapping DD method, the user primarily needs to carry out a coding task in form of deriving two small-sized C++ subclasses. The first C++ subclass is for modifying and extending an existing PDE simulator so that it can work as a subdomain solver. The second C++ subclass is for controlling the global calculation and coupling the subdomain solver with the other components of the implementation framework. The coding task is quite different from that of incorporating a multigrid solver into a Diffpack simulator, see [6], where a toolbox is directly inserted into an existing PDE simulator without having to derive new C++ subclasses. This seemingly cumbersome implementation process of an overlapping DD method is necessary due to the existence of linear system solvers at different levels, see Section 4.5. We therefore let the global administrator take care of the global linear system, while the subdomain solvers control the solution of the subdomain linear systems. Besides, we believe this coding rule promotes a structured implementation process with sufficient flexibility.

6.3 Subdomain Solvers

A typical Diffpack PDE simulator has functionality for doing discretization on a grid and solving the resulting linear system. In order for such an existing Diffpack PDE simulator to be accepted by the implementation framework, it is important that the simulator is first "wrapped up" within a generic standard interface recognizable by the other generic components. We have therefore designed

```
class SubdomainFEMSolver : public virtual HandleId
```

that provides such a generic interface. The purpose of this class is to allow different components of the implementation framework to access the subdomain data structure and invoke the functions related to the subdomain solves. The main content of class `SubdomainFEMSolver` consists of a set of data object handles and virtual member functions with standardized names. It is also through these handles and functions that the generic implementation framework utilizes the data and functions belonging to an existing PDE simulator.

The most important internal variables of class `SubdomainFEMSolver` are the following data object handles:

```
Handle(GridFE)    subd_fem_grid;
Handle(DegFreeFE) subd_dof;
Handle(LinEqAdm)  subd_lineq;
Handle(Matrix(real)) A_orig;
Handle(Matrix(real)) A_new;
Handle(Vec(real)) global_solution_vec;
Handle(Vec(real)) global_orig_rhs;
Handle(Vec(real)) global_residual_vec;
Handle(Vec(real)) solution_vec;
Handle(Vec(real)) orig_rhs;
Handle(Vec(real)) rhs_vec;
Handle(Vec(real)) residual_vec;
```

The above handles can be divided into three types. The first type of handles are to be bound to external data objects created by some other components of the generic implementation framework. The `subd_fem_grid` handle is such an example. More precisely, the subdomain solvers of the implementation framework are no longer responsible for constructing the subgrids, but receive them from the global administrator (see Section 6.5). It is important to note that the subgrid is the starting point for building up the data structure of a subdomain solver. The second type of handles are to be bound to external data objects living inside an existing PDE simulator, thus enabling the implementation framework to utilize those data objects. Handles `subd_dof`, `subd_lineq`, `A_orig`, and `global_solution_vec` belong to this second type. The rest of the above handles belong to the third type and are to be bound to some internal data objects, which are specially needed for carrying out a generic subdomain solution process.

When deriving a new subdomain solver, which is to be a subclass of both class `SubdomainFEMSolver` and an existing PDE simulator class, we normally only make direct use of the handles `subd_fem_grid`, `subd_dof`, `subd_lineq`, and `global_solution_vec`. The other handles of `SubdomainFEMSolver` are normally used within the implementation framework. For example, the matrix object that contains the local matrix $\mathbf{A}_i^0$ lies physically inside an existing PDE simulator. It is assumed to be physically allocated in the `LinEqAdm` object to which the `subd_lineq` handle is bound. We therefore do not need to directly access the `A_orig` handle, which is to be bound to the $\mathbf{A}_i^0$ matrix object inside a

function named `SubdomainFEMSolver::modifyAmatWithEssIBC`. This function is normally invoked by the global administrator to produce the internal matrix object $\mathbf{A}_i$, which is pointed by the handles `A_new`.

We note that all the matrix and vector objects, which belong to one object of `SubdomainFEMSolver`, arise from the particular subgrid $\mathcal{T}_i$, not the global grid $\mathcal{T}$. That is, the prefix `global` rather reminds us of the fact that we need to operate with linear systems at two different levels. The first level concerns the logically existing global linear system. For example, `global_residual_vec` is used for monitoring the convergence of the global solution. The second level concerns the subdomain local linear system. For example, `solution_vec` represents $\mathbf{u}_i^n$ that is involved in every subdomain solve $\mathbf{A}_i\mathbf{u}_i^n = \mathbf{f}_i^n$. In other words, we avoid the physical storage of global matrices and vectors that are associated with the entire global grid $\mathcal{T}$. This is because any global linear algebra operation can be realized by local linear algebra operations plus necessary data exchange, as explained in Section 4.5.

The aforementioned function `modifyAmatWithEssIBC` is an important member function of class `SubdomainFEMSolver`. In addition, the other five important member functions of class `SubdomainFEMSolver` are:

```
virtual void initSolField (MenuSystem& menu) =0;
virtual void createLocalMatrix ();
virtual void initialize (MenuSystem& menu);
virtual void updateRHS ();
virtual void solveLocal ();
```

All these six member functions rarely need to be directly invoked by the user. Among them only `initSolField` needs to be *explicitly* re-implemented in a subclass. The task of this function is to make sure that the needed external data objects are constructed and the handles `subd_dof`, `subd_lineq`, and `global_solution_vec` are correctly bound to those external data objects. Besides, the member function `createLocalMatrix`, which is used for building $\mathbf{A}_i^0$ and $\mathbf{f}_i^0$, normally also requires re-implementation. For most cases, this can be done by simply using

```
FEM:: makeSystem (*subd_dof, *subd_lineq);
```

The other three member functions, like `modifyAmatWithEssIBC`, have a default implementation that seldom needs to be overridden in the subclass. The task of the member function `initialize` is to invoke the user re-implemented member function `initSolField`, in addition to preparing the internal data objects like `residual_vec`, `solution_vec`, `rhs_vec`, and `orig_rhs`. The member function `updateRHS` is normally used by the global administrator for updating the right-hand side vector $\mathbf{f}_i^n$, see Section 4.2, before solving the subdomain problem in iteration number n. Moreover, the member function `solveLocal` has the following default implementation:

```
void SubdomainFEMSolver:: solveLocal ()
{
```

```
    subd_lineq->solve (first_local_solve);
    first_local_solve = false;
  }
```

where `first_local_solve` is an internal boolean variable belonging to class `SubdomainFEMSolver`. Since the subdomain matrix $\mathbf{A}_i$ does not change in the DD iterations, the `first_local_solve` variable thus gives this signal to the internal `LinEqAdm` object. We remark that `first_local_solve` is assigned with a `dpTrue` value inside every call of the `modifyAmatWithEssIBC` function, which follows every call of the `createLocalMatrix` function. We also remark that the user is responsible for deciding how the subdomain linear system (with an updated right-hand side $\mathbf{f}_i^n$) should be solved in each DD iteration. A typical choice can be a Krylov subspace solver using a moderate-sized tolerance in the stopping criterion.

Such a generic subdomain solver interface makes it easy to incorporate an existing PDE simulator into the generic implementation framework. Given an existing PDE simulator, say `MySolver`, we can construct a new subdomain solver that is recognizable by the generic implementation framework as follows:

```
class MySubSolver : public MySolver, public SubdomainFEMSolver
```

After re-implementing the virtual member functions such as `initSolField` and `createLocalMatrix`, we can readily "plug" the new subdomain solver `MySubSolver` into the generic implementation framework.

Class CoarseGridSolver. The data structure and functionality needed in coarse grid corrections are programmed as a small general class with name `CoarseGridSolver`. Inside class `SubdomainFEMSolver` there is a following handle

```
Handle(CoarseGridSolver) csolver;
```

which will be bound to a `CoarseGridSolver` object if the user wants to use coarse grid corrections. The other variables and member functions of class `SubdomainFEMSolver` that concern coarse grid corrections are:

```
Handle(DegFreeFE) global_coarse_dof;
virtual void buildGlobalCoarseDof ();
virtual void createCoarseGridMatrix ();
```

When coarse grid corrections are desired, the implementation framework equips *each* `SubdomainFEMSolver` object with an object of `CoarseGridSolver`. Of course, only one `CoarseGridSolver` object (per processor) needs to contain the data for e.g. $\mathbf{A}_0$. But every `CoarseGridSolver` object has the data structure for carrying out operations such as to map a subdomain vector associated with $\mathcal{T}_i$ to a global vector associated with $\mathcal{T}_H$, where the combined result from all the subdomains is a mapping between $\mathcal{T}$ and $\mathcal{T}_H$. The user only needs to re-implement the two following virtual member functions of `SubdomainFEMSolver`

```
  virtual void buildGlobalCoarseDof ();
  virtual void createCoarseGridMatrix ();
```

in a derived subclass. The purpose of the first function is to construct the `global_coarse_dof` object and fill it with values of essential boundary conditions, if applicable. The user should normally utilize the default implementation of `SubdomainFEMSolver::buildGlobalCoarseDof`, which constructs the `global_coarse_dof` object without considering the boundary conditions. The second function is meant for carrying out an assembly process for building the global coarse grid matrix $\mathbf{A}_0$. In the following, we list the default implementation in class `SubdomainFEMSolver`.

```
void SubdomainFEMSolver:: buildGlobalCoarseDof ()
{
  // menu_input is an internal MenuSystem pointer
  String cg_info = menu_input->get("global coarse grid");
  Handle(GridFE) coarse_grid = new GridFE;
  readOrMakeGrid (*coarse_grid, cg_info);

  const int cg_ndpn
        = menu_input->get("degrees of freedom per node").getInt();
  global_coarse_dof.rebind (new DegFreeFE(*coarse_grid, cg_ndpn));
}

void SubdomainFEMSolver:: createCoarseGridMatrix ()
{
  // user must redefine this function to include assembly
  if (!csolver->coarse_lineq.ok() ||
      !csolver->coarse_lineq->bl().ok()) {
    warningFP("SubdomainFEMSolver::createCoarseGridMatrix",
              "You should probably redefine this function!");
    return;
  }

  if (csolver->coarse_lineq.ok() &&
      csolver->coarse_lineq->getMatrixPrm().storage=="MatBand") {
    // carry out LU-factorization only once
    FactStrategy fstrat;
    LinEqSystemPrec& sys=csolver->coarse_lineq->getLinEqSystem();
    sys.allow_factorization = true;
    csolver->coarse_lineq->A().factorize (fstrat);
    sys.allow_factorization = false;
  }
}
```

Examples of how these two member functions can be re-implemented in a subclass of `SubdomainFEMSolver` can be found in Section 8.

6.4 The Communication Part

Data exchange between neighboring subdomains within the overlapping regions is handled by class `CommunicatorFEMSP` in the generic implementation framework. This class is derived as a subclass of `GridPartAdm`, see [2], so the

functionalities for overlapping subgrid preparation and data exchange are automatically inherited.

We note that a sequential DD solver is simply a special case of a parallel DD solver. In a sequential DD solver all the subdomains reside on a single processor, and class `CommunicatorFEMSP` is capable of handling this special case. This part of the implementation framework seldom requires reimplementation and the user normally does not invoke it directly.

6.5 The Global Administrator

The global administrator of the implementation framework is constituted by several generic classes. The responsibility of the global administrator includes construction of the subdomain solvers and the communication part, choosing a particular DD method during a setting-up phase, and invoking all the necessary operations during each DD iteration.

Class ParaPDESolver. We recall that overlapping DD methods can work as both stand-alone iterative solution methods and preconditioners for Krylov subspace methods. By using the object-oriented programming techniques, we wish to inject this flexibility into the design of the global administrator. We have therefore designed class `ParaPDESolver` to represent a generic DD method.Two subclasses have also been derived from `ParaPDESolver`, where the first subclass `BasicDDSolver` implements an overlapping DD method to be used as a stand-alone iterative solver, and the second subclass `KrylovDDSolver` implements an overlapping DD preconditioner.

Class ParaPDESolver_prm. In order to allow the user to choose, *at run-time*, whether to use an overlapping DD method as a preconditioner or as a stand-alone iterative solver, we have followed the Diffpack standard and devised a so-called parameter class with name `ParaPDESolver_prm`. The most important member function of this simple parameter class is

```
virtual ParaPDESolver* create () const;
```

which creates a desired `BasicDDSolver` or `KrylovDDSolver` object at run-time. In addition to the type of the DD method, class `ParaPDESolver_prm` also contains other parameters whose values can be given through the use of `MenuSystem`. The most important parameters of class `ParaPDESolver_prm` are:

1. A flag indicating whether the overlapping DD method should be used as a preconditioner for a Krylov method, or as a stand-alone iterative solver.
2. A `LinEqSolver_prm` object for choosing a particular global Krylov method when DD is to work as a preconditioner, the maximum number of Krylov or DD iterations, and a prescribed accuracy.
3. A `ConvMonitorList_prm` object for choosing a global convergence monitor.

Below, we also list a simplified version of the `defineStatic` function that belongs to class `ParaPDESolver_prm`.

```
void ParaPDESolver_prm:: defineStatic(MenuSystem& menu, int level)
{
  String label = "DD solvers and associated parameters";
  String command = "ParaPDESolver_prm";
  MenuSystem::makeSubMenuHeader(menu,label,command,level,'p');
  menu.addItem (level,
                "domain decomposition scheme","dd-scheme",
                "class name in ParaPDESolver hierarchy",
                "KrylovDDSolver",
                "S/BasicDDSolver/KrylovDDSolver");
  LinEqSolver_prm::defineStatic (menu, level+1);
  ConvMonitorList_prm::defineStatic (menu, level+1);
}
```

Class SPAdmUDC. The main class of the global administrator is `SPAdmUDC`, which controls parameter input and the creation of a desired DD solver. In addition, class `SPAdmUDC` also controls the communication part and all the generic subdomain solvers. We remark that "`UDC`" stands for "user-defined-codes" and is used to indicate that the user has the possibility of re-implementing `SPAdmUDC`'s member functions and introducing new functions when he or she derives a new subclass. The simplified definition of class `SPAdmUDC` is as follows:

```
class SPAdmUDC : public PrecAction
{
  int num_local_subds;
  bool use_coarse_grid;
  bool additive_schwarz;
  Handle(MenuSystem) menu_input;

  VecSimplest(Handle(GridFE)) subd_fem_grids;
  VecSimplest(Handle(SubdomainFEMSolver)) subd_fem_solvers;
  Handle(ParaPDESolver_prm) psolver_prm;
  Handle(ParaPDESolver) psolver;
  Handle(CommunicatorFEMSP) communicator;

  virtual void setupSimulators ();
  virtual void setupCommunicator ();
  virtual void prepareGrids4Subdomains();
  virtual void prepareGrids4Subdomains(const GridFE& global_grid);
  virtual void createLocalSolver (int subd_id);
  virtual void init (GridFE* global_grid = NULL);
  // DD as stand-alone solver
  virtual void oneDDIteration (int iteration_counter);
  virtual void genAction   // DD used as preconditioner
    (LinEqVector&, LinEqVector&, TransposeMode);

  static  void defineStatic (MenuSystem& menu, int level = MAIN);
  virtual void scan         (MenuSystem& menu);
  virtual bool solve ();
};
```

Among the above member functions of `SPAdmUDC`, three of them are meant to be invoked directly by the user:

1. The `defineStatic` function designs the layout of an input menu.

```
void SPAdmUDC:: defineStatic (MenuSystem& menu, int level)
{
  // choices such as whether DD is to be used as preconditioner
  ParaPDESolver_prm::defineStatic (menu, level);

  String label
    = "User-defined components of domain decomposition";
  String command = "SPAdmUDC";
  MenuSystem::makeSubMenuHeader(menu,label,command,level,'U');

  GridPartAdm::defineStatic (menu,level+1);  // subd grid creation

  menu.addItem (level,
                "use additive Schwarz","additive-Schwarz",
                "additive Schwarz","ON","S/ON/OFF");

  // concerning coarse-grid-correction
  menu.addItem (level,
                "use coarse grid","use_coarse_grid",
                "use coarse grid","OFF","S/ON/OFF");
  menu.addItem (level,
                "global coarse grid","gcg",
                "global coarse grid","global-coarse.grid","S");
  menu.addItem (level,
                "degrees of freedom per node","ndpd",
                "ndpn-global-coarse-grid","1","I1");
  menu.setCommandPrefix ("coarsest");
  LinEqAdm::defineStatic (menu,level+1); //for coarse grid correc.
  menu.unsetCommandPrefix ();
}
```

2. The `scan` function reads the input parameters and does the necessary initialization work.

```
void SPAdmUDC:: scan (MenuSystem& menu)
{
  menu_input.rebind (menu);

  psolver_prm.rebind (new ParaPDESolver_prm);
  psolver_prm->scan (menu);

  additive_schwarz = menu.get("use additive Schwarz").getBool();
  use_coarse_grid = text2bool (menu.get("use coarse grid"));

  // other menu items are handled by 'SubdomainFEMSolver' later

  init ();   // do all the initialization work
}
```

3. The `solve` function carries out the solution of a logically existing global linear system using a particular DD method.

```
bool SPAdmUDC:: solve ()
{
  return psolver->solve();
}
```

Inside the `scan` function, the initialization of all the components of a chosen DD method is done by the `init` function, which invokes in sequence: `prepareGrids4Subdomains`, `setupSimulators`, and `setupCommunicator`. The final work of the `init` function is to create the chosen DD method by

```
psolver.rebind(psolver_prm.create());
```

The solution of a virtual global linear system by the chosen DD method will be done by the virtual member function `ParaPDESolver::solve`. For class `BasicDDSolver` the `solve` function has the following algorithmic core:

```
bool BasicDDSolver:: solve ()
{
  // udc is of type SPAdmUDC*
  iteration_counter = 0;
  while (iteration_counter++<max_iterations && !satisfied())
    udc->oneDDIteration (iteration_counter);
  return convflag;
}
```

where the `oneDDIteration` function of class `SPAdmUDC` has the following simplified implementation:

```
void SPAdmUDC:: oneDDIteration (int /*iteration_counter*/)
{
  if (use_coarse_grid)
    coarseGridCorrection ();
  for (int i=1; i<=num_local_subdomains; i++) {
    local_fem_solvers(i)->updateRHS ();
    local_fem_solvers(i)->solveLocal ();
  }
  updateGlobalValues ();
}
```

We mention that it is possible for the user to re-implement the `oneDDIteration` function in a derived subclass to design a non-standard DD method as a stand-alone solver. We also mention that the `genAction` function does the work of a DD preconditioner, associated with class `KrylovDDSolver`. The function is invoked inside every iteration of a chosen global Krylov method. Re-implementation of `genAction` is also allowed in a derived subclass.

Unlike most member functions of class `SPAdmUDC`, the `createLocalSolver` function *must* be re-implemented in a derived subclass. The task of the function is to bind each entry of the `subd_fem_solvers` handle array to a newly defined subdomain solver, say `MySubSolver`. Therefore, the simplest re-implementation of `createLocalSolver` in a subclass of `SPAdmUDC` is

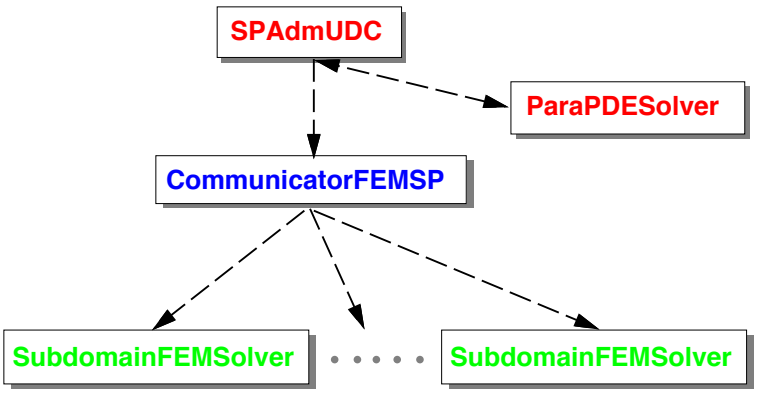

Fig. 3. A generic implementation framework for sequential overlapping DD methods. Note that the dashed arrows mean the "has a" relationship.

```
void MyUDC:: createLocalSolver (int local_id)
{
  subd_fem_solvers(local_id).rebind (new MySubSolver);
}
```

However, if the global administrator also wants to use some non-generic data objects or functions of the newly defined subdomain solver, it is necessary to introduce in the class definition of `MyUDC` an additional handle array

```
VecSimplest(Handle(MySubSolver)) subd_my_solvers;
```

Then a more advanced version of `MyUDC::createLocalSolver` can be as follows:

```
void MyUDC:: createLocalSolver (int local_id)
{
  MySubSolver* my_solver = new MySubSolver;
  subd_my_solvers.redim (num_local_subds);
  subd_my_solvers(local_id).rebind (*my_solver);
  subd_fem_solvers(local_id).rebind (*my_solver);
}
```

6.6 Summary of the Implementation Framework

To summarize, we present a schematic diagram of the generic implementation framework in Figure 3. In the diagram, the top two classes `SPAdmUDC` and `ParaPDESolver` constitute the global administrator. Class `CommunicatorFEMSP` constitutes the communication part, which has a direct connection to all the subdomain solvers. We note that we have omitted drawing the direct connection from `SPAdmUDC` to the subdomain solvers in the diagram.

7 Parallel Overlapping DD Methods

Although one major strength with the overlapping DD methods is their suitability for parallel computing, they can achieve superior numerical efficiency even on sequential computers. In particular, the content of Sections 2-6 applies for both a sequential and a parallel implementation. For sequential overlapping DD methods, all the subdomains reside on the same processor, and one object of class `SubdomainFEMSolver` is used for each subdomain. The global iteration administration and data exchange are handled by a *single* set of class objects containing `SPAdmUDC`, `ParaPDESolver_prm`, `ParaPDESolver`, and `CommunicatorFEMSP`.

For running e.g. an additive Schwarz method on a multi-processor platform, the only difference is that every involved processor will use its own set of class objects for iteration administration and data exchange. Furthermore, one processor may have one or several subdomain solvers, depending on the relation between the number of subdomains and the number of processors. Since class `CommunicatorFEMSP` is derived from `GridPartAdm`, the situation where one processor is responsible for multiple subdomains can be handled by exactly the *same* Diffpack code as that for the situation of one subdomain per processor. By considering a sequential overlapping DD execution as a special case of a parallel overlapping DD execution, we can easily understand that the difference between the sequential code and the parallel code exists only in the parallel Diffpack library and is thus invisible to the user.

Following the implementation framework explained in the preceding section, every processor has a global coarse grid solver with complete data structure for $\mathbf{A}_0$. Before each processor calculates the same coarse grid correction, *all* the subdomain solvers need to communicate with each other to form a global residual vector $\mathbf{w}_0$ that is associated with the global coarse grid $\mathcal{T}_H$. Therefore, inter-processor communication after solving

$$\mathbf{A}_0\mathbf{u}_0 = \mathbf{w}_0 \tag{14}$$

is not necessary. However, the size of the coarse grid problem needs some attention. A finer $\mathcal{T}_H$ normally gives better overall convergence, but it may come at the cost of lower parallel efficiency, due to increased overhead associated with the coarse grid correction. So it is important to have a coarse grid problem that is considerably smaller than every subdomain problem. Under this assumption, it suffices to use a sequential solution procedure for (14).

In practice, it is unusual that a parallel DD solver demonstrates linear speed-up results, i.e., the execution time decreases linearly as a function of the number of processors. This is due to the overhead that consists of (a) computation not being parallelized, like the above mentioned way of treating the coarse grid correction, and (b) work that is only present in a parallel execution. The latter type of overhead again can be attributed to two sources:

1. overhead of communication, and

2. overhead of synchronization.

The communication overhead is determined by both the amount of information neighboring subdomains need to exchange and the number of neighbors each subdomain has. This is because the cost of sending/receiving a message between two processors roughly consists of two parts: a constant start-up cost and the message exchange cost that is proportional to the message size. So one of the requirements for the overlapping grid partitioning should be to keep the average number of neighbors small. Meanwhile, reducing the size of overlap between neighboring subdomains will also reduce the overhead, but may however affect the convergence speed. *Therefore, a good overlapping grid partitioning algorithm should provide the user with the freedom of adjusting the size of overlap to reach a compromise between fast convergence and low communication overhead.*

The synchronization overhead is primarily due to the difference in subgrid size. That is, some subdomains have more computation because they have more grid points. However, unbalanced local convergence may also destroy the work balance, even if the grid partitioning algorithm produces subgrids of an identical size (e.g. out from a structured global grid). For example, if a Krylov subspace method is used in the subdomain solves, different subdomains may then need different numbers of iterations to achieve the local convergence. This situation of unbalanced local convergence may change from one DD iteration to another. *Therefore, if applicable, a fixed number of multigrid V-cycles are optimal subdomain solvers with respect to both computational efficiency and synchronization overhead.*

8 Two Application Examples

In order to show the reader how an overlapping DD method can be incorporated into an existing PDE simulator, we present in this section two application examples. The first example is for solving a stationary PDE, whereas the second example is for solving a time-dependent PDE.

8.1 The Poisson1 Example

We use the standard Diffpack sequential Poisson equation solver `Poisson1`[1] as the starting point for this example. The coding effort needed for incorporating an overlapping DD method is in form of creating two new C++ classes: `SubdomainPESolver` as the new subdomain solver and `Poisson1DD` that controls the global computation.

[1] The `Poisson1` solver can be found in the `$NOR/doc/Book/src/fem/Poisson1/` directory of a standard Diffpack distribution.

Definition of Class SubdomainPESolver

```
class SubdomainPESolver : public SubdomainFEMSolver, public Poisson1
{
protected:
  virtual void initSolField (MenuSystem& menu);
  virtual void createLocalMatrix ();
  virtual void buildGlobalCoarseDof ();
  virtual void createCoarseGridMatrix ();

public:
  SubdomainPESolver () {}
  virtual ~SubdomainPESolver () {}
  static  void defineStatic  (MenuSystem& menu, int level = MAIN);
};
```

Implementation of Class SubdomainPESolver

```
void SubdomainPESolver:: initSolField (MenuSystem& menu)
{
  // subd_fem_grid should already be sent in by SPAdmUDC
  grid.rebind (SubdomainFEMSolver::subd_fem_grid.getRef());

  // read info (except for grid) from menu
  menu.setCommandPrefix ("subdomain");
  A.redim (grid->getNoSpaceDim());
  A.scan (menu.get ("A parameters"));
  // database is to be given by Poisson1DD
  FEM:: scan (menu);
  lineq.rebind (new LinEqAdmFE);
  lineq->scan (menu);
  menu.unsetCommandPrefix ();

  // allocate data structures in the class:
  u.rebind    (new FieldFE(*grid,"u"));    // allocate, field name "u"
  error.rebind(new FieldFE(*grid,"error"));// another scalar field
  flux.rebind (new FieldsFE(*grid,"flux"));// vector field

  dof.rebind   (new DegFreeFE(*grid, 1));
  linsol.redim (dof->getTotalNoDof());
  uanal.rebind (new El1AnalSol(this)); // functor for analy. solution

  // connection between SubdomainFEMSolver and SubdomainPESolver
  subd_lineq.rebind (*lineq);
  subd_dof.rebind (*dof);
  global_solution_vec.rebind (linsol);

  // fill the DegFreeFE object with ess. BC, must be done here
  Poisson1:: fillEssBC ();
}

void SubdomainPESolver:: createLocalMatrix ()
{
  Poisson1:: makeSystem (*dof, *lineq);
  linsol.fill (0.0);        // set all entries to 0 in start vector
  dof->insertEssBC (linsol);// insert boundary values in start vector
}
```

```
void SubdomainPESolver:: buildGlobalCoarseDof ()
{
  SubdomainFEMSolver:: buildGlobalCoarseDof ();
  GridFE& coarse_grid = global_coarse_dof->grid();
  global_coarse_dof->initEssBC ();
  const int c_nno = coarse_grid.getNoNodes();
  Ptv(real) pt;
  for (int i=1; i<=c_nno; i++)
    if (coarse_grid.BoNode(i)) {
      coarse_grid.getCoor (pt,i);
      global_coarse_dof->fillEssBC (i,g(pt));
    }
}

void SubdomainPESolver:: createCoarseGridMatrix ()
{
  FEM:: makeSystem (csolver->coarse_dof(), csolver->coarse_lineq());
  SubdomainFEMSolver:: createCoarseGridMatrix (); // default work
}

void SubdomainPESolver:: defineStatic (MenuSystem& menu, int level)
{
  menu.addItem (level,          // menu level: level+1 is a submenu
                "A parameters", // menu command/name
                "A_1 A_2 ... in f expression", // help/description
                "3 2 1");       // default answer
  // submenus:
  LinEqAdmFE::defineStatic (menu, level+1);  // linear solver
  FEM::defineStatic (menu, level+1);         // num. integr. rule etc
}
```

Some Explanations. It is mandatory to redefine the `initSolField` function inside class `SubdomainPESolver`. First of all, it is needed for reading input from `MenuSystem` and constructing the subdomain data structure. Secondly, it is needed for coupling the external data objects with the generic data handles: `subd_lineq`, `subd_dof`, and `global_solution_vec`. The functions `defineStatic` and `createLocalMatrix` also need re-implementation in most applications. Thirdly, if coarse grid corrections are desired, the `createCoarseGridMatrix` function must be re-implemented. Part of this coding work can be done by re-using the default implementation from class `SubdomainFEMSolver`, see the above program. In addition, we also extend the default implementation of the `buildGlobalCoarseDof` function in this example, because we want to fill in the desired essential boundary conditions.

Definition of Class Poisson1DD

```
class Poisson1DD: public SPAdmUDC
{
protected:
  Handle(SaveSimRes) database;   // used by all the local solvers
  VecSimplest(Handle(SubdomainPESolver)) subd_poisson_solvers;
  virtual void createLocalSolver (int local_id);
```

```
  virtual void setupSimulators ();

public:
  Poisson1DD () {}
  virtual ~Poisson1DD () {}

  virtual void adm    (MenuSystem& menu);
  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void solveProblem ();  // main driver routine
  virtual void resultReport ();  // write error norms to the screen
};
```

Implementation of Class Poisson1DD

```
void Poisson1DD:: createLocalSolver (int local_id)
{
  SubdomainPESolver* loc_solver = new SubdomainPESolver;
  loc_solver->database.rebind (*database);

  subd_poisson_solvers.redim (num_local_subds);
  subd_poisson_solvers(local_id).rebind (*loc_solver);
  subd_fem_solvers(local_id).rebind (*loc_solver);
}

void Poisson1DD:: setupSimulators ()
{
  database.rebind (new SaveSimRes);
  database->scan (*menu_input, subd_fem_grids(1)->getNoSpaceDim());

  SPAdmUDC:: setupSimulators();  // do the default work
}

void Poisson1DD:: adm (MenuSystem& menu)
{
  define (menu);
  menu.prompt ();
  SPAdmUDC::scan (menu);
}

void Poisson1DD:: define (MenuSystem& menu, int level)
{
  SPAdmUDC:: defineStatic (menu,level+1);
  SaveSimRes:: defineStatic (menu, level+1);  // storage of fields

  menu.setCommandPrefix ("subdomain");
  SubdomainPESolver:: defineStatic (menu,level);  // for subd-solvers
  menu.unsetCommandPrefix ();
}

void Poisson1DD:: solveProblem ()
{
  // solution of the global linear system
  // DD either as stand-alone solver or precond
  SPAdmUDC:: solve ();

  // save results for later visualization
  SubdomainPESolver* loc_solver;
```

```
  for (int i=1; i<=num_local_subds; i++) {
    loc_solver = subd_poisson_solvers(i).getPtr();
    loc_solver->dof->vec2field (loc_solver->linsol, loc_solver->u());
    database->dump (loc_solver->u());
    loc_solver->makeFlux (loc_solver->flux(), loc_solver->u());
    database->dump (loc_solver->flux());
  }
}
```

Some Explanations. As class `Poisson1DD` now replaces class `Poisson1` as the main control class, the administration of the `SaveSimRes` object is transferred from `Poisson1` to `Poisson1DD`. Thus, a sub-menu for `SaveSimRes` is inserted into the `define` function of class `Poisson1DD`. The `setupSimulators` function is also modified to include the building-up of the `SaveSimRes` object. The re-implemented `createLocalSolver` function has also taken this to account. We note that class `Poisson1DD` has an additional array of subdomain simulator handles:

```
VecSimplest(Handle(SubdomainPESolver)) subd_poisson_solvers;
```

This handle array allows direct access to the public member functions and data objects that belong to class `Poisson1`.

We can see that the `solveProblem` function for this particular example is very simple. The reason is that the `scan` function of class `SPAdmUDC` has taken care of all the initialization work, including building up the subdomain linear systems. Recall that the global linear system is virtually represented by all the subdomain linear systems, the overall solution process is simply invoked by

```
SPAdmUDC:: solve ();
```

The Main Program. The following `main` program is primarily written for use on a parallel computer, this explains the appearance of the function pair `initDpParallelDD` and `closeDpParallelDD`, which belong to the parallel Diffpack library. However, as we have mentioned before, using the same code for a sequential DD solver can be achieved by simply running it on one processor.

```
#include <Poisson1DD.h>
#include <initDpParallelDD.h>

int main (int nargs, const char** args)
{
  initDpParallelDD (nargs, args);
  initDiffpack (nargs, args);
  global_menu.init ("Solving the Poisson equation","PDD approach");

  Poisson1DD udc;
  udc.adm (global_menu);
```

```
  udc.solveProblem ();
  udc.resultReport ();

  closeDpParallelDD();
  return 0;
}
```

An Input File for MenuSystem. An input file for the Diffpack `MenuSystem` may be composed as follows. Note that for this particular input file the overlapping DD method will be used as a preconditioner. To use it as a stand-alone iterative solver, it suffice to change the value of `domain decomposition scheme` to `BasicDDSolver`.

```
sub ParaPDESolver_prm
set domain decomposition scheme = KrylovDDSolver
sub LinEqSolver_prm
set basic method = ConjGrad
set max iterations = 200
ok
sub ConvMonitorList_prm
sub Define ConvMonitor #1
set #1: convergence monitor name = CMRelResidual
set #1: convergence tolerance = 1.0e-4
ok
ok
ok

sub SPAdmUDC
sub GridPart_prm
set grid source type = GlobalGrid
set global grid=P=PreproBox|d=2[0,1]x[0,1]|d=2e=ElmB4n2D[40,40][1,1]
set number overlaps = 1
ok
set use additive Schwarz = ON
set use coarse grid = ON
set global coarse grid=P=PreproBox|d=2[0,1]x[0,1]|d=2e=ElmB4n2D[4,4][1,1]
set degrees of freedom per node = 1
ok

set subdomain A parameters = 2 1
sub subdomain LinEqAdmFE
sub subdomain Matrix_prm
set subdomain matrix type = MatSparse
ok
sub subdomain LinEqSolver_prm
set subdomain basic method = ConjGrad
set subdomain max iterations = 100
ok
sub subdomain ConvMonitorList_prm
sub subdomain Define ConvMonitor #1
set subdomain #1: convergence monitor name = CMRelResidual
set subdomain #1: convergence tolerance = 1.0e-2
ok
ok
ok
ok
```

8.2 The Heat1 Example

We use the standard Diffpack sequential heat conduction simulator `Heat1`[2] as the starting point. The purpose of this example is to show the additional considerations that are required for incorporating an overlapping DD method into a simulator that solves a time-dependent PDE. Similar to the previous example, the coding effort is also in form of creating two new C++ classes: `SubdomainHeatSolver` and `Heat1DD`.

Definition of Class SubdomainHeatSolver

```
class SubdomainHeatSolver : public SubdomainFEMSolver, public Heat1
{
protected:
  virtual void initSolField (MenuSystem& menu);
  virtual void buildGlobalCoarseDof ();
  virtual void integrands (ElmMatVec& elmat, const FiniteElement& fe);

public:
  SubdomainHeatSolver () { make4coarsegrid = false; }
  virtual ~SubdomainHeatSolver () {}
  static  void defineStatic  (MenuSystem& menu, int level = MAIN);
  virtual void createLocalMatrix ();
  bool make4coarsegrid;
  virtual void createCoarseGridMatrix ();
};
```

Implementation of Class SubdomainHeatSolver

```
void SubdomainHeatSolver:: initSolField (MenuSystem& menu)
{
  // subd_fem_grid should already be sent in by SPAdmUDC
  grid.rebind (SubdomainFEMSolver::subd_fem_grid.getRef());

  // read info (except for grid and tip) from menu
  menu.setCommandPrefix ("subdomain");
  assignEnum(error_itg_pt_tp,menu.get("error integration point type"));
  diffusion_coeff = menu.get ("diffusion coefficient").getReal();
  FEM::scan (menu);
  uanal.rebind (new Heat1AnalSol (this));
  // 'database' and 'tip' are to be given by Heat1DD

  // fill handles with objects:
  u.     rebind (new FieldFE  (*grid, "u"));
  u_prev.rebind (new FieldFE  (*grid, "u_prev"));
  error. rebind (new FieldFE  (*grid, "error"));
  flux.  rebind (new FieldsFE (*grid, "flux"));
  u_summary.attach (*u);  u_summary.init ();

  dof.rebind(new DegFreeFE(*grid,1));
  lineq.rebind(new LinEqAdmFE);
  lineq->scan(menu);
```

[2] The `Heat1` simulator can be found in the `$NOR/doc/Book/src/fem/Heat1/` directory of a standard Diffpack distribution.

```
  linsol.redim(grid->getNoNodes());
  linsol.fill(0.0);
  menu.unsetCommandPrefix ();

  // connection between SubdomainFEMSolver and SubdomainHeatSolver
  subd_lineq.rebind (*lineq);
  subd_dof.rebind (*dof);
  global_solution_vec.rebind (linsol);

  // fill the DegFreeFE object with ess. BC, must be done here
  Heat1:: fillEssBC ();
}

void SubdomainHeatSolver:: buildGlobalCoarseDof ()
{
  SubdomainFEMSolver:: buildGlobalCoarseDof ();
  GridFE& coarse_grid = global_coarse_dof->grid();
  global_coarse_dof->initEssBC ();
  const int c_nno = coarse_grid.getNoNodes();
  Ptv(real) pt;
  for (int i=1; i<=c_nno; i++)
    if (coarse_grid.BoNode(i)) {
      coarse_grid.getCoor (pt,i);
      global_coarse_dof->fillEssBC (i,g(pt));
    }
}

void SubdomainHeatSolver:: integrands (ElmMatVec& elmat,
                                       const FiniteElement& fe)
{
  if (!make4coarsegrid)
    return Heat1::integrands (elmat, fe);

  const real detJxW = fe.detJxW();
  const int  nsd = fe.getNoSpaceDim();
  const int  nbf = fe.getNoBasisFunc();
  const real dt  = tip->Delta();
  const real t   = tip->time();
  real gradNi_gradNj;
  const real k_value = k (fe, t);
  int i,j,s;
  for(i = 1; i <= nbf; i++)
    for(j = 1; j <= nbf; j++) {
      gradNi_gradNj =0;
      for(s = 1; s <= nsd; s++)
        gradNi_gradNj += fe.dN(i,s)*fe.dN(j,s);

      elmat.A(i,j) +=(fe.N(i)*fe.N(j) +
                      dt*k_value*gradNi_gradNj)*detJxW;
    }
  // right-hand does not need to be filled here
}

void SubdomainHeatSolver:: createLocalMatrix ()
{
  Heat1:: makeSystem (*dof, *lineq);
}
```

```
void SubdomainHeatSolver:: createCoarseGridMatrix ()
{
  if (!make4coarsegrid) return;
  FEM::makeSystem (csolver->coarse_dof(), csolver->coarse_lineq());
  SubdomainFEMSolver::createCoarseGridMatrix (); // default work
}

void SubdomainHeatSolver:: defineStatic (MenuSystem& menu, int level)
{
  menu.addItem (level, "diffusion coefficient", "k", "1.0");
  menu.addItem (level, "error integration point type", "err_itg",
                "type of itg rule used in ErrorNorms::Lnorm",
                "GAUSS_POINTS", "S/GAUSS_POINTS/NODAL_POINTS");
  LinEqAdmFE::defineStatic(menu,level+1);
  FEM::defineStatic(menu,level+1);
}
```

Definition of Class Heat1DD

```
class Heat1DD: public SPAdmUDC
{
protected:
  Handle(TimePrm)    tip;      // time discretization parameters
  Handle(SaveSimRes) database; // used by all the local subd-solvers

  VecSimplest(Handle(SubdomainHeatSolver)) subd_heat_solvers;
  virtual void createLocalSolver (int local_id);
  virtual void setupSimulators ();

  virtual void timeLoop ();
  virtual void solveAtThisTimeStep ();

public:
  Heat1DD () {}
  virtual ~Heat1DD () {}

  virtual void adm    (MenuSystem& menu);
  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void solveProblem ();  // main driver routine
  virtual void resultReport ();  // write error norms to the screen
};
```

Implementation of Class Heat1DD

```
void Heat1DD:: createLocalSolver (int local_id)
{
  SubdomainHeatSolver* loc_solver = new SubdomainHeatSolver;
  loc_solver->tip.rebind (*tip);
  loc_solver->database.rebind (*database);

  subd_heat_solvers.redim (num_local_subds);
  subd_heat_solvers(local_id).rebind (loc_solver);
  subd_fem_solvers(local_id).rebind (loc_solver);
}
```

```
void Heat1DD:: setupSimulators ()
{
  // make 'tip' & 'database' ready for 'SPAdmUDC::setupSimulators'
  tip.rebind (new TimePrm);
  tip->scan(menu_input->get("time parameters"));
  database.rebind (new SaveSimRes);
  database->scan (*menu_input, subd_fem_grids(1)->getNoSpaceDim());

  SPAdmUDC:: setupSimulators();  // do the default work
}

void Heat1DD:: timeLoop()
{
  tip->initTimeLoop();

  // set the initial condition
  SubdomainHeatSolver* hsolver;
  int i;
  for (i=1; i<=num_local_subds; i++) {
    hsolver = subd_heat_solvers(i).getPtr();
    hsolver->setIC();
    database->dump (hsolver->u(), tip.getPtr(), "initial condition");
  }

  while(!tip->finished())
    {
      tip->increaseTime();
      solveAtThisTimeStep();
      for (i=1; i<=num_local_subds; i++) {
        hsolver = subd_heat_solvers(i).getPtr();
        hsolver->u_prev() = hsolver->u();
      }
    }
}

void Heat1DD:: solveAtThisTimeStep ()
{
  SubdomainHeatSolver* hsolver;
  int i;

  for (i=1; i<=num_local_subds; i++) {
    hsolver = subd_heat_solvers(i).getPtr();
    hsolver->fillEssBC ();
    hsolver->createLocalMatrix ();
    hsolver->modifyAmatWithEssIBC (); // must be called
    hsolver->dof->field2vec (hsolver->u(), hsolver->linsol);
    // only necessary to build the coarse grid matrix once
    if (use_coarse_grid && i==1) {
      hsolver->make4coarsegrid = true;
      hsolver->createCoarseGridMatrix ();
      hsolver->make4coarsegrid = false;
    }
  }

  SPAdmUDC:: solve ();   // solve the global linear system
  if (proc_manager->master()) {
    s_o << "t=" << tip->time();
```

```
    s_o << oform("  solver%sconverged in %3d iterations\n",
                 psolver->converged() ? " " : " not ",
                 psolver->getItCount());
    s_o.flush();
  }

  for (i=1; i<=num_local_subds; i++) {
    hsolver = subd_heat_solvers(i).getPtr();
    hsolver->dof->vec2field (hsolver->linsol, hsolver->u());
    database->dump (hsolver->u(), tip.getPtr());
    hsolver->u_summary.update (tip->time());
    hsolver->makeFlux (hsolver->flux(), hsolver->u());
    database->dump (hsolver->flux(), tip.getPtr(),
                    "smooth flux -k*grad(u)");
  }
}

void Heat1DD:: adm (MenuSystem& menu)
{
  define (menu);
  menu.prompt ();
  SPAdmUDC::scan (menu);
}

void Heat1DD:: define (MenuSystem& menu, int level)
{
  SPAdmUDC:: defineStatic (menu,level+1);
  menu.addItem (level, "time parameters", "step and domain for time",
                "dt =0.1 t in [0,1]");
  SaveSimRes::defineStatic (menu, level+1);

  menu.setCommandPrefix ("subdomain");
  SubdomainHeatSolver::defineStatic (menu,level); // for subd-solvers
  menu.unsetCommandPrefix ();
}

void Heat1DD:: solveProblem () { timeLoop(); }
```

Some Explanations. The `main` program and the `MenuSystem` input file are similar to those from the previous example, so we do not list them here. However, there are several new programming issues worth noticing. First, the explicit call of

```
u_prev->valueFEM(fe)
```

inside the original `Heat1::integrands` function makes it impossible to re-use the function in assembling the global coarse grid matrix. Therefore, we have to write a new version of the `integrands` function in class `SubdomainHeatSolver`, which makes use of a boolean variable `make4coarsegrid`, see the above code. Second, the `tip` handle is now controlled by class `Heat1DD`. This results in some extra code lines in the `createLocalSolver` and `setupSimulators` functions of class `Heat1DD`. Third, class `Heat1DD` also needs to re-implement the functions `timeLoop` and `solveAtThisTimeStep`. There is close resemblance between most

parts of these two functions and their counterparts in class `Heat1`. Fourth, re-generation of the virtually existing global linear system at each time level needs some attention. It is achieved on each subdomain by first creating the subdomain matrix $\mathbf{A}_i^0$ and then modifying it into $\mathbf{A}_i$, see Section 4.2. More precisely, the following two code lines are used:

```
hsolver->createLocalMatrix ();
hsolver->modifyAmatWithEssIBC ();
```

Note that we have assumed the most general situation, so the $\mathbf{A}_i^0$ matrix is re-generated and modified into $\mathbf{A}_i$ at every time level. In case the existing simulator `Heat1` only needs to modify the right-hand side vector, e.g., due to time-independent coefficients, then the above call to the `modifyAmatWithEssIBC` function should be replaced with

```
if (tip->getTimeStepNo()==1)
  hsolver->modifyAmatWithEssIBC ();
else {
  hsolver->orig_rhs() = (Vec(real)&)hsolver->subd_lineq->b();
  hsolver->global_orig_rhs() = hsolver->orig_rhs();
}
```

8.3 Some Programming Rules

To help the reader to extend a standard Diffpack simulator with overlapping DD functionality, we have summarized the following programming rules:

1. Two new C++ classes need to be created. The first class, e.g. with name `NewSubdSolver`, is to work as the new subdomain solver and should therefore be a subclass of both `SubdomainFEMSolver` and an existing simulator class. The second class is to work as the new global administrator and should be a subclass of `SPAdmUDC`.
2. The new subdomain solver must re-implement the `initSolField` function, which replaces the `scan` function of the existing simulator class and binds the handles of `SubdomainFEMSolver` to the correct data objects. Inside the `initSolField` function, the `subd_fem_grid` handle can be assumed to have already been bound to a subgrid covering the subdomain. It is also important to initialize the `subd_dof` object with correct essential boundary conditions, inside the `initSolField` function if applicable.
3. The new subdomain solver normally needs to have a new `defineStatic` function, which designs the `MenuSystem` layout part that is related to the subdomain solver, by including the necessary menu items from the original `define`/`defineStatic` function.
4. Normally, the new subdomain solver also needs to re-implement the `createLocalMatrix` function by e.g. utilizing the original `integrands` function and the generic `FEM::makeSystem` function.

5. If coarse grid corrections are desired, the functions `buildGlobalCoarseDef` and `createCoarseGridMatrix` normally need to be re-implemented in the new subdomain solver. The re-implemented functions normally make use of their default implementation inside class `SubdomainFEMSolver`. The following are two examples:

```
void NewSubdSolver:: buildGlobalCoarseDof ()
{
  // construct the coarse grid and its Dof
  SubdomainFEMSolver:: buildGlobalCoarseDof ();
  // fill the Dof object with essential BC if any
}

void NewSubdSolver:: createCoarseGridMatrix ()
{
  FEM::makeSystem (csolver->coarse_dof(),csolver->coarse_lineq());
  SubdomainFEMSolver::createCoarseGridMatrix ();
}
```

6. The new global administrator class should normally introduce an additional array of handles:
   ```
   VecSimplest(Handle(NewSubSolver)) subd_new_solvers;
   ```
   This is for accessing the non-generic functions of `NewSubSolver`.
7. The `createLocalSolver` function should be re-implemented in the new global administrator class.
8. The new global administrator class should also take over the responsibility for, e.g.,
   ```
   Handle(TimePrm) tip;
   Handle(SaveSimRes) data;
   ```
   Their corresponding menu items should be included in the `define` function of the new global administrator class. Normally, a slight modification of the `setupSimulators` function is also necessary. Re-implementing the default `SPAdmUDC::scan` function is, however, not recommended.
9. In case of solving a time-dependent PDE, the new global administrator class should also implement new functions such as `timeLoop` and `solveAtThisTimeStep`.

References

1. A. M. Bruaset, X. Cai, H. P. Langtangen, and A. Tveito. Numerical solution of PDEs on parallel computers utilizing sequential simulators. In Y. Ishikawa et al., editor, *Scientific Computing in Object-Oriented Parallel Environment, Springer-Verlag Lecture Notes in Computer Science 1343*, pages 161–168. Springer-Verlag, 1997.
2. X. Cai, E. Acklam, H. P. Langtangen, and A. Tveito. Parallel computing in Diffpack. In H. P. Langtangen and A. Tveito, editors, *Advanced Topics in Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*. Springer, 2003.

3. T. F. Chan and T. P. Mathew. Domain decomposition algorithms. In *Acta Numerica 1994*, pages 61–143. Cambridge University Press, 1994.
4. C. Farhat and M. Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *Internat. J. Numer. Meth. Engrg.*, 36:745–764, 1993.
5. W. Hackbusch. *Multigrid Methods and Applications*. Springer, Berlin, 1985.
6. K.-A. Mardal, H. P. Langtangen, and G. Zumbusch. Multigrid methods. In H. P. Langtangen and A. Tveito, editors, *Advanced Topics in Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*. Springer, 2003.
7. H. A. Schwarz. *Gesammelte Mathematische Abhandlungen*, volume 2, pages 133–143. Springer, Berlin, 1890. First published in Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich, volume 15, 1870, pp. 272–286.
8. B. F. Smith, P. E. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.
9. D. Vanderstraeten and R. Keunings. Optimized partitioning of unstructured finite element meshes. *Internat. J. Numer. Meth. Engrg.*, 38:433–450, 1995.
10. J. Xu. Iterative methods by space decomposition and subspace correction. *SIAM Review*, 34(4):581–613, December 1992.