WCCM V
Fifth World Congress on
Computational Mechanics
July 7–12, 2002, Vienna, Austria
Eds.: H.A. Mang, F.G. Rammerstorfer,
J. Eberhardsteiner

Developing Parallel Object-Oriented Simulation Codes in Diffpack

Xing Cai* **and Hans Petter Langtangen**

Simula Research Laboratory
P.O. Box 134, NO-1325 Lysaker, Norway
e-mail: {xingca,hpl}@simula.no
&
Department of Informatics
University of Oslo, P.O. Box 1080, Blindern, NO-0316 Oslo, Norway

Key words: object-oriented programming, generic PDE software, parallel computing

Abstract
This paper addresses the use of object-oriented programming techniques for developing simulation codes. We will argue why such modern programming techniques support the construction of flexible, maintainable and user-friendly software libraries for partial differential equations (PDEs). In particular, we will show some programming advantages associated with parallelizing object-oriented PDE software. First, parallelism can be incorporated into existing object-oriented PDE software libraries in a simple and flexible way. Second, object-oriented programming also simplifies the user effort needed for developing a parallel simulation code for a given PDE. The concrete programming issues will be demonstrated within the framework of Diffpack [1, 2], which is an object-oriented environment for scientific computing, with emphasis on the numerical solution of PDEs. The described programming paradigm and tools have been applied to numerous computational mechanics applications, including thermo-elasticity, incompressible viscous flow, and heat transfer.

1 Introduction

Non-numerical applications are nowadays often created using object-oriented programming techniques and languages like C++ and Java, which support this style of program development. The great advantage with object-oriented programming is information hiding and the ability to work with high-level abstractions. For a particular application area, one normally develops *class libraries* that offer the application programmer a high-level interface to problem solving. Each class typically contains data structures and member functions operating on the data structures. Very often the details of the data structures are hidden from a user of the class; the manipulation of the data can only take place through some high-level public member functions of the class. The application programmer declares *objects* of the library classes and develops the program as instantiation of objects and calls to the objects' public member functions.

Since the details of working with data structures can to a large extent be hidden inside the objects, the abstraction level tends to be significantly higher in object-oriented programs than in the traditional procedural programs. For example, there is no need to shuffle long lists of primitive variables back and forth between subroutines in object-oriented programs. It is sufficient to send objects, where each object may contain lots of variables. Adding a new variable is hence easy; just declare it as a data member in a class. The rest of the code will normally not notice such an extension.

Object-oriented programming is more than collecting data and functions inside classes. Classes can be related to each other through *inheritance* and organized as class hierarchies. This makes it possible to access different types of classes through a common (base class) interface. For example, one can have distinct classes for different finite element types, but access all of them through a generic element interface. This also helps to hide information and increase the abstraction level.

During the last decade, object-oriented programming has become more and more popular in the scientific computing community [3, 4, 5, 6, 7]. This is particularly true for the numerical solution of PDEs. One reason is that many components of a general numerical algorithm can be formulated in a generic fashion, independent of the specific PDEs. We can thus implement the generic components of the numerical algorithm as classes. A collection of such classes naturally forms a PDE software library. Object-oriented programming also enables organizing similar numerical methods that share common features into a class hierarchy. This is very advantageous for code maintenance and extension. Therefore, object-oriented programming strongly promotes code re-use and greatly increases human efficiency. For example, computational mechanics codes can utilize well-tested PDE software developed for other application areas. Examples of existing object-oriented PDE software libraries are Cogito [8], Diffpack [2], Overture [9], ParallelFEMCore [10], and PETSc [11], just to mention a few. When writing a specific simulation code, the coding work of a programmer, who uses object-oriented PDE software libraries, is normally reduced to designing a few small-size classes for treating the problem-specific details. The main body of computation is done by using the generic functionality of the PDE libraries.

Computing platforms with multiple processors are an indispensable tool for running large-scale simulations. Such parallel computers require specially adapted parallel software. In respect of parallel programming, *message-passing* has established itself as the most versatile paradigm for developing portable parallel simulation codes. Despite the widely used MPI standard [12], writing parallel simulation codes is still a demanding task. Due to the need of inserting MPI routines into serial codes, it is not uncommon that serial PDE software libraries undergo extensive re-programming to incorporate parallelism. However, if object-oriented programming is applied appropriately, the extra programming work for parallelization can be limited. It is in fact possible to collect all the parallel computing specific codes into

a small-size object-oriented parallel "toolbox". Thus, for an existing serial object-oriented PDE library, it suffices to only incorporate an interface to the toolbox. The modification of the serial library is very small, and all its serial functionality remains intact. The result is that parallelization of the standard linear algebra operations can be enabled (almost) automatically. For more advanced parallel numerical algorithms, such as domain decomposition (DD) methods [13, 14, 15], object orientation can help to produce user-friendly implementation frameworks. This will be exemplified in Section 4.

Overview of the Paper. In Section 2, we first explain why object-oriented programming is advantageous for developing PDE software libraries. This is followed shortly by a brief introduction to some important classes of Diffpack, which can be used to build serial PDE solvers. We turn our attention to parallelizing PDE solvers in Section 3, where we explain two parallelization approaches. Then, we present in Section 4 the object-oriented implementation of the two parallelization approaches within Diffpack. Illustrations of the two approaches for a nonlinear heat conduction problem are given in Section 5. Finally, we present some concluding remarks in Section 6.

2 Generic PDE Software and Diffpack

2.1 Components of a General Numerical Algorithm

A good design of a PDE software library relies on the dissection of general numerical algorithms into *generic* components. We start with a list of such components that are of interest for the topic in the present paper. Let us consider a time-dependent nonlinear PDE, e.g., the following nonlinear heat conduction equation:

$$\frac{\partial u}{\partial t} = \nabla \cdot [\lambda(u)\,\nabla u]\,, \qquad x \in \Omega \subset \mathbb{R}^d, \quad t > 0. \tag{1}$$

Although the above nonlinear PDE is very simple in its mathematical form, its general numerical algorithm, consisting of the following main components, is also applicable for other more complex PDEs.

1. A spatial computational grid $\mathcal{T}$.
2. A time-stepping scheme that discretizes the PDE in the temporal direction, e.g.,

$$\frac{u^l - u^{l-1}}{\Delta t} = \nabla \cdot \left[\lambda(u^l)\,\nabla u^l\right], \tag{2}$$

   where the superscript l denotes the number of time levels.
3. At each time level, a spatial discretization of (2) on $\mathcal{T}$, resulting in a system of nonlinear algebra equations:

$$\boldsymbol{F}(\boldsymbol{u}^l) = 0, \tag{3}$$

   where $\boldsymbol{F} = (F_1, F_2, \ldots, F_n)$ and $\boldsymbol{u}^l = (u_1^l, u_2^l, \ldots, u_n^l)$ is the solution vector, with n being the number of unknowns.
4. At each time level, a nonlinear system solver for (3), which generates, for an initial guess $\boldsymbol{u}^{l,0}$, a sequence of approximate solutions: $\boldsymbol{u}^{l,1}$, $\boldsymbol{u}^{l,2}$, ... until convergence at that time level.

5. A linear system solver that is used to solve a linear system in the form of

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{4}$$

   in each internal step of the above nonlinear system solver.

It should be noticed that the above general numerical algorithm should be supplemented with appropriate initial and boundary conditions. In case of a linear PDE, component number 3 directly gives rise to a system of linear equations, so component number 4 is not needed.

2.2 Generic PDE Software Using Object-Oriented Programming

In fact, all the components of the preceding numerical algorithm allow different choices.

- Should we use finite differences, finite elements or finite volumes for the discretization in component number 3?
- What nonlinear solver should we use in component number 4? Possible choices are, e.g., successive substitution and Newton-Raphson.
- What linear solver should we use in component number 5? Should we use a direct solver such as Gauss elimination or an iterative solver such as preconditioned BiCGStab (see e.g. [16])?

In addition, the spatial grid $\mathcal{T}$ should be compatible with the chosen discretization method, and we should allow using other temporal discretization schemes (e.g., using the θ-rule) in component number 2.

We remark that each of the components normally has a quite complex algorithmic structure and involves many different parameters. In order to handle the consequent programming difficulties and build a flexible PDE software library more efficiently, we can resort to object-oriented programming. First, an abstract data type–class–can be designed for each generic component. The class embeds many of the low-level details and provides a set of member functions as the algorithmic interface. Programming with the generic classes mimics more closely the underlying numerics. Second, the use of class inheritance and virtual member functions allows the construction of class hierarchies that comprise groups of numerical methods with common functionality and similar algorithmic features. For example, a class named `LinEqSolver` can be introduced as the generic representation of all the linear solvers for (4). The most useful member function of class `LinEqSolver` should be `solve`, which takes objects representing $\boldsymbol{A}$ and $\boldsymbol{b}$ as input and produces an object for the solution vector $\boldsymbol{x}$. The `solve` function needs only to be declared as `virtual` without having a concrete implementation. Moreover, class `LinEqSolver` can derive `DirectSolver` and `IterativeSolver` as two subclasses that represent the general direct and iterative linear system solvers, respectively. Further down in the `LinEqSolver` hierarchy tree, we can implement `GaussElim` as a subclass of `DirectSolver` and `BiCGStab` as a subclass of `IterativeSolver`. In these two classes, the virtual member function `solve` should be re-implemented according to the numerics. Other concrete linear system solvers can be implemented and incorporated into the `LinEqSolver` hierarchy similarly. The result is that for other parts of the PDE library, it suffices to work with objects of type `LinEqSolver`, whereas the choice of a concrete linear system solver can be delayed until runtime. As a third point, the programming effort needed for writing a simulation code for a specific PDE using the generic PDE library is quite limited. This is because the simulator is normally a subclass of some generic class in the library. Most of the code needed in the simulator is inherited from the library, and the programmer is required to just re-implement some problem-specific virtual member functions.

2.3 Diffpack

Diffpack (see [1, 2]) is an object-oriented environment for scientific computing, with emphasis on the numerical solution of PDEs. A collection of generic class libraries, written in the C++ programming language, constitute the main content of Diffpack. In order for the readers to understand the implementation issues about parallelizing Diffpack simulation codes in Sections 4 and 5, we give here a brief introduction to Diffpack's most important generic classes that are used for serial finite element computation.

- Class `GridFE` represents a computational finite element grid.
- Classes `FieldFE` and `FieldsFE` represent a scalar and a vector field, respectively. Internally, each of the two class has a pointer to one `GridFE` object and array objects to hold nodal values. The field classes offer, of course, member functions for interpolating the fields at an arbitrary point.
- Class `TimePrm` provides the functionality needed for a time-stepping scheme.
- Class `NonLinEqSolver` is a base class representing a generic nonlinear system solver. Two of its subclasses are `SuccessiveSubst` and `NewtonRaphson`. To enable the use of `NonLinEqSolver` in a simulator class, we have to first derive the simulator class as a subclass of the generic interface class `NonLinEqSolverUDC`, where `UDC` stands for user-defined code. Then, inside the simulator class we do the following three things:

  1. Re-implement the virtual member function `makeAndSolveLinearSystem` that belongs to class `NonLinEqSolverUDC`. This function carries out the work of the fourth and fifth component mentioned in Section 2.1.
  2. Instantiate and point to an object of a chosen subclass of `NonLinEqSolver`. We use, e.g., `nlsolver` as the name of the object pointer. The type of the pointer is `NonLinEqSolver`, i.e., we hide the information about which iteration method we actually use.
  3. Invoke `nlsolver->solve()` when (3) needs to be solved.

- Class `FEM` is the generic class representation of all PDEs that are to be solved by the finite element method. When treating a specific PDE, the user creates the simulator as a subclass of `FEM` and essentially only implements the virtual member function `integrands`. This function is supposed to calculate the contribution of one numerical integration point to the element matrix/vector, following the finite element weak form of the specific PDE (see [2, Ch. 3.1]). For assembling the linear system (4), the user normally just invokes the ready implemented function `FEM::makeSystem`, which goes through all the elements one by one and calls `integrands` internally for each of the numerical integration points. In case the problem at hand allows special optimizations of the assembly process to take place, the `makeSystem` function can be re-implemented in an optimized form in the simulator class.
- Class `LinEqAdmFE` is a generic control class for solving a linear system in the form of (4). Class `LinEqAdmFE` accesses the matrix $\boldsymbol{A}$ and right-hand side vector $\boldsymbol{b}$ through an internal pointer to an object of type `LinEqSystem`. The other important internal pointer inside `LinEqAdmFE` points to an object of type `LinEqSolver`. As mentioned in Section 2.2, the particular choice of the linear solver needs only to be decided at runtime. The user can also easily control other issues such as convergence monitoring and preconditioning through `LinEqAdmFE`. In a Diffpack program, all a user needs to do is invoking `LinEqAdmFE::solve`.

- Class `DegFreeFE` provides a mapping between the degrees of freedom of a field to the degrees of freedom of a linear system.

In addition, there are two more issues worth mentioning. First, a class with name `Handle(X)` is a "smart" pointer with internal reference counting functionality to objects of type `X`. These handles are used extensively throughout the Diffpack libraries. Second, Diffpack has a flexible menu system tool for providing values for diverse parameters needed for a simulation. The Diffpack programming standard demands each class to contain a pair of member functions: `define` and `scan`. The `define` function determines the "menu items" to appear in the menu system, whereas the `scan` function loads user-given menu answers into the variables in the class. The "appearance" of a menu system interface can be chosen at run time, either as a graphical user interface, a command-line interface, or simply commands stored in a file.

Solving a specific PDE is a matter of deriving a subclass of `FEM`, implementing the weak form in the `integrands` function, declaring variables to hold physical and numerical parameters, initializing these variables through the menu system, enforcing initial conditions (if present), and writing a managing routine. Normally, the length of the code is a few pages.

A Sample Diffpack Simulator. For the nonlinear heat conduction equation (1), a Diffpack simulator with name `NlHeat1` may have the following class declaration:

```
class NlHeat1 : public FEM, public NonLinEqSolverUDC
{
protected:
  Handle(GridFE)     grid;    // finite element grid
  Handle(DegFreeFE)  dof;     // field <-> linear system mapping
  Handle(FieldFE)    u;       // primary unknown
  Handle(FieldFE)    u_prev;  // u at previous time step
  Handle(TimePrm)    tip;     // time step etc
  Handle(LinEqAdmFE) lineq;   // linear system & solution
  Handle(NonLinEqSolver) nlsolver;
  // other internal data items ...

  virtual void setIC();       // treating the initial condition
  virtual void solveAtThisTimeStep();  // work per time step
  virtual void makeAndSolveLinearSystem ();
  virtual void integrands (ElmMatVec& elmat, const FiniteElement& fe);
  virtual real lambda  (real u);  // nonlinear coefficient
  // other protected member functions ...

public:
  NlHeat1 ();
 ~NlHeat1 () {}

  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan   ();
  virtual void solveProblem ();
  // other public member functions ...
};
```

We notice from the above class declaration that `NlHeat1` is a subclass of both `FEM` and `NonLinEqSolverUDC`. The simulator class mainly offers three public member functions for carrying out the needed operations. As mentioned earlier, the function pair `define` and `scan` provides a flexible mechanism of giving values to all the needed parameters at runtime. Besides, the `scan` function is also responsible for instantiating the internal data objects and binding handles to them. The main work of simulation is done by the `solveProblem` function, whose implementation looks like:

```
void NlHeat1:: solveProblem ()
{
  tip->initTimeLoop();
  setIC();                    // enforce the initial condition
  while(!tip->finished())
   {
     tip->increaseTime();     // update time for the next step
     solveAtThisTimeStep ();  // solve spatial problem by the FEM
     *u_prev = *u;            // update fields for next step
     // auxiliary operations such as data storage etc.
   }
}
```

As shown above, the computation per time step is done by the `solveAtThisTimeStep` function, whose actual work is done in the libraries by the following command:

```
  nlsolver->solve ();  // solve nonlinear algebraic system
```

Re-implementing the two virtual generic member functions `makeAndSolveLinearSystem` and `integrands` are mandatory. We list `NlHeat1::makeAndSolveLinearSystem` in the following, while referring to [2, Ch. 4.2.1] for the details of `NlHeat1::integrands`.

```
void NlHeat1:: makeAndSolveLinearSystem ()
{
  // some small preparation work
  FEM::makeSystem (*dof, *lineq); // create linear system Ax=b
  lineq->solve();                 // invoke a linear system solver
}
```

Finally, we remark that the class name `NlHeat1` indicates that this simple simulator can serve as a base class for deriving new subclasses that treat more complex nonlinear heat conduction equations.

3 Two Parallelization Approaches

3.1 Grid Partitioning and Distributed Matrices and Vectors

Let us review the general numerical algorithm from Section 2.1. We notice that the most computation-intensive operations are, viewed at the level of matrices and vectors, (i) calculation of element matrices and vectors, and (ii) solution of the resulting linear system (4). Following the most flexible parallel computer model of *distributed memory*, we start the incorporation of parallelism with partitioning the global grid $\mathcal{T}$ into a set of P smaller subgrids $(\mathcal{T}_i)_{i=1}^{P}$, which are to be hosted by different processors of a parallel computer. From now on, we will only consider numerical algorithms that use finite element discretizations and iterative solvers for (4).

Partitioning a global finite element grid primarily concerns dividing all elements into subgroups. The elements of the global finite element grid $\mathcal{T}$ are partitioned to form a set of smaller subgrids $\mathcal{T}_i$, $1 \leq i \leq P$, where P is typically the number of available processors. By a *non-overlapping partition*, we mean that each element of $\mathcal{T}$ belongs to only one of the subgrids. Element edges shared between neighboring subgrids constitute the so-called *internal boundaries*. For subgrid $\mathcal{T}_i$, we use Γ_i to denote its internal boundary. We note that although every element of $\mathcal{T}$ belongs to a single subgrid after a non-overlapping partition, the nodes lying on the internal boundaries belong to multiple subgrids at the same time. In Figure 1, the nodes marked with a circle denote such nodes.

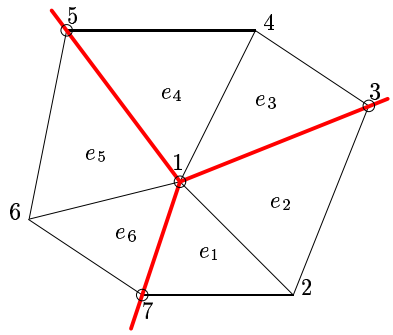

Figure 1: An example of internal boundary nodes that are shared between neighboring subgrids. The figure shows a small region where three subgrids meet. Elements e_1 and e_2 belong to one subgrid, e_3 and e_4 belong to another subgrid, while e_5 and e_6 belong to a third subgrid. Nodes marked with a circle are internal boundary nodes.

When a non-overlapping partitioning of $\mathcal{T}$ is ready, one subdomain is to be held by one processor. The discretization of the target PDE on one processor is done in the same way as in the serial case, only that the discretization is restricted to the assigned subdomain. The assembly of the local matrix and right-hand side vector in (4) needs *only* contribution from all the elements belonging to its own subgrid. Therefore, no inter-processor communication is necessary during the discretization. It is important to note that *there is no need to construct global matrices/vectors physically*. It suffices to deal only with local matrices and vectors that are associated with the subgrids. For solving a virtually existed global linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ by a parallel iterative solver, we can operate with only local matrices and vectors plus using inter-processor communication, as Section 3.2 will show.

3.2 Linear Algebra Level Parallelization

3.2.1 Three Types of Parallel Linear Algebra Operations

During a parallel solution of (4), the global matrices and vectors ($\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{x}$) do not need physical storage. They can be represented virtually by the local matrices and vectors ($\boldsymbol{A}_i$, $\boldsymbol{b}_i$, $\boldsymbol{x}_i$), which are held by the different processors. Iterative linear system solvers suit well for this situation, because it suffices to parallelize the following three types of global linear algebra operations: (i) vector addition: $\boldsymbol{w} = \boldsymbol{u} + \boldsymbol{v}$; (ii) inner-product between two vectors: $c = \boldsymbol{u} \cdot \boldsymbol{v} \equiv \sum_j u_j v_j$; (iii) matrix-vector product: $\boldsymbol{w} = \boldsymbol{A}\boldsymbol{u}$.

Parallel Vector Addition. The parallel vector addition $\boldsymbol{w} = \boldsymbol{u} + \boldsymbol{v}$ is straightforward and needs no inter-processor communication. The only assumption is that values of $\boldsymbol{u}_i$ and $\boldsymbol{v}_i$ are correctly duplicated on the internal boundary nodes on all the neighboring subdomains.

Parallel Inner-Product between Two Vectors. Computing the inner-product between two global vectors, which are distributed on different processors, can start with computing the local computation $c_i = \sum_j u_{i,j} v_{i,j}$. However, the c_i values can not just be added together to produce c, because internal boundary nodes are shared between multiple subgrids. The contribution from those internal boundary nodes must be scaled accordingly. For this purpose, we denote by $\mathcal{O}_i$ the set of all the internal boundary nodes of $\mathcal{T}_i$. In addition, each internal boundary node has an integer count o_k, $k \in \mathcal{O}_i$, which is the total

number of subgrids it belongs to, including $\mathcal{T}_i$ itself. Then, we can get the adjusted local result by

$$\tilde{c}_i \quad = \quad c_i - \sum_{k \in \mathcal{O}_i} \frac{o_k - 1}{o_k} u_{i,k} v_{i,k} \, .$$

Thereafter, the correct global result of $c = \boldsymbol{u} \cdot \boldsymbol{v}$ can be obtained by collecting all the adjusted local results in the form of $c = \sum_i^P \tilde{c}_i$. This is done by inter-processor communication in the form of an all-to-all broadcast operation.

Parallel Matrix-Vector Product. We recall that a global matrix $\boldsymbol{A}$ is now represented collectively by the local matrices $\boldsymbol{A}_i$. These local matrices arise from a local finite element discretization. The rows of $\boldsymbol{A}_i$ that correspond to the interior subgrid nodes are the same as those of $\boldsymbol{A}$. A local matrix-vector product $\boldsymbol{w}_i = \boldsymbol{A}_i \boldsymbol{u}_i$ will thus give correct values of $\boldsymbol{w}_i$ for the interior subgrid nodes. However, the rows of $\boldsymbol{A}_i$ that correspond to the internal boundary nodes have only partially correct values. More specifically, the correct value for an entry of $\boldsymbol{w}_i$, which is associated with an internal boundary node, can only be obtained by adding up contributions from *all* the neighboring subdomains. Take Figure 1 for instance, nodes with global numbers 3,5,7 need contributions from two subdomains, while node with global number 1 should add together contributions from all the three subdomains. This requires inter-processor communication in the form of two and two neighboring subdomains exchange values on the relevant internal boundary nodes.

3.2.2 Parallel Preconditioning

Preconditioners are often needed for speeding up iterative linear system solvers. Parallel linear solvers require parallel preconditioning operations. However, many standard preconditioners such as Gauss-Seidel relaxation, and RILU are inherently serial algorithms. A simple remedy is to just let each processor run the corresponding localized preconditioning operations, and then take some kind of average of the different values associated with each internal boundary node from the neighboring subgrids. Of course, such a remedy will normally result in weakened preconditioning effect. More powerful parallel preconditioners such as the overlapping DD methods will therefore be studied in the next section.

3.3 Simulator Level Parallelization

3.3.1 The Mathematical Method of Overlapping DD

The starting point of the so-called overlapping DD methods [13, 14, 15] is a set of overlapping subgrids. Based on the non-overlapping partition of $\mathcal{T}$ from Section 3.1, we can expand the subgrids slightly, so that every internal boundary node on Γ_i becomes an interior node of at least one neighboring subgrid. We shall only consider the additive version of the overlapping DD methods, because this version is most straightforward for parallelization. When solving a global linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ that is distributed on a set of overlapping subgrids, the additive DD method constitutes an iterative process. In iteration number $n+1$, when $\boldsymbol{x}^n$ is already computed and exists distributedly as ($\boldsymbol{x}_1^n$,$\boldsymbol{x}_2^n$,...,$\boldsymbol{x}_P^n$), the additive DD method proceeds by

$$\boldsymbol{x}^{n+1} = \boldsymbol{x}^n + \sum_{i=1}^{P} \boldsymbol{R}_i^T \boldsymbol{A}_i^{-1} \boldsymbol{R}_i (\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^n), \tag{5}$$

where the restriction matrix $\boldsymbol{R}_i$, which contains only ones and zeros, is used for "clipping out" the entries of a global vector that are associated with subgrid $\mathcal{T}_i$. The transpose matrix $\boldsymbol{R}_i^T$ is for interpolating a subgrid solution to a global solution. Parallelism is inherent with the above method because the subdomain solves $\boldsymbol{A}_i^{-1}$ can be carried out *completely independently*. We also mention that since solutions $\boldsymbol{x}^n$ and $\boldsymbol{x}^{n+1}$ are always distributed, there is no need to physically form $\boldsymbol{R}_i$ and $\boldsymbol{R}_i^T$ in an actual implementation; they are only needed in (5) for the mathematical correctness. To use (5) as a preconditioner, we just need to run one iteration with $\boldsymbol{x}^0 = 0$ for a given right-hand side vector.

3.3.2 The Simulator-Parallel Approach

There are two more reasons for our interest in the additive DD method (5), besides its inherent parallelism. The first reason is that such a DD solver/preconditioner, when coupled with coarse grid corrections (see [13, 14, 15]), normally has very good convergence properties. The second reason lies in the programming respect. We notice that the subdomain problems of (5) arise from discretizing the *same* PDE as the original global problem. This means that we should, in principle, be able to *re-use* an existing serial PDE simulator, which is designed for the original PDE, also as the subdomain solver for each subdomain. Of course, an administrator of the subdomain solvers on the global level has to be implemented somehow, whereas the needed inter-processor communication can be handled in the same way as in Section 3.2. Our goal is to implement the global administrator and the needed communication functionality in a generic fashion, independent of specific PDEs. We call this generic approach to programming parallel DD solvers/preconditioners as *simulator-parallel* [17]. In Section 4.3, we will show how object-oriented programming helps building a generic implementation framework.

4 Object-Oriented Implementation of the Parallelization Approaches

We present in this section a small object-oriented library that implements the two parallelization approaches. This parallel "toolbox" allows the user to parallelize serial Diffpack simulators in a flexible and structured way.

4.1 Coupling with the Serial Diffpack Libraries

One of our objectives when designing the parallel toolbox is to keep the huge serial libraries of Diffpack unchanged to as large extent as possible. The only new class that we have introduced into the serial Diffpack libraries is a light-weight interface class with name `SubdCommAdm` (subdomain communication administrator). This class declares functions for updating global and interior boundary values in distributed vectors. It also declares parallelized versions of basic matrix-vector operations, including matrix-vector products, and inner-products.

All the member functions of class `SubdCommAdm` are declared as `virtual` and have empty implementation. Therefore, `SubdCommAdm` only serves as a pure interface class, defining the name and syntax of different functions for inter-processor communication. Various subclasses of `SubdCommAdm` implement, e.g., the parallel matrix-vector operations according to a specific type of data distribution.

It suffices for the rest of the serial Diffpack libraries to only know how inter-processor communication can be invoked, and this information is provided by class `SubdCommAdm`. Hence, in classes that involve matrix-vector operations we include a `SubdCommAdm` pointer (handle). If this pointer is non-zero, i.e. the computations are performed in parallel, the appropriate matrix-vector operation is called through the

SubdCommAdm object. Otherwise, i.e. in serial calculations, we just issue a standard matrix-vector operation. In this way, the original serial Diffpack libraries can support parallel linear algebra through modest modifications. Since the parallel computing and communication strategy is hidden by the SubdCommAdm class, the library code is free of, e.g., long low-level MPI calls, which is the common result of parallelizing simulation codes.

4.2 Functionality for Linear Algebra Level Parallelization

Two main tasks of the parallel toolbox are to provide Diffpack users with different grid partitioning methods and several high-level inter-processor communication functions. In addition, the toolbox also implements parallel versions of some of Diffpack's most frequently used linear algebra operations.

A Hierarchy of Grid Partitioning Methods. Class GridPart is a generic grid partitioner. Its subclasses implement specific methods for dividing a grid into subgrids: GridPartUnstruct handles unstructured grids, GridPartUniform handles uniform rectangular or box-shaped grids, and GridPartFileSource reads subgrid specifications from a set of grid files.

The Main Control Class. We recall that SubdCommAdm is a pure interface class; all its inter-processor communication functions need to be re-implemented in a concrete subclass. For this purpose, we have created GridPartAdm as a concrete subclass of SubdCommAdm in the parallel toolbox. Thus, object-oriented programming sets SubdCommAdm as the formal connection between the existing serial Diffpack libraries and the parallel toolbox, while GridPartAdm is the actual working unit. It should also be mentioned that class GridPartAdm not only re-implements the different inter-processor communication functions, but at the same time provides a unified access to the different methods offered by the GridPart hierarchy. The member function prepareSubgrids is designed for this purpose. The relationship between SubdCommAdm, GridPartAdm, and GridPart is depicted in Figure 2. Moreover, a member function named prepareCommunication is designed for building up the internal data structure needed for later communication operations inside an object of SubdCommAdm. Therefore, it suffices for a user to only work with GridPartAdm when parallelizing of a serial Diffpack simulator.

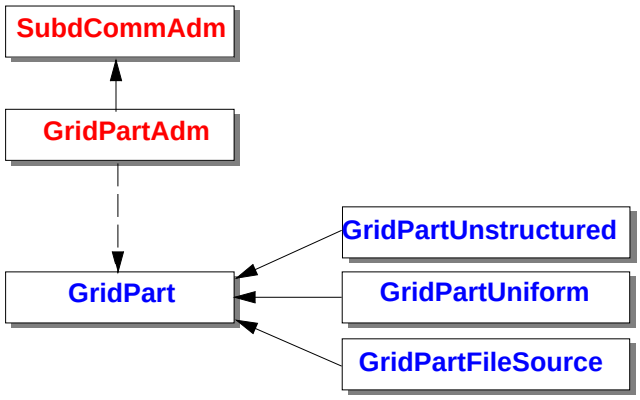

Figure 2: The relationship between SubdCommAdm, GridPartAdm, and GridPart. Note that the solid arrows indicate the "is-a" relationship, while the dashed arrows indicates the "has-a" relationship.

4.3 A Generic Implementation Framework for Parallel Overlapping DD Methods

As we have mentioned in Section 3.3.2, an existing serial simulator should be considered for re-use as the subdomain solver in a parallel overlapping DD solver/preconditioner. We want an implementation framework for parallel overlapping DD methods, where the non-PDE-specific components are provided as generic C++ library classes. In such a framework, object-oriented programming should be used to simplify the coding effort of a user, such as slight extension/modification of the existing serial simulator to fit with the generic components of the implementation framework. We present in the following a generic implementation framework made up of three parts: the subdomain solvers, a communication part, and a global administrator.

4.3.1 Subdomain Solvers

A typical Diffpack PDE simulator has functionality for generating the discrete equations 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` as a generic subdomain solver. A concrete subdomain solver must be derived as a subclass of *both* `SubdomainFEMSolver` *and* the existing simulator. The purpose of class `SubdomainFEMSolver` 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. The virtual member functions have all been given a default implementation. The only exception is member function `initSolField`, which must be explicitly re-implemented in a derived subclass. This function is used to bind the generic data handles of `SubdomainFEMSolver` to the concrete data objects in the existing simulator.

4.3.2 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`.

4.3.3 The Global Administrator

The global administrator of the implementation framework is mainly constituted by two generic classes. The function of the global administrator includes construction of the subdomain solvers and the communication part, plus controlling all the operations during each DD iteration.

Class ParaPDESolver. Overlapping DD methods can work either as stand-alone iterative solvers or as preconditioners for Krylov subspace methods. By using object-oriented programming, 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 SPAdmUDC. The other major class of the global administrator is `SPAdmUDC`, which controls parameter input and the creation of a desired DD solver (i.e., an object of either `BasicDDSolver` or `KrylovDDSolver`). In addition, class `SPAdmUDC` also controls the communication part and the subdomain solvers. We remark that "`UDC`" (user-defined code) is used to indicate that the user has the possibility of re-implementing `SPAdmUDC`'s member functions and introducing new functions when he derives a concrete new subclass.

Class `SPAdmUDC` has one member function with name `createLocalSolver` that *must* be re-implemented in a derived subclass. The purpose of this member function is to instantiate a local solver object on every subdomain. Normally, this is just a one-line call that creates an object of the modified PDE simulator (subclass of `SubdomainFEMSolver`). The advantage of having a simulator available as an object, i.e. a variable in the program, should be obvious in a setting where we need many simulators, each working on a subdomain.

4.3.4 Coding Effort

When using the framework for implementing an overlapping DD method, the user primarily needs to carry out a coding task in the form of deriving two small-size C++ subclasses. One C++ subclass (from `SubdomainFEMSolver`) is for modifying and extending an existing PDE simulator so that it can work as a subdomain solver, whereas the other C++ subclass (from `SPAdmUDC`) is for controlling the global calculation and coupling the subdomain solver with the other components of the implementation framework.

5 Parallelizing Serial Simulators

In this section, we will apply the two parallelization approaches to the serial Diffpack nonlinear heat conduction simulator `NlHeat1` described in Section 2.3.

5.1 Linear Algebra Level Parallelization

This parallelization approach requires only a small adjustment of the serial source code of the `NlHeat1` simulator. Basically, the `NlHeat1` class must be extended with a `GridPartAdm` object. All the programmer has to do is including the `GridPartAdm` header file, declaring a `GridPartAdm` pointer (handle), adding a `GridPartAdm` submenu to the menu system, initializing the `GridPartAdm` object from the menu system, and finally attaching the `GridPartAdm` object to the linear system toolbox (`LinEqAdmFE`). The linear system and solver are thus notified about parallel computing. Iterative solvers, like Krylov methods for (non-)symmetric matrix system, will therefore run in parallel. We refer to [18] for more details.

5.2 Simulator Level Parallelization

The above parallelization process is very simple and user friendly. However, the only disadvantage with the resulting parallel simulator is the absence of a parallel preconditioner. In case the need for an efficient parallel preconditioner arises, one should consider parallelizing `NlHeat1` using the DD-based parallelization approach. In this case, we need to develop two new classes:

```
class SubdomainNlHeat1 : public SubdomainFEMSolver, public NlHeat1
class NlHeat1DD : public SPAdmUDC, public NonLinEqSolverUDC
```

Class `SubdomainNlHeat1` works as solver on a subdomain, whereas class `NlHeat1DD` administers the global DD iterations or a Krylov solver with DD as preconditioner. The coding effort consists of re-implementing a few virtual member functions of `SubdomainFEMSolver` and `SPAdmUDC`. In addition, class `NlHeat1DD` needs to implement its own `solveProblem`, `solveAtThisTimeStep`, and `makeAndSolveLinearSystem` (see Section 2.3). This is for shifting the control of computation from `NlHeat1` to `NlHeat1DD`. Reference [19] provides more details and examples.

6 Concluding Remarks

Object-oriented programming is a powerful tool for developing flexible and re-usable serial PDE software libraries. We have explained in this paper that a serial library, having a suitable object-oriented design, can easily be parallelized and equipped with tools supporting rapid parallelization of serial simulators. Actually, our programming paradigm allows development of computational mechanics applications in Diffpack without paying attention to parallel computing. Thus, the application developer can focus on the physical problem being solved, the numerics, and a thorough verification of the code. When the serial code works well, it can be extended for parallel computing by a few additional lines or by adding two extra small classes. This stepwise program development reduces the debugging time of parallel codes significantly.

Of course, the ease of parallelization described here is closely connected to the chosen methods for parallel computing. The linear algebra level approach requires the application developer to rely on parallel finite element assembly and parallel Krylov solvers, with suboptimal parallel preconditioners. The advantage is that a serial simulator can be parallelized with about 10 new statements. The disadvantage is the efficiency of the numerics. A better approach from an efficiency point of view is to apply DD as preconditioner (or stand-alone solver). This second parallelization strategy requires somewhat more programming by the application developer. The performance of the approach is problem dependent, but our experience shows that close-to-optimal speed-up results are often obtained.

The outlined parallelization approaches have been successfully applied in the parallelization of numerous Diffpack simulators, including many computational mechanics applications. We have, for example, created parallel solvers for linear thermo-elasticity, the incompressible Navier-Stokes equations, two-phase porous media flow, and fully 3D nonlinear water waves, see [20, 21, 22, 23, 24].

Acknowledgement

The work has been supported by the Norwegian Research Council (NFR) through *Programme for Supercomputing* in form of a grant of computing time.

References

[1] Diffpack World Wide Web home page,
See URL *http://www.nobjects.com/Diffpack*.

[2] H. P. Langtangen, *Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*, Springer (1999), an enlarged 2nd edition is to be published in 2002.

[3] Object-Oriented Numerics Page,
See URL *http://www.oonumerics.org/*.

[4] C. D. Norton, *Object-Oriented Programming Paradigms in Scientific Computing*, PhD thesis, Rensselaer Polytechnic Institute (1996).

[5] E. Arge, A. M. Bruaset, H. P. Langtangen, *Object-oriented numerics*, in M. Dæhlen, A. Tveito, eds., *Mathematical Models and Software Tools in Industrial Mathematics*, Birkhäuser (1997).

[6] E. Arge, A. M. Bruaset, H. P. Langtangen, eds., *Modern Software Tools for Scientific Computing*, Birkhäuser (1997).

[7] H. P. Langtangen, A. M. Bruaset, E. Quak, eds., *Advances in Software Tools for Scientific Computing*, Springer (1999).

[8] M. Thuné, E. Mossberg, P. Olsson, J. Rantakokko, K. Åhlander, K. Otto, *Object-oriented construction of parallel PDE solvers*, in E. Arge, A. M. Bruaset, H. P. Langtangen, eds., *Modern Software Tools for Scientific Computing*, Birkhäuser (1997), pp. 203–226.

[9] D. L. Brown, W. D. Henshaw, D. J. Quinlan, *Overture: An object-oriented framework for solving partial differential equations*, in Y. Ishikawa, R. R. Oldehoeft, J. V. W. Reynders, M. Tholburn, eds., *Scientific Computing in Object-Oriented Parallel Environment, Springer-Verlag Lecture Notes in Computer Science 1343*, Springer-Verlag (1997), pp. 177–184.

[10] R. Wyrzykowski, T. Olas, N. Sczygiol, *Object-oriented approach to finite element modeling on clusters*, in T. Sørevik, F. Manne, R. Moe, A. H. Gebremedhin, eds., *Applied Parallel Computing; New Paradigm for HPC in Industry and Academia, Springer-Verlag Lecture Notes in Computer Science 1947*, Springer-Verlag (2001), pp. 250–257.

[11] PETSc World Wide Web home page,
See URL *http://www.mcs.anl.gov/petsc/petsc.html*.

[12] Message Passing Interface Forum, *MPI: A message-passing interface standard*, Internat. J. Supercomputer Appl., 8, (1994), 159–416.

[13] T. F. Chan, T. P. Mathew, *Domain decomposition algorithms*, in *Acta Numerica 1994*, Cambridge University Press (1994), pp. 61–143.

[14] B. F. Smith, P. E. Bjørstad, W. Gropp, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge University Press (1996).

[15] J. Xu, *Iterative methods by space decomposition and subspace correction*, SIAM Review, 34(4), (1992), 581–613.

[16] A. M. Bruaset, *A Survey of Preconditioned Iterative Methods*, Addison-Wesley Pitman (1995).

[17] A. M. Bruaset, X. Cai, H. P. Langtangen, A. Tveito, *Numerical solution of PDEs on parallel computers utilizing sequential simulators*, in Y. Ishikawa, R. R. Oldehoeft, J. V. W. Reynders, M. Tholburn, eds., *Scientific Computing in Object-Oriented Parallel Environment, Springer-Verlag Lecture Notes in Computer Science 1343*, Springer-Verlag (1997), pp. 161–168.

[18] X. Cai, E. Acklam, H. P. Langtangen, A. Tveito, *Parallel computing in Diffpack*, in H. P. Langtangen, A. Tveito, eds., *Advanced Topics in Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*, Springer (2002).

[19] X. Cai, *Overlapping domain decomposition methods*, in H. P. Langtangen, A. Tveito, eds., *Advanced Topics in Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*, Springer (2002).

[20] X. Cai, *Numerical simulation of 3D fully nonlinear water waves on parallel computers*, in B. Kågström, J. Dongarra, E. Elmroth, J. Waśniewski, eds., *Applied Parallel Computing; Large Scale Scientific and Industrial Problems, Springer-Verlag Lecture Notes in Computer Science 1541*, Springer-Verlag (1998), pp. 48–55.

[21] X. Cai, H. P. Langtangen, O. Munthe, *An object-oriented software framework for building parallel navier-stokes solvers.*, in D. Keyes, A. Ecer, N. Satofuka, P. Fox, J. Periaux, eds., *Parallel Computational Fliud Dynamics; Towards Tereflops, Optimization and Novel Formulations*, Elsevier (2000), pp. 147–154.

[22] H. P. Langtangen, X. Cai, *A software framework for easy parallelization of pde solvers*, in *Proceedings of the Parallel CFD 2000 Conference* (2000).

[23] X. Cai, Å. Ødegård, *On the performance of PC clusters in solving partial differential equations*, in *Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, SIAM (2001).

[24] X. Cai, H. P. Langtangen, *How modern programming techniques can greatly simplify the development of parallel simulation codes in computational mechanics*, in B. Skallerud, H. I. Andersson, eds., *MekIT'01 First National Conference on Computational Mechanics*, Norwegian University of Science and Technology (2001), pp. 63–76.