42

Domain Decomposition in High-Level Parallelization of PDE Codes

Xing Cai [1]

INTRODUCTION

We introduce a high-level approach to parallelizing sequential software for solving partial differential equations (PDEs). In short, we use domain decomposition (DD) at a higher level than that of linear algebra. Combined with extensive use of object-oriented (O-O) programming techniques, this approach promotes an efficient, flexible and systematic parallelization process. We present the software engineering aspects of this approach by explaining a generic implementation framework in the O-O scientific computing environment—Diffpack. Finally, a concrete case study will demonstrate the efficiency and flexibility of the parallelization process in this scenario.

Domain decomposition and parallel PDE codes

DD methods (see e.g. [CM94, SBG96]) are not only numerically efficient, but also inherently parallel. The original solution domain is partitioned into many subdomains that can be assigned to different processors of a parallel computer. The global solution is then found by an iterative process in which subdomain solves are invoked under a global administration. Therefore, using DD as the mathematical foundation for writing parallel PDE codes is becoming quite standard nowadays. However, the common

1 Department of Informatics, University of Oslo, P.O. Box 1080, N-0316 Oslo, Norway.
email: xingca@ifi.uio.no
Eleventh International Conference on Domain Decomposition Methods
Editors Choi-Hong Lai, Petter E. Bjørstad, Mark Cross and Olof B. Widlund

practice has been to address the DD methods directly at the level of linear algebra, i.e., in terms of parallel matrix-vector operations. In addition, the implementation of parallel PDE codes is often done using procedural programming languages. These two factors together bring the consequence that parallel PDE codes need often to be written entirely from scratch, thus making little use of existing sequential PDE codes.

Object-oriented programming and Diffpack

In recent years, the application of object-oriented (O-O) programming techniques in developing PDE software has shown its many advantages. The C++ programming language, which is a foremost representative in this respect, has inherent features that are well suited for the complicated process of numerical PDE solution. (We refer to [BN94, Lan99] for an introduction to this topic.) The most attractive features of C++ are modularity, polymorphism and inheritance. So extensive code reuse is a direct advantage of O-O programming. The possibility of a modular implementation of mathematical abstractions gives rise to an application independent PDE kernel, which is reusable in many PDE applications from different disciplines of scientific computing. Moreover, other functionalities such as I/O, visualization and automatic result reporting can also be extracted into libraries, because O-O programming promotes a unified interface of PDE simulators. The application programmer needs only to concentrate on critical numerics, at a high abstraction level if desired. Therefore reliable, flexible and extensible simulators can be developed in an efficient way.

Diffpack is an O-O scientific computing environment (see [Lan99, Dif]). The design of Diffpack has taken numerical efficiency into consideration by confining O-O techniques to high-level administrative tasks, while using low-level C codes and carefully constructed for-loops in CPU intensive numerics. The C++ Diffpack libraries contain, among other things, user-friendly objects for I/O, GUIs, arrays, linear systems and solvers, grids, scalar and vector fields, visualization and grid generation interfaces. This makes it very easy to build prototypical PDE simulators based on reliable and optimized Diffpack components.

The simulator-parallel model

Parallel computing has the potential for not only reducing the computation time, but also concentrating memories belonging to different processors to carry out larger calculations. So the migration of sequential PDE simulators to multiprocessor platforms is well motivated. Basically, we want the time used to solve a PDE on a parallel computer with P homogeneous processors to be about T/P, where T is the solution time needed by a sequential simulator. This requires a good numerical scheme that promotes an optimal parallelization of a sequential PDE simulator.

In this paper, we will focus on the overlapping DD methods of the additive Schwarz type, partly due to their simple algorithmic structure and partly due to the readiness for parallelism. An important observation about the additive Schwarz methods is that the main ingredient is the subdomain solver that carries out local and sequential operations. Furthermore, these sequential operations are of the same type as those needed to solve the original PDE on the whole domain. So if a mechanism can be found to apply an existing sequential simulator in subdomain solves on each subdomain, the

parallelization work will reduce to global administration and inter-subdomain data exchange. The O-O programming techniques serve perfectly as such a mechanism. First, they can produce sequential PDE simulators with unified generic interfaces, which easily allow modification and extension in order to be incorporated into a parallel setting. Second, the data encapsulation feature of O-O programming makes it possible to hide the computational details, e.g., the matrix-vector operations. As a result, O-O programming techniques allow code reuse at the level of subdomain simulators, which is higher than the level of linear algebra.

We hence introduce the so-called *simulator-parallel* programming model that was first proposed in [BCLT97]. In this model, each processor of a parallel computer hosts one or several subdomains. Each subdomain is then handled by a subdomain solver, which is easily extended from an existing sequential simulator designed for the whole domain. It is the basic building block of overlapping Schwarz methods—the subdomain solver—which relates our simulator-parallel model to DD methods, because an O-O sequential PDE simulator for the whole domain has all the functionalities needed in the subdomain solves. Of course, running DD methods in parallel needs a global administration and additional communication functionalities. We will show in the next section that O-O programming techniques are the right tool to produce a generic implementation framework where users can plug-and-play ready-built components for global administration and communication, so that parallel PDE simulators can be developed on the basis of existing sequential simulators in an efficient and systematic way.

A GENERIC IMPLEMENTATION FRAMEWORK

Our objective is a generic implementation framework that is flexible, extensible and portable, so that the parallelization work following the simulator-parallel model is relatively straightforward. On the whole, the efficiency and flexibility of the existing sequential simulator will be maintained, while the intrinsic numerical efficiency of DD can enhance the overall efficiency of the resulting parallel simulator. We also allow the user to make run-time decision of whether using additive DD methods as stand-alone iterative methods or preconditioners for Krylov subspace methods.

We consider such a generic framework consisting of three main parts: the sequential subdomain simulators, a communication part and a global administrator. Two class hierarchies, whose base classes are `SubdomainSimulator` and `Communicator`, are built to represent the sequential subdomain simulators and to handle the needed communications, respectively. The different classes in the class hierarchies are designed to be used in different situations. In addition, the design of the global administrator offers great flexibility. It allows the user to choose, among other things, the specific numerical method at run-time. The part inside the global administrator that makes connections with the subdomain simulators and the communication part can also be easily modified by the user.

Subdomain simulators

The base class `SubdomainSimulator` gives a generic representation of any sequential subdomain simulator in our framework. Functionalities of `SubdomainSimulator` include a numerical discretization scheme and an assembly process for building up the local linear system. A linear algebra toolbox also exists in `SubdomainSimulator` to control the choice of the local solution method, preconditioner, stopping criterion etc. We have made most of the member functions of `SubdomainSimulator` to be *pure virtual*, so they need to be overridden in a derived subclass. These member functions constitute a standard interface shared by all the subdomain simulators. It is through this generic interface that the communication part and the global administrator of the implementation framework operate. Adapting an existing sequential simulator also becomes easy, because most of the work consists merely of binding the pure virtual member functions in `SubdomainSimulator` to the concrete member functions in the existing simulator. Building up the class hierarchy, we have derived `SubdomainFEMSolver` for simulators solving a scalar/vector elliptic PDE discretized by finite element methods, and `SubdomainFDMSolver` for simulators using finite difference discretizations. Furthermore, a subclass named `SubdomainMGSolver` is derived from `SubdomainFEMSolver` to represent simulators using multigrid V-cycles in subdomain solves.

Communication

During parallel DD iterations, the concrete communication between processors is in form of exchanging messages and is handled by objects of `Communicator`. On each processor, there is one such object connecting to the local subdomain simulator(s). The reason for separating the communication part from the global administrator is to hide the low-level message passing codes and instead offer convenient high-level communication commands. It is thus possible to change MPI, which is used in the current implementation, to another message passing standard without affecting the other parts of the framework. Different subclasses of `Communicator` are derived to handle different situations. For example, class `CommunicatorFEMSP` specializes in communication between subdomain simulators using the finite element discretization, whereas class `CommunicatorFDMSP` works for subdomain simulators using the finite different method.

The global administrator

The inheritance feature of C++ is reflected in the above two class hierarchies `SubdomainSimulator` and `Communicator`. The user can either take ready-built class objects directly from these two hierarchies and use them in a specific application, or derive new subclasses to incorporate new adaptations. However, the O-O design of the global administrator offers further flexibility and lets the user plug-and-play at run-time. For example, the user can decide whether to use DD as a stand-alone iterative method or a preconditioner for a specific Krylov subspace method. The user is also free to choose between running one-level and two-level DD by removal or addition of a coarse global grid. We have thus devised a main administrator class `PdeFemAdmSP`

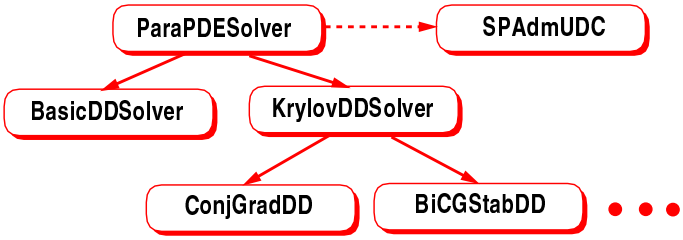

Figure 1 The class hierarchy of `ParaPDESolver` and its connection to `SPAdmUDC`. The solid arrows indicate inheritance ("is-a" relationship), whereas the dashed arrow indicates "has-a" relationship.

whose basic structure is as follows:

```
class PdeFemAdmSP
{
protected:
  ParaPDESolver_prm psolver_prm;
  Handle(ParaPDESolver) psolver;
  SPAdmUDC* udc;
  ...
};
```

In above, `ParaPDESolver_prm` is an object containing many parameters to be chosen by the user at run-time. Among the parameters there are: 1. flag indicating whether the overlapping Schwarz method should be used as a stand-alone iterative method, 2. name of the desired Krylov subspace method (when an overlapping Schwarz method is used as a preconditioner), and 3. number of maximum iterations, prescribed accuracy and type of the convergence monitor etc.

The second component of class `PdeFemAdmSP` is `ParaPDESolver` whose class hierarchy is depicted in Figure 1. In this hierarchy, class `BasicDDSolver` represents an overlapping Schwarz method to be used as a stand-alone iterative solver, and subclasses of `KrylovDDSolver` use one overlapping Schwarz iteration as the preconditioner. At run-time, when the user has chosen the parameters by e.g. filling items on a user-friendly menu, a concrete object of `ParaPDESolver` will be created. (We remark that `Handle(X)` is a safer pointer for class `X` in Diffpack.) The involved subdomain solves will then be undertaken by a sequential subdomain simulator, which `ParaPDESolver` gets connection through the third and last component of `PdeFemAdmSP`, namely `SPAdmUDC`. It is through `SPAdmUDC` that `ParaPDESolver` invokes member functions of the subdomain simulator.

In addition to making connection between `ParaPDESolver` and the subdomain simulators, `SPAdmUDC` is also responsible for getting access to the communication part for the global administrator. We remark that "UDC" stands for "user-defined-codes" and is used here to indicate that the user has the possibility of making modifications of the member functions. In a parallel simulation, one `SPAdmUDC` object resides on each processor and has one `Communicator` object plus one or several `SubdomainSimulator` objects under its control.

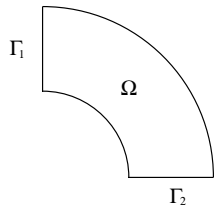

Figure 2 The solution domain for the linear elasticity problem.

A CASE STUDY

We demonstrate the efficiency and flexibility of the generic implementation framework by parallelizing an existing sequential Diffpack simulator for solving a linear elasticity problem in 2D. The mathematical model is the following vector PDE:

$$-\mu\Delta\mathbf{U} - (\mu+\lambda)\nabla\nabla\cdot\mathbf{U} = \mathbf{f},$$

where $\mathbf{U} = (\mathbf{u_1}, \mathbf{u_2})$ is the primary unknown and $\mathbf{f}$ is a given vector function, μ and λ are constants. The 2D domain is a quarter of a hollow disk (see Figure 2). On the boundary the stress vector is prescribed, except on Γ_1, where $u_1 = 0$, and on Γ_2, where $u_2 = 0$.

As stated above, a sequential Diffpack simulator exists for solving the 2D linear elasticity problem. The simulator is represented by class `Elasticity2D`, which has a grid, a finite element field for the unknowns, a member function modelling the integrands in the weak formulation, a linear system toolbox, and some standard functions for prescribing the boundary conditions. To extend `Elasticity2D` with the generic interface required by the implementation framework, and allow multigrid V-cycles as subdomain solvers, we derive a new class `SubdomainELSolver` as the subclass of *both* `Elasticity2D` *and* `SubdomainMGSolver`. A simplified definition of `SubdomainELSolver` is as follows:

```
class SubdomainELSolver : public SubdomainMGSolver,
                          public Elasticity2D
{
  ...
  virtual void markEssBCs ();
  virtual void createHierMatrices ();
};
```

The above code segment shows that the main work of class `SubdomainELSolver` is to give explicit definitions of the pure virtual member functions of class `SubdomainMGSolver`. This is essentially done by binding the virtual functions of `SubdomainMGSolver` to concrete member functions of `SubdomainELSolver`. For example, the member function `markEssBCs` uses a corresponding function of `Elasticity2D` to mark essential boundary conditions on all levels of subgrids, while the member function `createHierMatrices` creates stiffness matrices on all the grid levels and has the following implementation:

```
void SubdomainELSolver:: createHierMatrices () {
  for (int i=no_of_grids; i>=1; i--)
    Elasticity2D::makeSystem(dofs(i)(),Amats(i)(),rhs_vecs(i)());
}
```

The second job a user needs to do is to make a connection between the extended sequential simulator `SubdomainELSolver` and the global administrator. This is achieved by deriving a new class `ELSolverSP` as a subclass of `SPAdmUDC`. Inside `ELSolverSP`, the user basically creates an object of `SubdomainELSolver` for each subdomain and puts them under the control of `SPAdmUDC`. Finally, the Diffpack main program will look as follows:

```
int main (int nargs, const char** args)
{
  initDiffpack (nargs, args);   // MPI_Init etc called inside
  MenuStream* menu = new MenuStream;
  menu->setInputFile ("el_parameters.txt");
  menu->init ("Linear Elasticity Test","DD Approach");

  ELSolverSP udc;
  PdeFemAdmSP adm;
  udc.define (*menu, MAIN); adm.define (*menu, MAIN);
  menu->prompt();
  udc.scan (*menu); adm.scan (*menu);
  adm.attachSPAdmUDC (&udc);
  adm.init ();
  adm.solve ();
}
```

For the numerical experiments we use a structured 241×241 curvilinear global grid. We use one processor per subdomain and denote the number of processors by P. For $P > 1$ the additive Schwarz method is used as the preconditioner for a parallel BiCGStab method for solving the linear system. The stopping criterion requires that the discrete L_2-norm of the global residual is reduced by a factor of 10^6. We introduce a global coarse grid to allow two-level DD iterations and obtain the overlapping subgrids by extending a non-overlapping partition with a factor of $\frac{1}{15}$ in each direction. One local multigrid V-cycle with one pre- and post-SSOR-smoothing is used in subdomain solves. For $P = 1$ a sequential BiCGStab method preconditioned by one global multigrid V-cycle is used. The CPU measurements obtained on an SGI Cray Origin 2000 machine with R10000 processors are listed in Table 1, where I denotes the number of BiCGStab iterations.

CONCLUDING REMARKS

We have shown that addressing DD at the level of subdomain simulators, combined with extensive use of O-O programming techniques, results in an efficient, flexible and systematic process for producing parallel PDE codes. A generic implementation framework is presented in this paper, together with a concrete case study where we

Table 1 Solution of the linear system for the linear elasticity problem.

P	CPU	I	subdomain grid
1	66.01	19	241×241
2	24.64	12	129×241
4	14.97	14	129×129
8	5.96	11	69×129
16	3.58	13	69×69

parallelize an existing sequential Diffpack simulator for a linear elasticity problem. Finally, we mention that the generic implementation framework can be easily extended to parallelize sequential simulators for nonlinear elliptic PDEs and parabolic problems, where the computational kernel of the DD methods is still subdomain solves (see e.g. [Tai98, TE98]).

Acknowledgments. This work has received support from The Research Council of Norway (Programme for Supercomputing) through a grant of computing time. The author also acknowledges the help from Prof. Aslak Tveito, Dr. Hans Petter Langtangen and Dr. Are Magnus Bruaset.

REFERENCES

[BCLT97] Bruaset A. M., Cai X., Langtangen H. P., and Tveito A. (1997) Numerical solution of PDEs on parallel computers utilizing sequential simulators. In et al. Y. I. (ed) *Scientific Computing in Object-Oriented Parallel Environment, Springer-Verlag Lecture Notes in Computer Science 1343*, pages 161–168. Springer-Verlag.

[BN94] Barton J. J. and Nackman L. R. (1994) *Scientific and Engineering C++. An Introduction with Advanced Techniques and Examples.* Addison-Wesley.

[CM94] Chan T. F. and Mathew T. P. (1994) Domain decomposition algorithms. In *Acta Numerica 1994*, pages 61–143. Cambridge University Press.

[Dif] Diffpack Home Page. http://www.nobjects.com/Products/Diffpack.

[Lan99] Langtangen H. P. (1999) *Computational Partial Differential Equations – Numerical Methods and Diffpack Programming.* Springer-Verlag.

[SBG96] Smith B. F., Bjørstad P. E., and Gropp W. (1996) *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press.

[Tai98] Tai X.-C. (1998) A space decomposition method for parabolic problems. *Numer. Method for Part. Diff. Equat.* 14(1): 27–46.

[TE98] Tai X.-C. and Espedal M. (1998) Rate of convergence of some space decomposition methods for linear and nonlinear problems. *SIAM J. Numer. Anal.* 35(4): 1558–1570.