How Modern Programming Techniques Can Greatly Simplify the Development of Parallel Simulation Codes in Computational Mechanics

Xing Cai  Hans Petter Langtangen

Department of Informatics
University of Oslo
{xingca,hpl}@ifi.uio.no

Abstract

We explain how domain decomposition methods combined with object-oriented programming techniques can simplify the process of developing parallel simulation codes in computational mechanics. More specifically, we present a generic software framework for implementing overlapping Schwarz methods that possess high numerical efficiency and inherent parallelism. The actual parallelization work in this framework reduces to deriving a couple of small C++ subclasses. We also examine a concrete example of parallelizing a nonlinear water wave simulator in the program system Diffpack, with a detailed report on the speed-up results obtained on a Beowulf PC cluster.

Introduction

Parallel computing is a key technology for large-scale simulations in the field of computational mechanics. However, developing huge parallel finite element codes is known to be a complicated and error-prone process when applying common low-level programming tools such as MPI [12] or PVM. The ultimate goal would be to first develop sequential code, without paying attention to parallel computing aspects, and then as a post-process add the necessary code
to utilize parallel computers. A first step in this direction is presented in the present paper. The methodology is based on parallelization of the mathematical model via domain decomposition methods and implementation using object-oriented programming (OOP) techniques. By representing the sequential code as an object and using OOP technology, typically supported by languages such as C++, Java or Python, one can develop a general parallelization toolbox, which can be combined with the original sequential solver to form a parallel version of the solver.

The modern OOP techniques [1] have many advantages in developing software for scientific computing. The powerful features of OOP, such as modularity, polymorphism, and inheritance, allow the user to create and manipulate complex data structures at a high abstraction level, close to the physics, mathematics, and numerics of the problem. Experience also shows that the resulting code is easier to extend and maintain than typical Fortran 77 or C counterparts.

Scientific software based on programming with objects typically consists of classes in layers of increasing abstractions. Each class can correspond to a data structure and/or operation in the numerical algorithms. Consider, for instance, software for partial differential equations (PDEs) using the finite element method. A well-written C++ finite element library translates components of the numerics into C++ classes such as finite element grid, scalar and vector fields, different element types, numerical integration rules, stiffness matrix, right-hand side vector, linear and nonlinear solution methods etc. Diffpack [8, 10] is such a library. Solving a specific PDE in Diffpack consists of deriving a simulator class from a general-purpose finite element class in the libraries. The simulator class typically contains the problem-specific physical parameters, basic high-level data structures such as finite element grids and fields, in addition to some virtual functions that provide the library with problem-specific information, like the integrand in the variational formulation of the PDE problem.

Our parallel software development strategy is based on parallelizing the mathematical model through a domain decomposition approach. The sequential solver is applied in each subdomain, while a generic domain decomposition algorithm takes care of communication aspects and the outer iteration over the subdomains. This algorithm is, in principle, applicable to all kinds of PDE problems, although the convergence properties in a specific problem may constitute an open question. To build a general parallelization tool based on this algorithm, one needs to view the subdomain PDE solver (the original code) in a generic fashion. That is, all solvers must look the same, meaning that they must have the same programming interface when viewed from the domain decomposition algorithm. This is exactly what OOP provides; related software
components (classes) can be given a common, unified interface.

The methodology proposed in this paper has been implemented in a general framework on top of Diffpack and applied to a wide range of models from mechanics with promising speed-up results, see [4, 5, 6, 11]. These models include heat transfer, viscous flow, two-phase porous media flow, elasticity, and wave propagation. The application in focus of the present paper is three-dimensional, fully nonlinear water waves described by classical potential theory.

Parallel Domain Decomposition Methods

Domain decomposition (DD) methods are well known to have superior numerical efficiency for solving PDEs. Roughly speaking, these methods search for the solution of the original global problem by iteratively solving smaller subproblems. The idea for parallelization is that the subproblems can be solved concurrently. The so-called overlapping Schwarz methods are a sub-category of the DD methods and operate on overlapping subdomains. They have a simple algorithmic structure because there is no need to solve special interface problems as are required in the common non-overlapping DD methods (also known as Schur complement methods).

A very brief mathematical description of the overlapping Schwarz methods is given next. More detailed information can be found in the references [7, 13, 15]. Suppose we want to solve a PDE on the operator form

$$Lu = f$$

in a global solution domain $\Omega$. The overlapping Schwarz methods start with a partition of $\Omega$ into a set of overlapping subdomains $\Omega_1, \Omega_2, \ldots, \Omega_P$. The global solution $u$ is sought by iterating over subdomain solves

$$L_i u_i^n = f_i^n,$$  \hspace{1cm} (2)

where the superscript $n$ denotes the iteration number and $L_i$ is the restriction of $L$ onto $\Omega_i$. The right-hand side $f_i^n$ arises from restricting $f$ onto $\Omega_i$. In our applications, (2) is the same PDE as in (1), but the domain is $\Omega_i$ instead of $\Omega$, and the boundary data include (in general) physical conditions on physical boundaries and the solution in neighboring subdomains on artificial, internal boundaries.

The so-called *additive Schwarz method* uses $u^{n-1}$ in the boundary data for all the subdomains. This type of overlapping Schwarz method is particular
interesting in the context of parallel computing, because all the subproblems can be solved in parallel, independently of each other.

For more robust convergence, it is quite common to use one additive Schwarz iteration as preconditioner for some Krylov subspace method applied to the linear system arising from (2). That is, DD is used as preconditioner instead of as a stand-alone iterative method. Given a right-hand side \( w \), the result \( v \) of applying an additive Schwarz preconditioner can be constructed as:

\[
v = \sum_{i=1}^{P} R_i^T L_i^{-1} R_i w,
\]

where \( R_i \) denotes a restriction operator from \( \Omega \) to \( \Omega_i \) and \( R_i^T \) is the associated interpolation operator. We also remark that in (3) an inexact subdomain solver \( L_i^{-1} \) may be used to save computational effort. In addition, to achieve better convergence, one may use an extra so-called coarse grid correction, giving rise to two-level overlapping Schwarz methods. One can view the resulting approach as a two-level multigrid algorithm using domain decomposition for the fine-grid solve. In the present paper, we consider applying the coarse grid correction in a multiplicative fashion, i.e.,

\[
\tilde{v} = R_0^T L_0^{-1} R_0 w, \quad v = \sum_{i=1}^{P} R_i^T L_i^{-1} R_i (w - L \tilde{v}).
\]

In above, \( R_0 \) and \( R_0^T \) are the restriction and interpolation operators associated with a coarse grid over \( \Omega \). The reason for not using the coarse grid correction in a fully additive fashion is the unnecessary difficulty in connection with parallelizing a (direct) coarse grid solver. Besides, applying the coarse grid correction in a multiplicative fashion will normally gives slightly better convergence. In regard of parallel implementation, it implies some overhead as the coarse grid problem is to be solved by a sequential method, but as long as the size of the coarse grid is considerably smaller than that of a subgrid, this overhead does not signify.

**A Generic Implementation Framework**

An important observation about overlapping Schwarz methods is that the subproblems arise from restricting the original PDE(s) onto the subdomains. That
is, every subproblem is a miniature of the original global problem. The basic building block of overlapping Schwarz methods is the subdomain solver. Parallelism arises naturally in the above additive Schwarz method because the subdomain solves can be carried out completely independently. Equally important is the fact that if there exists a sequential simulator for solving the global problem, in principle it should also be applicable to the subproblems.

The traditional way of parallelizing DD methods is done at the level of linear algebra, i.e., direct parallelization of involved linear algebra operations. However, we advocate a so-called simulator-parallel approach [2] that aims to apply extended/modified sequential simulators as individual units in overlapping Schwarz methods. More precisely, we assign each subdomain with a sequential simulator and leave the computation coordination of the processors to a global administrator, which is implemented at a high abstraction level close to the mathematical formulation of overlapping Schwarz methods.

In this parallelization approach we only work with local PDE problems, the data distribution is implied by an overlapping partition, so there is no explicit need for global representation of data. Moreover, treating subproblems at its own level allows using physical properties of the involved PDEs. Furthermore, the local administration of each subdomain simulator allows flexible choice of its own solution method, preconditioner, stopping criterion etc. Most importantly, parallelization at the level of subdomain simulators opens the possibility of reusing existing reliable and optimized sequential simulators. The parallelization approach can essentially take any sequential simulator, which handles arbitrary grid and boundary conditions and is capable of assembling and solving the subdomain linear system \( A_i x_i = b_i \). Numerically, the combination of the efficiency of the sequential simulator and the overlapping Schwarz method ensures the overall numerical efficiency of the resulting parallel simulator.

The simulator-parallel approach strongly promotes code reuse, because an original sequential simulator can be easily applied in subdomain solves. Besides, most of the global administration and the related communication between processors can be extracted into a parallelization toolbox, independent of specific applications. Of course, minor modifications of an overlapping Schwarz method for treating specific situations are frequently necessary. On the other hand, subdomain solvers, especially those extended from existing sequential simulators, need to have flexible enough implementations to fit into a DD framework.

OOP techniques are advantageous for the implementation of the simulator-parallel approach not only because they are convenient for minor extension or modification of an existing sequential simulator, but more importantly because they can be used to build up a generic implementation framework that encap-
ulates the necessary inter-processor communication and global administration. The framework promotes a systematic parallelization approach and thus greatly simplifies the coding effort of a user, so that it can produce parallel simulators in a flexible and efficient way.

Design overview

Our generic framework consists of three main parts: The sequential subdomain simulators, a communication part and a global administrator, see Figure 1. A class hierarchy with base class SubdomainSimulator is built to give a generic representation of any sequential subdomain simulator. Different classes in the SubdomainSimulator hierarchy are designed to be used in different situations. The communication part incorporates the complex message passing functionalities and hides the nasty low-level details of MPI programming. OOP techniques also inject great flexibility into the design of the global administrator. It allows the user to choose, among other things, whether to use an additive Schwarz method as a preconditioner or a stand-alone iterative solver. The part inside the global administrator that makes connection with the subdomain simulators and the communication part can also easily be modified by the user.

![Diagram](image)

Figure 1: Three software modules on each processor: the administration of the DD algorithm, one or more subdomain solvers, and a communication tool for exchanging internal boundary values between subdomains.
Subdomain simulators

Class SubdomainSimulator gives a generic representation of any sequential subdomain simulator in our framework. The local data structure of such a simulator contains, among other things, the subgrid, the local stiffness matrix, the local right-hand side vector and the local solution vector. The basic functionalities include a numerical discretization scheme and an assembly process for building up the local linear system. A linear algebra toolbox is also necessary for SubdomainSimulator to control the choice of the local solution method, preconditioner, stopping criterion etc. In addition, there are functionalities for accessing the local data. We have made most of the member functions in SubdomainSimulator pure virtual: they need to be overridden in a derived subclass. These member functions thus constitute a standard interface shared by all the subdomain simulators. It is through this standard interface that the communication part and the global administrator of the implementation framework operate. Adapting an existing sequential simulator is therefore 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.

As subclasses of SubdomainSimulator, we have derived SubdomainFEMSolver for simulators solving a scalar/vector elliptic PDE discretized by FE methods, and SubdomainFDMSolver for simulators using finite difference discretizations. A subclass with name SubdomainFEMMGSolver derived from SubdomainFEMSolver is also derived for simulators using multigrid V-cycles in subdomain solves. In short, the user is provided with a selection of ready-made classes. The user can either use these ready-made classes directly in a specific application or derive from them new subclasses to incorporate new adaptations.

The global administrator

Overlapping Schwarz methods can either work as stand-alone iterative solution methods or as preconditioners for Krylov subspace methods. The first objective of the global administrator is therefore to offer the above choice at run-time. If the user decides to use an overlapping Schwarz method as the preconditioner for a Krylov subspace method, the administrator should also let him/her pick up a particular Krylov method in the standard Diffpack way. We have thus devised a main administrator class PdeFemAdmSP whose basic structure is as follows:

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

In the above code segment, ParaPDESolver_prm is an object containing many parameters to be chosen by the user at run-time. Among the parameters there are

1. flags indicating whether the overlapping Schwarz method should be used as a preconditioner,
2. number of maximum DD iterations, prescribed accuracy and type of the convergence monitor etc.

The second component of class PdeFemAdmSP is ParaPDESolver that has two subclasses. The first subclass BasicDDSolver represents an overlapping Schwarz method to be used as a stand-alone iterative solver, and the second subclass KrylovDDSolver works as a preconditioner. At run-time, when the user has chosen the parameters by e.g. filling items on a user-friendly menu, a concrete parallel PDE solver is created. In other words, Handle(ParaPDESolver) will be bound to a concrete object of ParaPDESolver:

    psolver.rebind(p solver_prm.create());

The parallel solution of a PDE will be carried out by the virtual member function ParaPDESolver::solve. For BasicDDSolver the solve function has the following implementation:

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

The last component of PdeFemAdmSP is SPAdmUDC, which connects the global administrator with the subdomain simulators and the communication part. In a
parallel simulation, one SPAdmUDC object resides on each processor and has one CommunicatorFEMSP object plus one or several SubdomainFEMSolver objects under its control. 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 its member functions. One of the major member functions of SPAdmUDC is oneDDIteration whose simplified implementation reads:

```cpp
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 ();
}
```

A Case Study

The purpose of this section is to demonstrate how OOP and the above mentioned implementation framework can ease the development of a particular parallel water wave simulator. We will also show the portability and efficiency of the parallel simulator by concrete CPU-measurements obtained on a Beowulf-class [14] Linux cluster.

The mathematical model

We consider the following system of PDEs modeling fully nonlinear 3D water waves with an inviscid fluid model:

\[
\begin{align*}
-\nabla^2 \varphi &= 0 \quad \text{in the water volume}, \\
\eta_t + \varphi_x \eta_x + \varphi_y \eta_y - \varphi_z &= 0 \quad \text{on the free surface}, \\
\varphi_t + \frac{1}{2} (\varphi_x^2 + \varphi_y^2 + \varphi_z^2) + g \eta &= 0 \quad \text{on the free surface}, \\
\frac{\partial \varphi}{\partial n} &= 0 \quad \text{on solid boundaries}.
\end{align*}
\]

Here, the primary unknowns are the velocity potential \( \varphi(x, y, z, t) \) and the the free surface elevation \( \eta(x, y, t) \).
An efficient numerical method

At each time step we first update \( \eta \) by an explicit finite difference scheme and then solve \( \varphi \) implicitly. More specifically, the Laplace equation in the water volume is discretized by finite elements. The dynamic water volume, which is the physical domain for the Laplace equation, is transformed onto a fixed domain. This will introduce an elliptic boundary value problem with a time-dependent variable coefficient, however regridding of the computational domain at each time step can be avoided. We refer to [3] for the details of the numerical method. The numerical efficiency of the whole numerical method relies on an efficient solution of this elliptic boundary value problem. We hereby apply in the present paper the additive Schwarz method as preconditioner to speed up a Krylov subspace method (in the case the Conjugate Gradient method) for solving the linear system at each time step.

A parallel DD 3D wave simulator

Our starting point for developing a parallel wave simulator is an existing sequential Diffpack wave simulator implementing the numerical method just outlined. This simulator is available as a C++ class with name NWaVEq3D. The work of parallelization is twofold. First, we derive a new C++ class (say) NWaVEq3Ds to function as the subdomain solver. In the C++ terminology this means:

```cpp
class NWaVEq3Ds : public NWaVEq3D, 
                   public SubdFEMMGSolver
{
    // ....
    virtual void createLocalMatrix () {
        NWaVEq3D::makeSystem ();
    }
};
```

The above code segment shows a simplified definition of NWaVEq3Ds that is the extended sequential simulator to be used in the framework. We see that class NWaVEq3Ds is derived from both the existing sequential simulator NWaVEq3D and SubdFEMMGSolver, which we recall from Section is a subclass of SubdFEMSolver (therefore also subclass of SubdomainSimulator) and has functionalities for running multigrid V-cycles. We also see an example of binding a virtual member function of SubdomainFEMSolver to a concrete member function of NWaVEq3Ds. We emphasize that NWaVEq3Ds is a small piece of
code, gluing the sequential solver and the parallelization toolbox such that it can be used in a completely generic implementation of the DD algorithm.

The second part of the parallelization work consists in the derivation of a new class WASolverSP from the base class SPAdmUDC. The main functionality of WASolverSP is to carry out grid partition and make run-time connection between the different components of the generic framework. As mentioned above, the computation at each time step consists of (i) explicit update of the surface elevation $\eta$ and $\varphi|_{z=\eta}$, and (ii) implicit solution of an elliptic boundary value problem with an variable coefficient. Parallelization of substep (i) is straightforward done by a 2D partition of the water surface. The partition of the 3D solution domain arises from extending in the $z$-direction the internal boundaries of the 2D partition. In short, the parallelization process is considerably easy, compared with the work of implementing the original wave simulator.

**CPU-measurements obtained on a Beowulf cluster**

The above parallel wave simulator is fully portable and can be run at any parallel platform where Diffpack and MPI is installed. The parallel computer that is used for the following experiments is a Beowulf-class cluster consisting of 24 dual Pentium-III computing nodes, where all the 48 processors run at 500MHz. Each computing node has 512MB of memory, amounting to a total of 12GB for the entire cluster. Every computing node is equipped with a 3com905B network interface card inter-connected with a 100Mbit/s ethernet network, through a 26-port Cisco Catalyst 2926 switch. The Debian GNU/Linux operating system runs on the cluster.

The solution domain is $(x,y,z) \in [0,80] \times [0,80] \times [-50,0]$ after a transformation at each time step. We use a global grid of size $49 \times 49 \times 41$. The simulation time period is $0 < t \leq 4$ seconds, which is divided into 32 time steps. The conjugate gradient (CG) method is used to solve the resulting linear system at each time step. For the additive Schwarz preconditioner, we use one multigrid V-cycle (see e.g. [9]) as an inexact subdomain solver. That is, the preconditioner for the single processor case is directly a multigrid V-cycle. We denote the number of processors by $P$ and the average number of CG iterations required at every time step by $I$. As the effect of the additive Schwarz preconditioner differs slightly for different values of $P$, we remark that $I$ differs also. For comparison of the parallel efficiency, we calculate the speed-up and efficiency results based on the CPU time per CG iteration. The latter quantity is denoted by $\hat{T}$ in Table 1.
### Table 1: CPU-measurements of a 3D simulation of nonlinear water waves; speed-up and parallel efficiency.

<table>
<thead>
<tr>
<th>$P$</th>
<th>Total CPU</th>
<th>$I$</th>
<th>$T$</th>
<th>Speed-up</th>
<th>Efficiency</th>
<th>Sub-grid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>561.56</td>
<td>6.16</td>
<td>2.849</td>
<td>N/A</td>
<td>N/A</td>
<td>49 × 49 × 41</td>
</tr>
<tr>
<td>2</td>
<td>596.99</td>
<td>14.12</td>
<td>1.321</td>
<td>2.16</td>
<td>1.08</td>
<td>49 × 26 × 41</td>
</tr>
<tr>
<td>4</td>
<td>386.83</td>
<td>15.69</td>
<td>0.770</td>
<td>3.70</td>
<td>0.92</td>
<td>26 × 26 × 41</td>
</tr>
<tr>
<td>8</td>
<td>272.22</td>
<td>21.31</td>
<td>0.399</td>
<td>7.14</td>
<td>0.89</td>
<td>26 × 14 × 41</td>
</tr>
<tr>
<td>16</td>
<td>150.01</td>
<td>22.59</td>
<td>0.208</td>
<td>13.73</td>
<td>0.86</td>
<td>14 × 14 × 41</td>
</tr>
<tr>
<td>24</td>
<td>124.52</td>
<td>26.50</td>
<td>0.147</td>
<td>19.40</td>
<td>0.81</td>
<td>14 × 10 × 41</td>
</tr>
<tr>
<td>48</td>
<td>124.84</td>
<td>30.13</td>
<td>0.129</td>
<td>22.00</td>
<td>0.46</td>
<td>8 × 10 × 41</td>
</tr>
</tbody>
</table>

**Remark.** For $P \leq 24$, i.e., when one processor is used per computing node, we have achieved quite satisfactory speed-up and parallel efficiency results on the low-cost Beowulf cluster. It is only when both processors on the same node are in use that performance really suffers. In our case, this is mainly due to the fact that the two processors compete against each other for the usage of the same network card and cable that connects them to the switch. This situation can be improved if two network cards are installed on every computing node.

### Conclusions

We have presented a parallelization strategy for simulation software in computational mechanics. The parallelization strategy is based on overlapping Schwarz methods applied to the original PDE problem. This splits the original PDE problem into a set of subproblems involving the same PDE, but with different boundary data. We have shown how OOP facilitates a generic implementation framework consisting of extensible and flexible components, where the original sequential simulator can be reused (mainly “as is”) in the new parallel solver.

The concrete work of parallelization thus reduces to easy derivation of short extensions of the components to fit into the generic framework. The framework has been applied to a range of problems in computational mechanics, but in the present paper we have focused at 3D fully nonlinear water waves. The parallel framework encourages us to apply state-of-the-art numerical methods that are “optimal” in a sequential setting and simple to parallelize. For example, the Laplace equation in the water wave model is solved by a Conjugate Gradient method with Domain Decomposition as preconditioner and a Multigrid V-cycle
as subdomain solve. When tested on a low-cost Beowulf cluster, the speed-up results are satisfactory, showing that the overhead of using object-oriented programming in the parallel framework is almost negligible.

References


