Parallel Distributed Object-Oriented Framework for Domain Decomposition

S.P. Kopyssov, I.V. Krasnopyorov, A.K. Novikov, and V.N. Rytchkov[1]

UB RAS, Institute of Applied Mechanics kopyssov@udman.ru

Summary. The aim of this work is to reduce the development costs of new domain decomposition methods and to develop the parallel distributed software adapted to high performance computers. A new approach to development of the domain decomposition software system is suggested; it is based on the object-oriented analysis and middleware CORBA, MPI. In this paper, the main steps of domain decomposition are determined, the object-oriented framework is described, and then it is extended for parallel distributed computing. The given examples demonstrate that the software developed in such a way provides mathematical clarity and rapid implementation of the parallel algorithms.

1 Introduction

The idea of domain decomposition (DD) used not to be applied to parallel algorithms and it gave rise to substructuring (Przemieniecki [1968]), subconstruction, macroelement, superelement, fragment, module-element, reduced element, Schwartz (Sobolev [1936]), capacity matrix and other methods. Usually these methods have been applied to reduce an initial problem in the domain with a complex boundary to the sequence of problems in the subdomains with sufficiently simple boundaries. Nowadays the parallel implementations of DD allow improving the computational performance.

The most of DD software is based on one or another approach to the approximation of a differential problem, mostly on the finite element (FE) method. The complexity of FE models results in the necessity of using suitable programming techniques such as the object-oriented (OO) analysis. At present, there are many publications on the FE OO models (Zimmermann et al. [1992]), but OO analysis is rarely applied to the DD. There are some references to the OO scientific software: Diffpack, PETSc, SPOOLES, Overture. Diffpack (Cai [1998]) is an OO environment aimed at solving partial differential equations (PDE). Overture (Brown et al. [1999]) provides the OO framework for solving PDE on overlapping grids. The fundamental abstractions are divided into functionality groups: data structures, linear and nonlinear solvers, PDE, utilities. Recently, DD and multigrid methods have been included. The MPI-based libraries PETSc (Portable Extensible Toolkit for Scientific computation) and SPOOLES (Sparse Object-Oriented Linear Equations Solver) use the OO style for matrix representation of PDE. In this work the fundamental OO framework consisting of the general FE and DD entities is suggested, and then it is extended by introducing new objects that implement specific algorithms including parallel ones.

For parallel distributed implementation of the DD framework, it is appropriate to apply the existing techniques and middleware. MPI and CORBA are the most commonly used ones. MPI (Message Passing Interface) is used in massively parallel systems. An MPI-based program describes one of the identical processes handling its own portion of the data (SIMD). MPI provides blocking/nonblocking communications between the groups of processes. C++ can be used to implement a parallel distributed object; for that, it is necessary to implement dynamic creation of an object and remote method invocation. The main shortcomings from the point of view of flexibility are the following: the procedure orientation and the primitiveness of the program starting system.

CORBA (Common Object Request Broker Architecture) is used to create OO distributed applications. CORBA provides synchronous/asynchronous remote method invocations and allows creating complex, high performance, cross platform applications. The performance measurements of TAO (CORBA) and MPICH (MPI) middleware were taken on Gigabit Ethernet; they showed the same throughput, with MPICH giving lower latency. The recently published results of comparison of OmniORB and MPICH on Myrinet and SCI communications (Denis et al. [2003]) prove ours. CORBA is an interesting alternative to MPI for flexible and high performance implementation of complex models.

2 Representation of the main steps of DD in the OO framework

The OO analysis was applied for creating an abstract software model, the C++ language was used for programming. Inheritance and polymorphism provided the flexibility of the framework. Data encapsulation brought about creating three subsystems: modeling classes, numerical classes, and analysis classes. The OO model of analysis is shown in Figure 1. Let us determine the main steps of DD and consider them from the point of view of OO programming.

1. Building the finite element model. The `DomainBuilder` class is the base class of design model editors. It provides the methods to create and edit the domain represented by the `Domain` class, which contains a geometry and a FE mesh consisting of nodes (`Node`), different types of elements (`Triangle`, `Tetrahedron`, `Hexahedron` and others), and boundary conditions

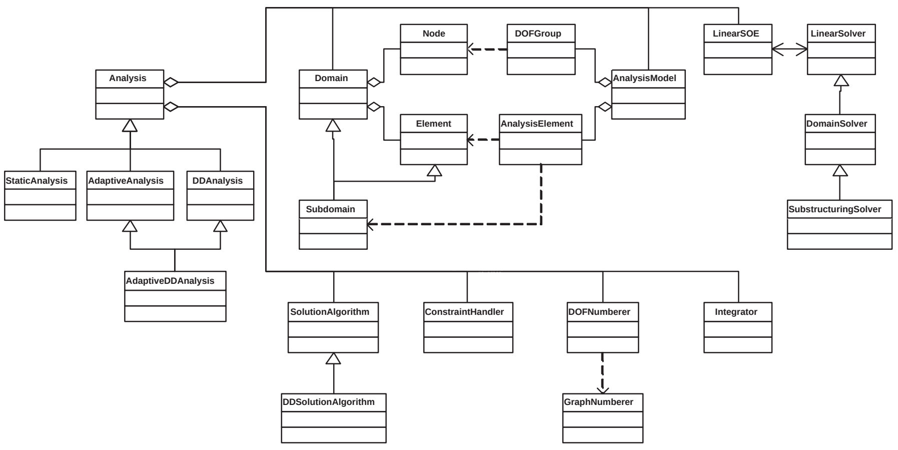

Fig. 1. The object-oriented model of analysis

(`Nodal/ElemanalLoad`, `SP/MPConstraint` and others). The `DomainBuilder` derived classes operate the data from files and CAD systems.

2. Partitioning the domain into subdomains. It is appropriate to represent a FE mesh as a graph of the element connectivity and then to apply any graph partitioning algorithm. The object of the `DomainPartitioner` class gets the element graph `Graph` built by the object of the `PartitionedDomain` class and divides it by any graph partitioning algorithm represented by the object of the `GraphPartitioner` class. The `GraphPartitioner` subclasses are based on the algorithms implemented in METIS and ParMETIS libraries (Karypis and Kumar [1998]). The `Subdomain` class extends the `Domain` interface to make distinction whether the nodes in the subdomain are internal or external. The `PartitionedDomain` and `Subdomain` derived classes are designed to partition the domain into both non-overlapping and overlapping subdomains. For additive Schwartz methods, the best of all would be the synchronous handling of the objects in the intersection; in this case the objects are instantiated once. For multiplicative Schwartz methods, it is the other way round; in each `Subdomain` object the copies of the objects that are included into the intersection are created to be independently calculated and periodically synchronized.

3. Assignment of both local and global equation numbers to nodal degrees-of-freedom. To make such a mapping there is a need to apply any graph numbering algorithm to the mesh node graph. The mapping can have a significant influence on the amount of computation required to solve the system of equations and on the amount of memory required to store it.

The `AnalysisModel` is a container for storing and providing access to the objects of the `DOFGroup` and `AnalysisElement` classes. The `DOFGroup` objects represent the degrees-of-freedom at the nodes and new degrees-of-freedom introduced into the analysis to enforce the constraints. The `AnalysisElement` objects represent the elements and subdomains or they are introduced to add stiffness and/or load to the system of equations in order to enforce the constraints. The `DOFGroups` and `AnalysisElements` remove from the `Node` and `Element` objects the need to worry about the mapping between the degrees-of-freedom and equation numbers. They also have the methods for forming tangent and residual vectors that are used to form the system of equations. Besides, they handle the constraints.

The `DOFNumberer` is responsible for mapping the numbers of equation to the degrees-of-freedom in the `DOFGroup` objects.

4. Assembling the systems of equations using elemental and nodal contributions determined by the integration scheme chosen. Assembling the systems of equations is also based on the FE graphs and on defining the contributions for the different types of element as well. According to the integration scheme, the local systems of equations are formed by the FE contributions. For the DD methods that need assembling the global system of equations, it seems efficient to represent all subdomains as the graph of special-purpose elements (superelements) in order to apply the approach stated above. Different FE contributions and various integration schemes give a wide range of assembling methods for DD. On multiprocessors the contributions from internal elements of the subdomain can be calculated on the processor that handles the subdomain; to determine them from external ones it is necessary to use different approaches to distributed computing.

The `Integrator` is responsible for defining the contributions of the `DOFGroup` and `AnalysisElement` objects to the system of equations and for updating the response quantities in the `DOFGroup` objects with the appropriate values given the solution to the system of equations.

5. Imposing boundary conditions. Applying the constraints may involve transformation of the elemental and nodal contributions or adding new terms and unknowns to the matrix equations.

The `ConstraintHandler` class is responsible for handling the constraints by creating appropriate `DOFGroup` and `AnalysisElement` objects. It also allows to introduce the multiple constraints arising from adaptive refinement.

6. Solving the system of equations. Different DD methods are similar in the presence of the local and possibly, global systems of equations and in the performance of the local and sometimes, global matrix-vector operations according to the solution algorithm.

The `Analysis` class is a container for all of the analysis objects mentioned above. It is responsible for starting the analysis steps specified in the `SolutionAlgorithm` class. The `Analysis` class is associated with either a domain or a subdomain and allows describing either a global solution or one branch of the solution in the subdomain. In the second case, several sets of analysis objects are executed simultaneously.

The `LinearSOE` class stores the matrix, the right hand side and the solution of a linear system of equations. `LinearSOE` derived classes correspond to the systems with different types of matrices (band, profile, etc). The

`LinearSolver` class is responsible for performing the numerical operations on the equations. `LinearSolver` subclasses encapsulate linear algebra libraries LAPACK, PETSc, SuperLU.

7. Update of nodal degrees-of-freedom with the appropriate response quantities.

8. Determining the rated conditions in finite elements.

Nowadays it is generally accepted that the effective solution of applied problems is almost impossible without using an adaptive process when the obtained solution is examined to determine the strategy of further calculations: mesh refinement with the same connectivity (r-version of FEM), local mesh refinement (h-version), increase of the degree of approximation basis functions (p-version) or whatever combinations (h-p, h-r versions). The best choice gives the maximum precision with the minimum computational costs. The OO model for adaptive analysis allows to build the optimal computational model with the given precision and minimum computational costs.

9. A-priori error estimation. The additional data included in the `AdaptiveAnalysis` class are the following: error estimation `ErrorEstimation`, error indicator `ErrorIndicator`, refinement strategy `Refinement`. `ErrorEstimation` subclasses represent a-priori and a-posteriori error estimations based on: residual, interpolation, projection, extrapolation, dual method.

10. Determining the objects to be more precise. The `ErrorIndicator` subclasses provide the selection of the part of the domain to refine: global refinement, strategy of maximum, equidistribution, guaranteed error reduction.

11. Repartitioning the mesh in accordance with the criterion of refinement. Mesh repartitioning gives rise to redistributing the work among the processors, with each processor busy in actual loading as long as possible, in other words, to load balancing. The main difference between the dynamic load balancing and static one is the necessity to redistribute the work among the processors; it brings about considerable computational costs (Kopyssov and Novikov [2001]).

Different improvements of the solution are inherited from the `Refinement` class: relocation of the nodes in 2D/3D area (r-version), local refinement and coarsening for 2D triangle meshes (h-version), increasing the degree of integrated Legendre polynomials for 3D hierarchical hexahedral elements (p-version).

The dynamic load balancing is implemented by including the `Refinement` object to `DomainPartitioner`. Using them on every iteration, one could efficiently redistribute the FE mesh objects, with `Refinement` providing graph weights handling, and `GraphPartitioner` partitioning the graph.

3 Extension for parallel distributed computing

To provide the interprocess communication between objects on multiprocessor computers, it is necessary to implement the remote method invocation. In addition, it is required to implement the migration of the objects representing the distributed data and the asynchronous invocation of the methods of the objects representing parallel functions. Some parallel distributed implementations of objects were examined on MPI and CORBA middleware (Kopyssov et al. [2003]). A new approach is suggested to develop parallel distributed OO software for DD. It is based on CORBA, the AMI (Asynchronous Method Invocation) callback model (Schmidt and Vinoski [1999]) and integration of MPI applications.

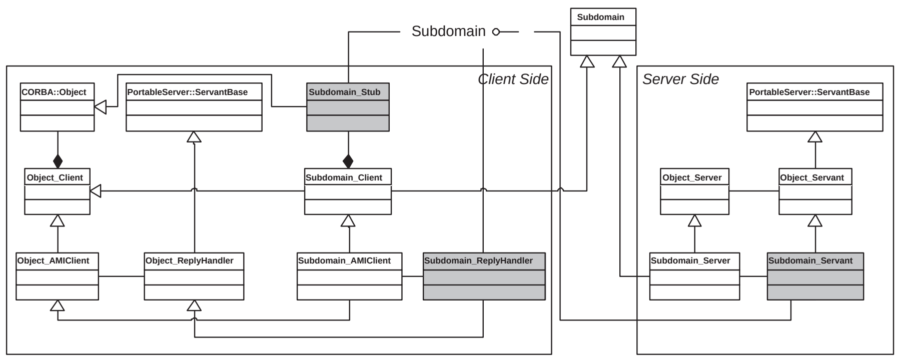

Fig. 2. CORBA implementation of *Subdomain* class

The CORBA implementation of data and function objects consists of:

IDL interface. The objects to be called remotely are specified as IDL interfaces (`interface`), with the migrated objects specified as IDL structures (`struct`). According to the IDL interface CORBA generates the C++ templates (stubs, servants, AMI handlers) designed to develop parallel distributed objects. Figure 2 shows the interface and the client-server (gray filled) templates for the `Subdomain` class.

Client-server library. It is based on the C++ library for the DD and CORBA client-server templates. The client classes are inherited from the DD ones and aggregate the stubs of the CORBA interfaces, with the virtual functions of the DD objects overloaded: the input parameters are converted from the DD types to those of CORBA, the method of the remote CORBA object is invoked, the output parameters are converted backward and returned as the result of the remote invocation. The server classes are inherited from the DD ones and aggregate the CORBA servants, with the pure virtual functions of servants implemented: the input parameters are converted from the CORBA types to those of DD, the method of the DD object is called, the output parameters are converted backward to be sent to the client side. The asynchronous method invocation is implemented with the help of the AMI templates: the client objects send the object reference to the aggregated AMI handlers within asynchronous invocation and return control to the main process. Having addressed to the result objects, the main process is blocked until the remote methods finish and return their results to the AMI handlers. Figure 2 shows the base client-server classes (denoted by `Object_` prefix) and the example of their use for implementing the client-server classes `Subdomain`.

Components. The components are the executable modules that include the CORBA objects for DD. It is available to launch the components with the help of MPI as the set of identical CORBA servers, which provide distributed data and carry out the operations in parallel via MPI communications; the server with a null identifier synchronizes all others.

The parallel distributed OO model for DD is the extension of the original model, of several objects in particular: `Subdomain` (Figure 2), `Node`, `Element`, `Nodal/ElementalLoad`, `SP/MPConstraint` and some others. There are some development principles.

Remote method invocation. The `DomainBuilder` subclass creates the server finite-element objects on the governing computing node. On other nodes the `Subdomain_Server` objects are launched, with an array of `Subdomain_-Client` objects, that aggregates the stubs to them, being created on the governing node. Thus, without any modifications of the C++ library, the DD steps could be performed in the distributed address space through the `_Client` and `_Server` objects.

Object migration. In the initial OO model for DD the `DomainPartition-er` handles the pointers to the C++ objects when it is called to distribute the data among the subdomains. The object migration is more complicated; it includes creating the destination remote server object, copying the data of the source object and removing the source one. For that, it is necessary to modify the `DomainPartitioner` class: it has to include virtual functions with an empty body to collect garbage; the `Partition` method has to include the calls to them. The `DomainPartitioner` subclass overloads these functions and thus, it removes all the transferred objects in the end of partitioning.

Asynchronous method invocation. The principal operations to be performed simultaneously in the subdomains are forming the blocks of the global system of equations (if it is need) and solving the local systems of equations. For that, the `Subdomain_AMIClient` object is instantiated; it aggregates the AMI handler `Subdomain_ReplyHandler` for this methods. Invoking the remote method, the `Subdomain_AMIClient` object does not wait for its completion and returns control to the main process. As the main process needs the results, it calls the `Subdomain_AMIClient` object that, in its turn, blocks the main process until the `Subdomain_ReplyHandler` receives the result from the remote `Subdomain_Servant` object.

4 Examples

The parallel distributed OO framework for DD is intended for representation of a wide range of DD methods by using different: types of FE; mesh partitioning algorithms; ordering, storage and solution methods for the system of equations; means of handling of boundary conditions; error estimations; refinement strategies. Let us consider the substructuring method as an example of usage of the framework. It is suited to demonstrate the features of the analysis classes for both a partitioned domain and a subdomain, and the scope for the parallel distributed computing as well.

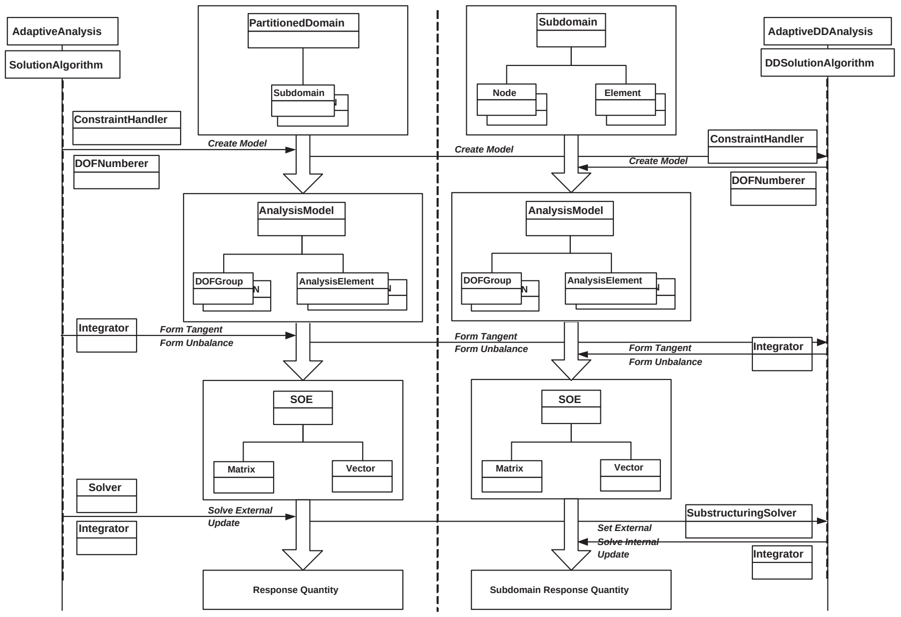

Fig. 3. The object-oriented model of substructuring

After the initial partitioning of the domain, analysis classes are used twice (see Figure 3): to solve the problem in the subdomains and on the interface. In the subdomain the objects aggregated by the `DDAnalysis` cooperate under the `DDSolutionAlgorithm` control in the following way: `ConstraintHandler` and `DOFNumberer` take `Subdomain` as the input data to create the `AnalysisModel`. The Integrator performs static condensation and forms the matrix and the right hand side for the Schur complement system block `LinearSOE` from the `AnalysisModel`.

After that, the interface problem is solved by the `AdaptiveAnalysis` object with its own `SolutionAlgorithm`, `PartitionedDomain`, `AnalysisModel`, `DOFNumberer`, `ConstraintHandler`, `Integrator`, `LinearSOE`, `LinearSolver`

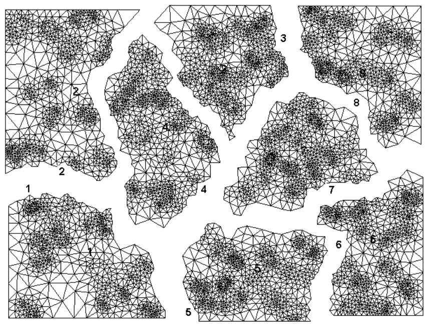

Fig. 4. Initial partition: 8 subdomains, 5959 nodes, 11740 elements

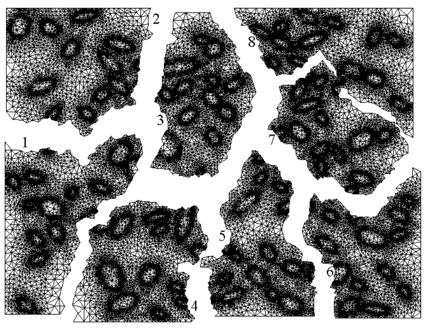

Fig. 5. Partition after 5 adaptive refinements: 42470 nodes, 84621 elements

objects. In that case, the `AnalysisModel` includes the `AnalysisElement` objects corresponding to the subdomains; the `Integrator` forms the system of equations from the interface. The `LinearSolver` object solves the global interface problem.

The `Subdomain` objects get the equations numbers from the response quantities obtained in the previous calculations on the whole `AnalysisModel`. The `SubstructuringSolver` objects solve the internal systems of equations. The `Integrator` objects update the response quantities.

The results of numerical experiments are presented in Figure 4-6. It is 2D/3D strain stress analysis; iterative/direct substructuring method is used; solution is adaptively refined by h-version/p-version of FEM.

5 Conclusions and Further research directions

The analysis allowed us to represent the main steps of DD in the form of objects and their relations. The main features of the OO framework for DD have been described. The framework was extended on CORBA middleware for parallel distributed computing. The given examples demonstrated its expressiveness and flexibility.

Further research directions are as follows:

1. extension of the OO framework for DD by new algorithms
2. inclusion of geometrical data and encapsulation of CAD systems
3. implementation of parallel algorithms of mesh generation; integration of existing mesh generators
4. development of the visual editor for DD

Acknowledgement. The work is supported by Russian Foundation for Basic Research (grants 02-07-90265, 03-07-06119, 03-07-06120, and 03-07-90002) and UB of RAS.

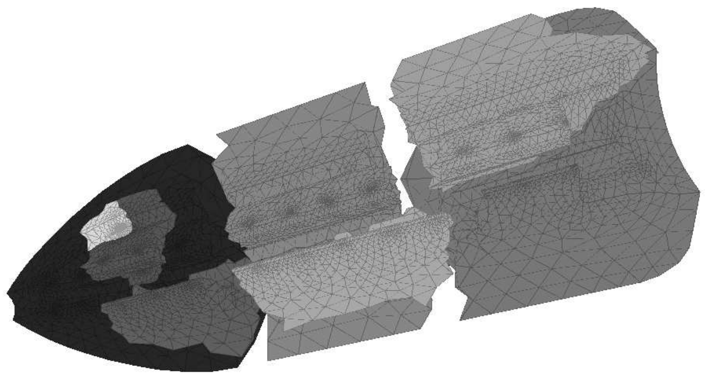

Fig. 6. Domain partition for p-version of FEM based on substructuring

References

D. Brown, W. Henshaw, and D. Quinlan. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In *SIAM conference on Object Oriented Methods for Scientfic Computing*, UCRL-JC-132017, 1999.

X. Cai. Domain decomposition in high-level parallelization of PDE codes. In *Eleventh International Conference on Domain Decomposition Methods*, pages 388–395, Greenwich, England, 1998.

A. Denis, C. Perez, T. Priol, and A. Ribes. Parallel CORBA objects for programming computational grids. *Distributed Systems Online*, 4(2), 2003.

G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

S. Kopyssov, I. Krasnopyorov, and V. Rytchkov. Parallel distributed CORBA-based implementation of object-oriented model for domain decomposition. *Numerical Methods and Programming*, 4(1):194–206, 2003.

S. Kopyssov and A. Novikov. Parallel adaptive mesh refinement with load balancing for finite element method. *Lecture Notes in Computer Science*, 2127:266–276, 2001.

J. Przemieniecki. *Theory of Matrix Structural Analysis*. McGaw-Hill, N.Y., 1968.

D. Schmidt and S. Vinoski. Programming asynchronous method invocations with CORBA messaging. *C++ Report, SIGS*, 11(2), 1999.

S. Sobolev. Schwartz algorithm in elasticity theory. *RAS USSR*, 4(6):235–238, 1936.

T. Zimmermann, Y. Dubois-Pelerin, and P. Bomme. Object-oriented finite element programming: I. governing principles. *Computer Methods in Applied Mechanics and Engeneering*, 98(2):291–303, 1992.