An Object Oriented Framework for Computational Fluid Dynamics Simulations

Freddy Perez and Wilson Rivera

Electrical and Computer Engineering Department
Parallel and Distributed Computing Laboratory
University of Puerto Rico Mayaguez
Mayaguez, PR 00681, USA
{f_perez, wrivera}@ece.uprm.edu

Abstract. This paper describes an object-oriented framework for solving computational fluid dynamics problems on parallel computers. The design and components of the framework are discussed related to design patterns methodology. The proposed framework offers higher-level programming abstractions for parallelization and improves the overall efficiency of implementation.

Keywords: object-oriented programming, pattern design, computational fluid dynamics, Java.

1. Introduction

The rapid development of parallel and distributed systems, numerical algorithms, and high-speed data networks has resulted in dramatically increased computational power and efficiency. As a result, Computational Fluid Dynamics (CFD) has emerged as an essential analysis tool applied extensively in analyzing fluid mechanics, heat and mass transfer, hydrodynamics, atmospheric sciences, solid mechanics, water quality, and transport problems. Thus in the CFD process, the partial differential equations (PDEs), which govern the problem of interest, are solved using numerical methods on high performance parallel computers [1, 2, 3]. One of the challenges in this arena is creating a flexible and open development environment that help reduce the high cost of implementing parallel codes as comparing to the traditional approach in which the application programmer handles all the implementation details. From the implementation point of view, modern programming languages offer powerful tools for flexibility, such as the inheritance of object-oriented programming. The numerical approach however should be flexible as well. Consequently, flexible domain decomposition techniques need to be implemented, without compromising the accuracy of the algorithms [4]. In addition, efficient management strategies are needed to deal with all the software components.

We propose a high-level parallelization of CFD codes through an extensive use of object-oriented programming techniques. A modular implementation of mathematical abstractions, which is a direct advantage of object-oriented programming, allows for the generalization of computational kernels, which are reusable in many simulation applications. This approach makes it possible to hide computational details when it is

needed as well as produce simulators with unified generic interfaces. In this paper, we describe our experience in building an object-oriented framework for parallel flow simulations. The description of the framework is presented with emphasis on design patterns methodology [5].

Design patterns provides a high level perspective on both the problem and the process of design and object orientation [6]. A pattern describes a core solution of a problem that occurs frequently in an environment. There are three types of patterns: Creational, structural and behavioral patterns [7]. Creational patterns create objects rather than instantiate objects directly. Examples of such patterns are *builder, factory, prototype*, and *singleton* patterns. Structural patterns help compose groups of objects into large structures such as complex user interfaces. Representatives of these patterns are *adapter, bridge, composite,* and *proxy*. Behavioral patterns, in turn, help define the communication between objects and how the flow is controlled in a complex system. Examples of such patterns are *interpreter, mediator, observer, strategy,* and *template*. The base pattern of our framework is the *builder* pattern, which creates a context to use others patterns. The *factory* pattern is used to select different approaches to solve a CFD problem. We have also used the *mediator* pattern for the communication among objects created by the *builder* pattern, and the *observer* pattern for data communication and parallelization.

This paper is organized as follows. Session 2 depicts the design and components of our framework. Session 3 discusses a study case. Finally, conclusions and future work are listed in session 4.

2. Framework Architecture

We use design patterns to define the software architecture and design of the framework. The main pattern of our framework is the builder pattern (see Figure 1). A builder pattern simplifies the creation of complex objects by defining a class whose purpose is to build instances of other classes. Since each CFD problem may have different configuration and requirements, it is needed to construct a particular complex object for each particular problem. Each complex object, referred to as Solver, is a solver for an equation on a particular mesh, using a numerical method with particular initial and boundary conditions. The class *ConcreteBuilderSolver* is responsible for the creation of Solver. The four components of Solver are the classes *Equation, Mesh, NMethod,* and *IBConditions*. Instances of these classes will be created in the class *BuilderSolver* through the methods *buildEquation(), buildMesh(), buildNMethod(),* and *buildIBCondition()*, respectively.

The object Solver is created according to information provided by the user through a graphical user interface. Such input data is obtained by the class *Director* through the method *contruct()*. The *Equation* class has a reference to the interface *ProductEquation* and defines the same methods of this interface. This definition allows for the communication between the class *Equation* and any instance of the classes that implement *ProductEquation*, such as *EulerEquation*, *HeatEquation,* and *NavierEquation*. This design allows us, when needed, include another type of

equations with the implementation of the methods defined in the interface *ProductEquation*.

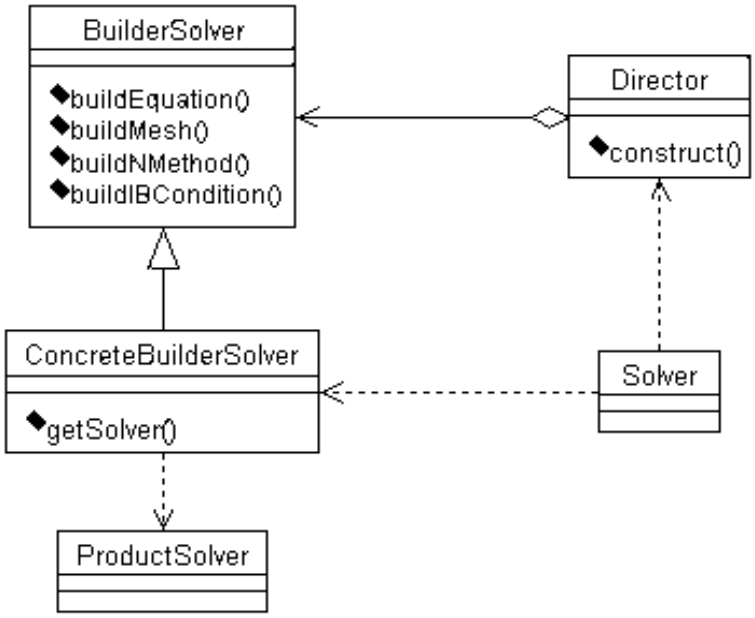

Fig. 1. Builder Pattern Implementation

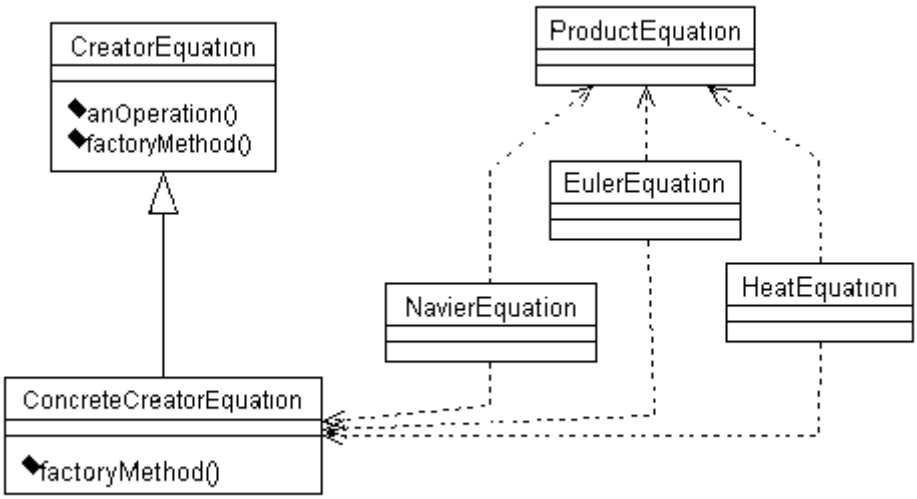

Fig. 2. Factory Pattern Implementation

The interface *ProductEquation* is implemented as a factory pattern (see Figure 2). The main idea behind factory patterns is to delegate the decision of the kind of object to be created to other subclasses. The method *factoryMethod()*, which is defined in the class *CreatorEquation* and implemented in the class *ConcreteCreatorEquation*, returns an instance of *ProductEquation* (e.g., *EulerEquation, HeatEquation* or *NavierEquation*). The methods of this instance are called using the methods of the class *Equation* because there is a reference to the class *ProductPanel*. In the same way we can obtain a specific instance of the classes that implement the interface *ProductMesh* using a factory pattern again. The class *Mesh* will be an interface between Solver and any instance of the classes *UnstructuredMesh* or *StructuredMesh*. Both classes implement the interface *ProductPanel*. With little or none modification we can obtain a specific instance of *IBCondition* using the technique described above. In order to obtain an instance of the classes ExplicitNMethod or ImplicitNMethod, certain modifications are needed according to the numerical method used to solve the equations.

The communication among the different objects is the core of the framework. If we considered direct communication among objects, we would lose modularity by a tight coupling. This problem is solved by using a mediator pattern. This pattern is a center of communication among the components that simplifies communication. The instances of *Equation, Mesh,* and *NMethod* send information to *Mediator*. On the contrary, *IBCondition* receives information from *Mediator*. The communication is carried out by the methods *sendData()* and *receivedData()* defined in the different classes, and administered by the methods in the class *Mediator* (see Figure 3).

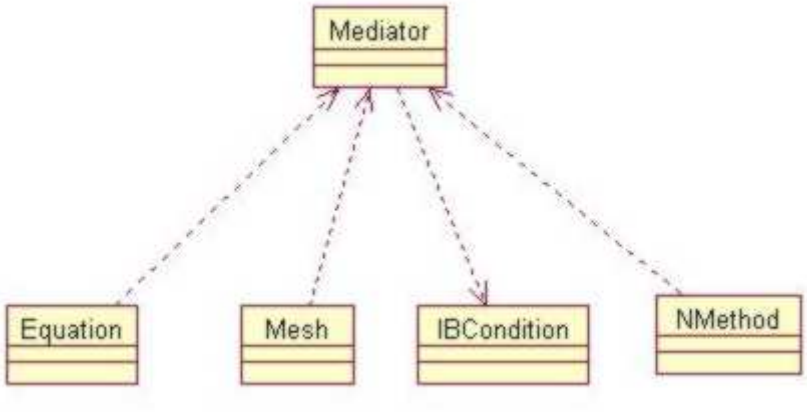

Fig. 3. Mediator Pattern Implementation

3. Case Study

To illustrate the usefulness of the proposed framework, a case study of a subsonic unsteady turbulent flow over a NACA0012 airfoil has been performed. The governing equations are the Navier Stokes equations, which can be written as:

$$\begin{aligned} &\frac{\partial \rho u}{\partial t} + \nabla.(\rho u) = 0 \\ &\frac{\partial \rho u}{\partial t} + \nabla.(\rho u \otimes u) + \nabla.(pu) = \nabla.((\mu + \mu_t)S) \\ &\frac{\partial \rho E}{\partial t} + \nabla.((\rho E + p)u) = \nabla.((\mu + \mu_t)Su) + \nabla.((\kappa + \kappa_t)T) \end{aligned} \quad (1)$$

where ρ is the density, u the velocity, T temperature, E the total energy, p the pressure, $S = (\nabla u + \nabla u^t) - \frac{2}{3} DI$ the deformation tensor, and μ and μ_t the laminar and turbulent viscosities. To model turbulence a κ-ε model is used, which can be written as:

$$\frac{\partial \rho k}{\partial t} + \nabla.(\rho u k) - \nabla((\mu + \mu_t)\nabla k) = S_k$$
$$\frac{\partial \rho \varepsilon}{\partial t} + \nabla.(\rho u \varepsilon) - \nabla.((\mu + c_\varepsilon \mu_t)\nabla \varepsilon) = S_\varepsilon \quad (2)$$

The Navier-Stokes and κ-ε equations are solved by a finite-volume Galerkin upwind technique [8] using Roe Riemann solver [9]. The viscous terms are computed using a standard Galerkin method. The computational configuration and mesh of the case study are shown in Figure 4, such as they appear in the graphical user interface. The plot of Mach number lines shown in Figure 5 is for a low-Mach number (M_∞=0.1), turbulent (Reynolds number =10^6) flow. We have obtained encouraged numerical results and performance for diverse configurations.

The architecture of the framework facilitates the implementation of different numerical methods without major efforts. The use of domain decomposition techniques as described in [4] are significantly simplified by using the *observer* pattern. This pattern governs the domain decompositions strategies and the communication between subdmoains. We have used mpiJava [10] for the implementation of the parallel algorithms. However, the framework is independent of the message passing implementation used and can be modified easily as consequence of the programming abstractions provided by the *observer* pattern.

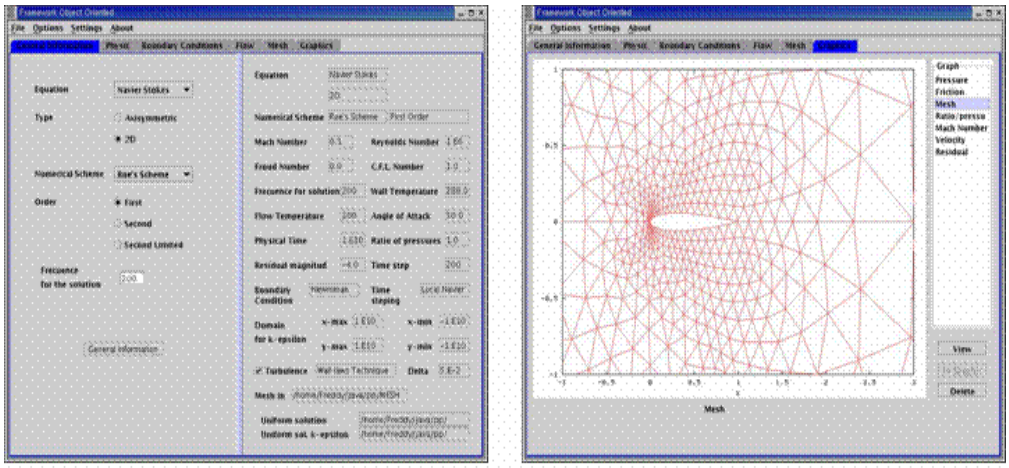

Fig. 4. Case Study General Information and Mesh

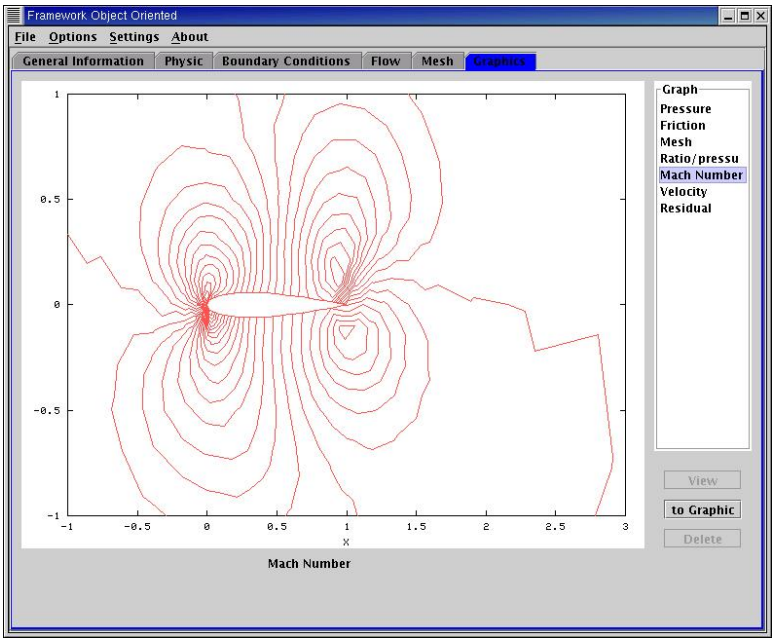

Fig. 5. NACA 0012 Airfoil Mach Number Lines

4. Conclusions

An object-oriented framework for solving computational fluid dynamics problems on parallel computers has been presented. We argue that the combination of flexible domain decomposition methods with extensive use of object-oriented techniques will result in an efficient, flexible, and systematic process for developing parallel codes. We have shown how design patterns methodology contributes to produce reusable software. The effectiveness and advantages of the framework is illustrated upon a case study of a subsonic unsteady turbulent flow over a NACA0012 airfoil. Further research and development is needed to make the framework capabilities complete and tuned for performance.

Acknowledgements

This work was supported by the UPRM-NSF PRECISE Project (EIA-NSF 99-77071).

References

1. S. W. Hammond and T. J. Barth, “Efficient massively parallel Euler solver for two-dimensional unstructured grids.” AIAA, 30(4): 947-952, 1992.
2. D. Drikakis and E. Schreck, “Development of parallel implicit Navier-Stokes solvers on MIMD multiprocessor systems," AIAA, 93-0062.
3. R. Pankajakshan and W. R. Briley, “Parallel solution of viscous incompressible flow on multi-block structured grids using MPI.” In Parallel Computational Fluid Dynamics: Implementation and Results Using Parallel Computers, Elseiver Science, 601-608, 1996.
4. W. Rivera, J. Zhu, and D. Huddleston, “An Efficient Parallel Algorithm with Application to Computational Fluid Dynamics." To appear in Computers and Mathematics with Applications.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software.” Addisson-Wesley, 1995.
6. J. Copper, “The Design Patterns Java Companion.” Addisson-Wesley, 1998.
7. A. Shalloway and J. Trott, “Design Patterns Explained.” Addisson-Wesley, 2002.
8. K. W. Morton, “On the analysis of finite volume methods for evolutionary problems.” *SIAM Journal on Numerical Analysis*, 35 (6): 2195-2222, 1998.
9. P. L. Roe, “Approximate Riemann solvers, parameter vector, and difference schemes.” *Journal of Computational Physics*, 43: 357-372, 1981.
10. M. Baker, B. Carpenter, G. Fox, S. Hoon Ko, and X. Li. mpiJava: A Java interface to MPI. In *First Workshop on Java for High Performance Network Computing, Europar '98*.