Advances in Engineering Software 47 (2012) 17–23

Contents lists available at SciVerse ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

gdbOF: A debugging tool for OpenFOAM®

Santiago Márquez Damián *, Juan M. Giménez, Norberto M. Nigro

International Center for Computational Methods in Engineering (CIMEC), INTEC-UNL/CONICET, Güemes 3450, Santa Fe, Argentina

ARTICLE INFO

Article history:
Received 5 August 2011
Received in revised form 6 December 2011
Accepted 11 December 2011
Available online 5 January 2012

Keywords:
Computational fluid dynamics
Finite volume method
OpenFOAM
Data structures
Debugging
gdbOF

ABSTRACT

OpenFOAM® libraries are a great contribution to CFD community and a powerful way to create solvers and other tools. Nevertheless in this creative process a deep knowledge is needed concerning with classes structure, for value storage in geometric fields and also for matrices resulting from equation systems, becoming a hard task for debugging.

To help in this process a new tool, called gdbOF, attachable to gdb (GNU debugger) is presented in this paper. It allows to analyze classes structure at debugging time. This application is implemented by gdb macros, these macros can access to code classes and also to their data in a transparent way, giving the requested information. This tool is tested for different application cases, such as the assemble and storage of matrices in a scalar advective–diffusive problem, non orthogonal correction methods in purely diffusive tests and multiphase solvers based on Volume of Fluid Method. In these tests several types of data are checked, such as: internal and boundary vector and scalar values from solution fields, fluxes in cell faces, boundary patches and boundary conditions. As additional features of this tool data dumping to file and a graphical monitoring of fields are presented.

All these capabilities give to gdbOF a wide range of use not only in academic tests but also in real problems.

1. Introduction

OpenFOAM® is a CFD library that allows users to program solvers and tools (for pre-processing or post-processing) in a high-level specific language. This high-level language refers to the fact of writing in a notation closer to the mathematical description of the problem, releasing the user from the internal affairs of the code.

This programming approach contrasts with procedural languages approach, such as Fortran, that are widely used in academic and scientific environments but oriented to the low-level problem resolution, i.e., the manipulation of individual floating-points values. Thus, in order to achieve the abstraction from the low-level coding it is necessary to follow another way, so that the Object-Oriented Programming (OOP) paradigm is selected. This methodology produces code which is easier to write, to validate and to maintain compared with purely procedural techniques. Respect to OpenFOAM® it is completely written is C++. This language is less rigorously object-oriented than the others languages (such as SmallTalk or Eiffel), due to the inclusion of some characteristics that are not strictly object-based. The main add-on is operator overloading, which is essential to working with tensor, vector and scalar fields objects concepts as in the mathematical notation. On the other hand, it is a multiplatform language and, due to that it is based on C, is as fast as any other procedural languages [1].

There are five fundamental concepts in OOP, whereby OpenFOAM® achieves its objectives: *modularization*, *abstraction*, *encapsulation*, *inheritance* and *polimorphism* [2]. All of them are widely used in the code. Polymorphism is a key concept in OpenFOAM®, which is clearly demonstrated by the proliferation of virtual methods (methods that must be implemented in child classes). Examples of this include the implementation of boundary conditions, which inherit from a base class `patchField`, so they have the same interface but different implementations. Another example is the representation of tensor fields: in this case `geometricField` is the parent class and various tensor fields inherit from it: `scalarField` (rank 0), `vectorField` (rank 1) and `tensorField` (rank 2), each one implementing the interface provided by the parent class in different ways.

In addition to these OOP features, there are other tools of the C++ language which are not strictly object-based and those are used in OpenFOAM®. They are the aforementioned operator overloading and the use of preprocessor macros. Macros allow to insert code directly in the program, avoiding the overhead of invoking a function (passing parameters to the stack, do a jump, take parameters), without losing the code readability [3].

As it was mentioned, using these techniques a library oriented to high-level development is generated, ensuring that the user only

* Corresponding author.
E-mail address: santiagomarquezd@gmail.com (S. Márquez Damián).
URL: http://www.cimec.org.ar (S. Márquez Damián).

0965-9978/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.advengsoft.2011.12.006

has to take care about the model to solve and not other details of coding [4]. On the other hand, some problems could arise in the application creation stage yielding to undesired results. There begins the code debugging work, and this includes monitoring values corresponding to variables involved in the resolution, such as, tensors, vectors and/or scalar fields defined at cell or face centers, coefficients in the system matrix, and many other examples. In addition, debugging is not ever motivated by problems, but simply for exploratory or control purposes [5].

From the side of debugging tools in GNU-Linux platforms, *gdb* (GNU-debugger) is the *defacto* standard. It includes a variety of tools for code analysis and data inspection at run-time [6] which gives a successful environment for OpenFOAM® debugging. gdb offers a powerful print command likely to inspect arrays in memory, nevertheless it can be used directly only in simple data structures like `lists` or `Fields`. Data examination gets hard when viewing the desired data involves polymorphism and inheritance connected with the virtual methods used by the library. This work requires to walk through the general class tree looking for the attributes which are wanted to be inspected. Moreover, once desired attributes are found, these maybe do not directly represent the information required by the developer. In the case of the matrices generated by `fvm` methods, they store the coefficients using the LDU Addressing technique (see gdbOF User's Manual, Appendix A[1]), so it is necessary to apply a decoding algorithm to transform it into the traditional format (full or sparse), and to control and operate with their values.

The main objective of the *gdbOF* tool is to solve problems like those explained in the previous paragraph. This tool is implemented by gdb macros and it is based on an implementation of gdb macros for STL (Standard Library for C++) debugging [7]. These macros simplify the task of debugging the OpenFOAM® libraries, performing the work actions transparently to the user: the simple call of a gdb macro from console triggers a sequence of actions that include: navigate the OpenFOAM® class tree, collect information and reorder it for representation in an user readable format. Moreover, *gdbOF* includes the option of writing the output into a file on disk and to view it graphically. This output is formatted appropriately to be imported in numerical computation software such as Octave or Matlab®, thus allowing the developer to expand the possibilities of data inspection at debugging time.

In this work the design concept of the tools will be presented and several cases will be solved as examples of use. These problems not only emerge in an academic context but also occur in real application environments: the first consists in a scalar advective–diffusive problem in which the emphasis will be placed on the assembling and storage of matrices; the second consists in a non-orthogonal correction method in purely diffusive tests; and the third is an analysis of multiphase solvers based on Volume of Fluid Method. The last examples are focused in volumetric and surface data inspection both in array and graphical format.

2. Basic debugging

One of the most common tasks in the debugging process is to look at the values stored in an array, that is possible in gdb with the command of Example 1, where `v` is the array to analyze.

Example 1 View array.

```
$(gdb) p *v@v_size
```

Nevertheless, as it was pointed out in the previous section, data inspection in OpenFOAM® requires often more complex sentences. A typical example is to verify at debugging time that a certain boundary condition is being satisfied (typically when the boundary condition is coded directly in the solver and the next field information is obtained after solving the first time-step). Boundary conditions in OpenFOAM® are given for each patch in a `GeometricField`, then, assuming that the inspected patch is indexed as 0 (the attribute `BoundaryField` has information of all the patches), sentence presented in Example 2 is needed to observe the values on this patch, where `vSF` is a `volScalarField`.

Example 2 View Boundary Field values.

```
$(gdb) p *(vSF.boundaryField_.ptrs_.v_[0].v_)
    @(vSF.boundaryField_.ptrs_.v_[0].size_)
```

Note that the statement in Example 2 does not include any call to inline functions, which could generate some problems in gdb, giving even more complex access to information.

gdbOF solves the inconvenience of knowing the attribute's place and using long statements. Using *gdbOF* commands, as it is shown in Example 3, the same results are obtained. Note the simplification of the statement, this is the *gdbOF* spirit, reducing the work needed to debug and perform the same tasks more simply and transparently.

Example 3 View Boundary Field values with gdbOF.

```
$(gdb) ppatchvalues vSF 0
```

There are many examples in OpenFOAM® like the previous one in which the necessity of a tool that simplifies the access to the complex class diagram can be useful. Note that in the last example it was not mentioned how the index of the desired patch was known. Usually OpenFOAM® user knows only the string that represents the patch, but not the index by which it is ordered in the list of patches. Here *gdbOF* simplifies the task again, providing the `ppatchlist` command which displays the list of patches with the corresponding indexes. Regarding to other basic gdfOF tools please refer to the gdbOF User's Manual, Chapter 2.

3. Advanced debugging

3.1. System matrix

Increasing the complexity of debugging, there can be found cases involving not only the search and dereference of some plain variables. A typical case is the dumping of the linear system, $Ax = b$, generated by the discretization of a set of differential equations which are being solved. This is stored using the *LDUAddressing* technique which takes advantage of the sparse matrix format and saves the coefficients in an unusual way. This storing format and the necessity of accessing to individual matrix coefficients lead to trace the values one by one and to apply a decoding algorithm. There are two commands to do this task, one to dump the data as full matrices and the other to dump the data as sparse matrices.

In order to implement the necessary loops over the matrix elements, gdb provides a C-like syntax to use iterative (while, do-while) and control structures (if, else). These commands have a very low performance, so the iteration over large blocks of data

[1] http://openfoamwiki.net/index.php/Contrib_gdbOF.