Advances in Engineering Software 42 (2011) 735–742

Contents lists available at ScienceDirect

Advances in Engineering Software

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

An experimental approach to the performance penalty of the use of classes in Fortran 95

Luit J. Slooten [a,*], Fransisco Batle [b], Jesus Carrera [c]

[a] Hydrogeology Group, Polytechnical University of Catalonia, Jordi Girona 1-3, Edificio D2, 08034 Barcelona, Spain
[b] Geomodels, Parc Cientific de Barcelona, Edifici Florensa, Universitat de Barcelona, C/Adolf Florensa s/n, 08028 Barcelona, Spain
[c] Jaume Almera Institute, Llus Sol i Sabars s/n, 08028 Barcelona, Spain

ARTICLE INFO

Article history:
Received 8 January 2008
Received in revised form 28 January 2010
Accepted 11 May 2011
Available online 15 June 2011

Keywords:
Fortran
Object-oriented
Performance
Abstraction
Software design
Finite elements

ABSTRACT

Fortran 95 is used often for "number crunching": scientific and engineering applications where performance is important and which operate with large datasets. The language allows the implementation of certain elements of object oriented design, which facilitate code expansion, reuse and maintenance. In this paper we discuss two series of tests to measure how different object oriented design elements of Fortran 95 affect overall performance. The first series of tests consists of several implementations for multiplying two matrices. These tests are focused exclusively on computation time, not considering other parts of the object life cycle such as construction and destruction. The second series consists of computing a finite element matrix for a diffusion term. A more complex environment with different classes is studied. Here, we consider not only the time spent doing useful computations but the integral object life cycle. Results show that object oriented design comes at a cost in all cases. However, given the right compiler, using the right compiler optimization techniques and keeping the amount of objects and method calls low, Fortran 95 designs can benefit from object oriented design techniques with a less than 10% running time increase.

1. Introduction

Mathematical modelling software is commonplace in many branches of science and engineering, ranging from hydrogeology to industrial design. The process of writing, debugging and testing such software is often costly and lengthy. The same can often be said about the simulations the end user makes with the product: these may have long running times. This implies that finding a proper trade-off between reusability and performance is especially important. The object oriented programming (OOP) paradigm has an excellent and proven reputation of allowing reusability and extensibility ([15,6]). In OOP, problems are solved by representing the different aspects of the problem to be solved by classes. This can be done on all levels: on high levels (a class representing an optimization problem), on intermediate levels (a class representing a matrix) or on low levels (a class representing a real number). However, it is well known that the use of small classes can cause an important deterioration in performance. This extra cost associated to abstraction is compiler-dependent and diminishes with increasing global optimization during compilation. A test problem known as the Stepanov benchmark ([16,19]) was written for C++ with the aim of quantifying the cost of abstraction.

It is well known that the Fortran programming language can be used to implement certain elements of object oriented designs (see e.g. [1,12,21]) and that its compilers are widely respected for the quality of their optimization. These two facts suggest that Fortran programs could conceivably benefit from object oriented design elements at a relatively low cost.

The objective of this paper is to explore the performance deterioration caused by OO design in Fortran. In this work, Fortran 95 was used, which contains enough support for OOP to be able to benefit from it. Still, it is not very common to use OO design elements in Fortran 95 applications. This is reflected by the fact that benchmarks for Fortran 95 compiler testing hardly use OO design elements (e.g. [13,22,10]).

First, a set of synthetic tests loosely inspired on the Stepanov benchmark is discussed to study the abstraction penalty associated to different designs. A matrix–matrix multiplication is gradually made more reusable, by representing the matrix as a data type, gradually encapsulating it and finally representing the matrix elements themselves as encapsulated data types. In the second part, we test three designs on a code that computes a finite element matrix. The implementations are compared in terms of performance and potential for expansion and reuse. One of the implementations

* Corresponding author. Tel.: +34 93 401 7247; fax: +34 93 401 7251.
E-mail address: luitjan.slooten@upc.edu (L.J. Slooten).

is in purely procedural Fortran 77-style, whereas the other two have been programmed with a varying degree of object oriented orthodoxy.

2. An overview of OOP

This section contains a summary of OO concepts. It is provided for the sake of completeness, and can be skipped by readers familiar with these subjects. For others, useful books on the subject of OOP are e.g. [4,3]. Additional information on the use of Fortran 95 for OOP in general can be found in e.g. [12,1,21,17]. Information on how to use design patterns in Fortran can be found in [9].

In the object oriented paradigm, the problem that is to be solved by a program is represented by "objects". An object can be anything, from a matrix or an equation to a screen window. Each of these objects has associated data (its "attributes") and functionality (its "methods"). An abstract definition of the attributes and methods of objects of a certain type is called a "class" and serves as a blueprint of objects. It is possible to define a class A in terms of another class B. This can be done either by including objects of type B in the type defined in class A (*Composition*) or by defining the class A as a *specialization* of class B. In this case, class A will be supposed to store the same data and show the same behavior as class B, unless otherwise stated. This is known as *inheritance*. Thus, the programmer is only required to program the differences between A and B. In this example, B is termed the *base class* of A.

Encapsulation is the practice of strictly separating interfaces from implementation. In other words, the way of accessing data or functionality from objects is separated from the actual implementation. The motivation for this is that this implementation may change when the code is maintained or expanded. Encapsulation limits the effect that these changes will have on the rest of the code.

2.1. Implementing object oriented designs in Fortran

Fortran 95 does not explicitly support OO concepts such as classes, objects or encapsulation. However, classes as described above can be made using Fortran modules. In a module, one can define an "abstract data type" as a tuple of variables of standard Fortran types, and other abstract data types. Additionally, the module allows defining subroutines and functions that have variables of the abstract data type as formal parameters. Finally, the module allows specifying interfaces for these subroutines. These three capabilities are everything that is needed to implement classes. In the rest of this paper, we will use the word "class" to refer to a Fortran module with an abstract data type, subroutines and interfaces. Encapsulation of data can be achieved in Fortran by using the private attribute on abstract data type members. The result of this is that while users of a module may still declare variables of the type defined in the module, they may not directly access its attributes. Encapsulation of functionality can be achieved through the interfaces defined in the modules.

Implementing composition of classes in Fortran is straightforward: it can be done by constructing abstract data types that contain other abstract data types as attributes. Inheritance is not supported, and can be emulated in part as discussed in [8]. This way of achieving inheritance is less powerful than the way inheritance is supported in many pure OO languages, and costs a bit more work. However, the main advantages of specialization inheritance are preserved: the specific specialization type of an instance of B can be set during runtime, and the code re-use associated to this kind of inheritance is preserved. As such Fortran programs can benefit from it.

2.2. Limitations of Fortran 95

Fortran 90/95 allows no parametric polymorphism. This means, that it is impossible to write code without specifying the type of the variables which appear in the code. This is a drawback, as parametric polymorphism is often used for implementing tools of general use ("foundation classes"), such as linked lists, stacks, and iterators. In Fortran, such constructs must be made for each data type separately.

It is also impossible to establishing circular module references. If a module A contains a use statement for module B, then module B cannot contain a use-statement for module A. When using modules to define classes, this means that a class A cannot use any object or method defined in module B if module B contains a use-statement of module A. This lack of circular referencing must be taken into account starting from the earliest design phase of a program.

Although Fortran 95 allows the use of pointers, and a variable of type A can be declared as being a pointer to an object of type A, there is no easy way to define an array of pointers. Declaring a variable to be an array and to be a pointer, has the effect of making the variable a pointer to an array, not an array of pointers. This can be overcome by making a data type B containing as its sole attribute a pointer p_A to a variable of type A. By creating an array of variables of type B an array of pointers to objects of type A is obtained. An unfortunate consequence of A and B having different type signatures is that subroutines operating over an array of type A cannot be used to operate over an array of type B.

Fortran 95 is a modern programming languages with an unusual richness in intrinsic mathematical functions. Many of these functions can operate both over scalars and over arrays, improving readability by making many loop structures unnecessary. It allows the use of pointers, dynamic allocation of memory and supports the use of modules. Furthermore, user defined procedures can be written in terms of scalars, and if they have no side effects, can be declared "elemental" and can then be used over arrays.

The limitations of Fortran with respect to OO language features limit the amount of abstraction that can be achieved in Fortran code. However, the features that it does implement or that can be emulated, can be put to use to implement object oriented designs and to benefit from the advantages associated to this. The true strengths of Fortran do not lie in the field of pure OO programming, but rather in functionality that can be put to use both in a procedural and in a OO environment. The most important of these are the array language and the large amount of intrinsic mathematical functions. A discussion on the choice of programming languages in scientific applications is given in [14]. Comparisons between C, C++ and Fortan90 can be found in terms of performance in [20], whereas a more qualitative comparison is presented in [5]. The conclusions reached in these last two papers are somewhat in disagreement, underlying the complexity of the issue.

3. Optimization

The aim of compiler optimization is usually to maximize the execution speed of the compiled program. Applying optimization reduces the amount of technical know-how required for making fast applications, allowing the programmer to favor more readable and intuitive constructs. When compiler optimization is applied, the set of instructions contained in the executable program stops being a literal "translation" of the source code, and rather becomes an "equivalent" set of instructions, in the sense that the result is the same. As object oriented programming is known to have a certain extra cost ("overhead") associated a.o. to an increase in function calls, the subject of optimization is important because it is