A Catalog and Two Possible Classifications of Fortran Refactorings

Mariano Méndez[1] Jeffrey Overbey[2] Alejandra Garrido[1,*]
Fernando G. Tinetti[1,†] Ralph Johnson[2]

[1] Facultad de Informática
Universidad Nacional de La Plata
50 y 120, La Plata, Buenos Aires, Argentina

[2] Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave., Urbana, Illinois, USA

Technical Report FR-UIUC/UNLP-2010-1

July 2010

Abstract

This paper presents a catalog of code refactorings that are intended to improve different quality attributes of Fortran programs. We classify the refactorings according to their purpose, that is, the internal or external quality attribute(s) that each refactoring targets to improve. We sketch the implementation of one refactoring in Photran, a refactoring tool for Fortran.

1 Introduction

In 1956, the first draft of The IBM Mathematical Formula Translating System was finished [3]. This first version of Fortran (called FORTRAN I) was the start of a complex evolutionary process. This process led to many different versions of the Fortran language, each of them with features required by the historical moment. Various features were incorporated during its evolution: subprograms (FORTRAN 66), an improved set of control structures to support structured programming (FORTRAN 77), modules and

[*] also LIFIA and CONICET Argentina
[†] also III-LIDI and Comisión de Investigaciones Científicas de la Prov. de Bs. As.

pointers (Fortran 90/95), object-oriented capabilities (Fortran 2003), submodules and co-arrays (Fortran 2008) [14]. For this evolution to be practical, the backward compatibility with the older versions of the language was essential [14]. Annex B of Fortran 2008 standard (ISO/IEC 2008) enumerates the obsolete features of the language that have not been deleted, some of which may be located in the FORTRAN 66 and FORTRAN 77 specification. Even when obsolete features were deleted, compatibility remained: "Unlike Fortran 90, Fortran 95 was not a superset; it deleted a small number of so-called obsolescent features. This incompatibility is more theoretical than real however, as all existing Fortran 95 compilers include the deleted features as extensions" [5].

Over so many years of evolution, program maintenance becomes challenging. Many operations can be written in three or four different ways. For example, a Loop that initializes a matrix may be written in four different ways:

```
....                    ....
    DO 110 I=1,30           DO 100 I=1,30
    DO 100 J=1,30           DO 100 J=1,30
    MATRIX(I,J)=0       100 MATRIX(I,J)=0
100 CONTINUE            ....
110 CONTINUE            ....
...                     ....

....                    ....
DO I=1,30               ....
   DO J=1,30            ....
     MATRIX(I,J)=0      MATRIX(1:30,1:30)=0
   END DO               ....
END DO                  ....
....                    ....
```

The magnitude of maintenance tasks is increased not only by the evolution of versions, but also by the large amount of Fortran code in production and the importance of Fortran as a programming language in several disciplines such as meteorology, physics and mathematics.

Refactoring is a technique used to improve internal qualities of the code like readability, flexibility, understandability and maintainability [8]. It is applied interactively on code with "bad smells" like duplication and lack of parameters [8], and after a series of small transformations, it beautifies the code preserving its behavior. Using refactoring, developers and maintainers can manage the code and then extend it with new functionality. In the case of Fortran, refactoring can make substantial improvements to readability and maintainability, and it can also modernize the code by replacing obsolete constructs with newer alternatives. [14]. Moreover, refactoring may be used to improve external qualities like performance [15], which is highly

beneficial in the case of Fortran since it is used mostly for high performance computing. However, the goals of high performance many times seem to oppose other goals of software engineering. For example, a transformation like loop unrolling will definitely worsen readability and maintainability of the code. The focus is not on these kinds of compiler optimizations, but on refactorings that improve maintainability as well as maintain high performance, and vice versa. Even more, refactorings that improve readability, an internal quality, may be applied to gain a better understanding of the program and then improve performance by parallelization.

The real value of refactoring comes from its automation and its integration in development environments. Manual refactoring is time-consuming and, also, prone to errors. The first refactoring tool was built for the Smalltalk language, but it was not until it was integrated into the Smalltalk browser that it became successful [16]. Nowadays there are several refactoring tools for Java like the one integrated in Eclipse JDT [7] and Idea [10]. The situation is not different for Fortran. Refactoring is almost impossible without a tool that can analyze the preconditions under which a refactoring is "safe" and then performs the transformations without breaking the code. For this purpose, refactoring engines parse the code, create a program representation based on abstract syntax trees, and perform analysis and transformations on the trees. Syntax trees contain enough editing information to be able to pretty-print the trees as close as possible to the original code. Photran is an Eclipse-based Fortran IDE [2], which started as a project of the University of Illinois at Urbana-Champaign and it is now an open-source project with many contributors. Photran has a sophisticated infrastructure for development and refactoring, and new refactorings are being added to it. In this paper we propose a catalog of refactorings and a classification in terms of purpose. It is also possible to list the refactorings that have been already implemented in Photran, the ones that are under construction and those which has not been started.

The structure of this technical report is as follows. The next section describes the main features that characterize Fortran programs. In Section 3 we propose a catalog of refactorings for Fortran programs. Section 4 describes Photran in detail and how it is like to implement refactorings on its infrastructure. Section 5 presents related work and finally Section 6 presents conclusions and future work.

2 Characteristics of Fortran Programs

Fortran is one of the most ancient programming language still being used. Fortran programs have a combination of

- Old-style Fortran language constructs, such as those designed in the early stages of the language, up to the '70s.

- Old-style software design methodology or no software development methodology at all. This lack of methodology has been partially mitigated by the strong relationship among programs and mathematical methods implemented.

Fortran evolution has resulted in a wide range of equivalent syntactical constructions. From those equivalent constructions, the older ones (coming from old language version/s) have many disadvantages/drawbacks. Programmers do not need to be aware of all these variations and/or Fortran's dialects in an academic course about Fortran programming, but the scenario radically changes if a programmer is working on a twenty year old application that has been written by others in FORTRAN 77 [14, 17].

However, not all Fortran code is legacy code. Fortran has gained a leading role in the High Performance Computing world throughout the years. High Performance Fortran is an extension of Fortran 90 that supports parallel/vector computing [11]. Co-Array Fortran is an extension of Fortran 95 supported by Cray compilers [1]. Currently, old Fortran programs need to be made more efficient in multiprocessing systems with multi-core architectures [18]. Furthermore, multi-core processors are making single threaded (or, directly, sequential) software obsolete, such as most of the legacy Fortran programs.

Other characteristics of old Fortran programs, such as using COMMON blocks for saving memory, give raise to numerous problems for identifying data as global or local to each subroutine. Automated and graphics tools for Fortran has not been used extensively, and refactoring is a good scenario to introduce and use tools such as Photran in daily software programming/maintenance work.

3 A Catalog of Fortran Refactorings

This section presents a catalog of refactorings for Fortran code. This list of refactorings does not intend to be exhaustive but we aim at providing a complete classification of refactorings according to their specific purpose. Classifying Fortran refactorings by purpose is not easy since a refactoring may belong to more than one category, and we need to decide where it provides the most benefit. However, we think it is worth the effort so developers can make a better decision at selecting the most advantageous refactoring for their needs. We have found two categories of Fortran refactorings: *Refactorings to Improve Maintainability* and *Refactorings to Improve Performance*. Each one of these classes may be divided into subclasses. This categorization is not the only possible one. Many classical refactorings have been intentionally omitted from this list since they are widely described in the literature [12, 8], although they fit into this categorization as well.

3.1 Refactorings to Improve Maintainability

The refactorings in this category are intended to improve internal quality attributes of the code such as: readability, understandability and extensibility (attributes that refactoring has been recognized to improve) and also refactorings that allow upgrading the code to newer versions of Fortran, removing obsolete features.

- **Refactorings to Improve Presentation/Readability:**
  - ***Rename:*** change the name of a variable, subprogram, etc.
  - ***Change Keyword Case:*** change the case of keywords in the source code.
  - ***Extract Local Variable:*** remove a subexpression from a larger expression and assign it to a local variable.
  - ***Extract Internal Procedure:*** remove a sequence of statements from a procedure, place them into a new subroutine, and replace the original statements with a call to that subroutine.
  - ***Canonicalize Keyword Capitalization:*** make all applicable keywords the same case throughout the selected Fortran program files.

- **Refactorings to Facilitate Design/Interface Changes:**
  - ***Encapsulate Variable:*** create getter and setter methods for the selected variable.
  - ***Make* Private *Entity* Public*:*** switch a module variable or subprogram from Private to Public visibility.
  - ***Change Subprogram Signature:*** allow the user to add, remove, reorder, rename, or change the types of the parameters of a function or subroutine, updating call sites accordingly.
  - ***Add* Only *Clause To* Use *Statements:*** create a list of the symbols that are being used from a module, and adds it to the Use statement.
  - ***Move Entity Between Modules:*** move a module variable or procedure from one module to another and adjust Use statements accordingly.

- **Refactorings to Avoid Poor Fortran Coding Practices:**
  - ***Remove Unreferenced Labels:*** delete a label if it is never referenced.

- ***Remove Real Type Iteration Index:*** change non-integer Do parameters or control variables.
- ***Remove Reserved Words As Variables:*** rename variables named equal to Fortran reserved keywords.
- ***Introduce* Implicit None*:*** add Implicit None statements to a file and add explicit declarations for all variables that were previously declared implicitly.
- ***Introduce Intent* In/Out*:*** introduce intent In or Out in each variable declaration within functions and subroutines.
- ***Remove Unused Local Variables:*** remove declarations of local variables that are never used.
- ***Minimize* Only *List:*** delete symbols that are not being used from the Only list in a Use statement.
- ***Make* Common *Variable Names Consistent:*** give variables the same names in all definitions of the Common block.
- ***Delete Unused* Common *Block Variable:*** remove unused variables declared in a Common Block.
- ***Add* Dimension *Statement:*** add the Dimension statement to declare an array.
- ***Remove* Format *Statement Labels:*** replace the format code in the read/write statement directly, instead of specifying the format code in a separate format statement.

- **Refactorings to Remove Outdated, Obsolete and Non-Standard Constructs:**
  - ***Replace Obsolete Operators:*** replace all uses of old-style comparison operators (such as .LT. and .EQ.) with their newer equivalents (symbols such as ¡ and ==).
  - ***Change Fixed Form To Free Form:*** change Fortran fixed format files to Fortran free format files.
  - ***Transform* Character* * *to* Character(Len =) *declaration:*** replace Character* with the equivalent Character(Len =) for string declaration.
  - ***Remove Computed* Go To *statement:*** replace a computed Go To statement with an equivalent Select-Case construct containing Go To or if possible remove the Go Tos statement entirely.
  - ***Remove Arithmetic* If *Statement:*** replace an old arithmetic If statement, being analogous to removing computed Go To.

- ***Remove Assigned* Go To*s:*** remove assigned Go To statements.
- ***Replace Old Styles DO loops:*** replace old styles Do Loop Continue with the equivalent Do Loop with End Do statement.
- ***Replace Shared* Do *Loop Termination:*** replace all shared Do Loop termination construct with the equivalent Do Loop with End Do statement.
- ***Transform To* While *Sentence:*** remove simulated While made by If and Go To statement.
- ***Move* Common *Block to Module:*** remove all declarations of a particular Common block, moving its variable declarations into a module and introducing Use statements as necessary.
- ***Move* Saved *Variables To* Common *Block:*** create a Common block for all saved variables of a subprogram.
- ***Convert* Data *To* Parameter*:*** change a Data declaration to Parameter declaration making more clear which variables are constant and which ones are not.

3.2 Performance Refactorings

This category currently has two examples of how refactoring can be used to improve performance while preserving not only the behavior of the program but also the readability and maintainability of the code. This is one of the factors that sets refactoring apart from optimization.

- **Refactorings For Performance**
  - ***Change To Vector Form:*** rewrite a Do Loop into an equivalent Fortran vectorial notation, which allows the compiler to make better optimizations [18].
  - ***Interchange Loops:*** swap inner and outer loops of the selected nested do-loop, in the case that doing so optimizes memory access pattern and allows to take advantage of data prefetching techniques.

3.3 Another Categorization

Some of the refactorings proposed in this catalog are currently in process of implementation or were implemented in a development and refactoring tool for Fortran called Photran. Photran is described in the next section. Taking this into account, we may also categorize refactorings by its degree of implementation: *Finished*, *In Progress*, and *Planned*. Tables 1, 2 and 3

list the refactorings in each of these categories respectively.

Table 1: Finished Refactorings

Replace Obsolete Operators
Canonicalize Keyword Capitalization
Change Keyword Case
Introduce Implicit None
Rename
Interchange Loops
Encapsulate Variable
Make COMMON Variable Names Consistent
Move Saved Variables To COMMON Block
Extract Local Variable
Extract Procedure
Make Private Entity Public
Remove Unused Local Variables
Minimize ONLY List
Add ONLY Clause To USE Statements
Data To Parameter

Table 2: In Progress Refactorings

Change Fixed Form To Free Form
Replace Old Styles Do loops
Replace Shared Do Loop Termination
Remove Unreferenced Labels
Add Parameter To SubProgram
Introduce Intent In / Out
Replace COMMON With Derived Type
Add Public Module to COMMON Block
Move Entity Between Modules

Table 3: Planned Refactorings

Remove Arithmetic IF Statement
Transform CHARACTER* to CHARACTER(LEN =) declaration
Remove FORMAT Statement Labels
Add DIMENSION Statement
Remove Real Type Iteration Index
Remove Reserved Words As Variables
Remove Computed GO TO statement
Remove Assigned GO TOs
Transform To While Sentence
Change To Vector Form
Delete Unused COMMON Block Variable

4 Photran: A Refactoring Tool for Fortran

Photran is an advanced, multiplatform integrated development environment (IDE) for Fortran based on Eclipse. Photran has a number of powerful features. As an IDE, it integrates editing, source navigation, compilation, and debugging into a single tool. It uses *make* for compilation, which allows it to work with virtually any existing Fortran compiler; so-called *error parsers* are provided which interpret the error messages from popular compilers, associating error markers with the appropriate lines of code. *Language-based searching* allows a Fortran programmer to quickly find a subprogram or module with a particular name, or to find all of the references to a particular variable or subprogram. From the beginning, Photran was designed to support refactoring, and much of its development effort has focused on providing a robust refactoring infrastructure. Version 6.0 (released June, 2010) contains 16 refactorings, and many more are under development. The development version of Photran provides name binding, control flow, and basic data flow information to support precondition checking.

4.1 Building New Refactorings

Photran divides refactorings into two categories: An *editor-based refactoring*, which requires the user to select part of a Fortran program in a text editor in order to initiate the refactoring, and a *resource refactoring* which applies to entire files. To create a new refactoring, the developer must decide whether it will be an editor-based refactoring or a resource refactoring. Photran provides different superclasses for each. The developer then creates a concrete subclass and adds a line of XML to a configuration file to make Photran aware of the new refactoring. The concrete subclass must define methods which first provide the name of the refactoring. This becomes

its label in the Refactor menu it is also used to describe the refactoring in the Edit > Undo menu item and in other user interface elements. Second, check initial preconditions. These are usually simple checks which verify that the user selected the correct construct in the editor, that the file is not read-only, etc. Third, it is necessary to acquire user input. For example, a refactoring to add a parameter to a subprogram must ask the user to supply the new parameter's name and type. and check final preconditions. These validate user input and perform any additional checks necessary to ensure that the transformation can be performed, the resulting code will compile, and it will retain the behavior of the original program. And Finally, perform the transformation. Once all preconditions have been checked, this method determines what files will be changed, and how. Thanks to the XML configuration file and Java's reflective facilities, much of the user interface for a refactoring comes "for free." Then Photran automatically adds the refactoring to the appropriate parts of the Eclipse user interface, and it provides a wizard-style dialog box which allows the user to interact with the refactoring. This dialog includes a *diff*-like preview, which allows the user to see what changes the refactoring will make before committing it.

4.2 Example: Replace Old-Style Do-Loops

One refactoring we implemented is called Replace Old-Style Do-Loops [14, 18]. There are many different ways to write a do-loop in Fortran, depending on what version of Fortran is being used. "Old-style" do-loops contain a numeric statement label in the loop header; the statement with that label constitutes the end of the loop (see Figure 1). In contrast, "new-style" do-loops consist of matched DO/END DO pairs, which are generally preferred (see Figure 2).

```
    ....                    ....
    DO 100 I=1,30           DO 100 I=1,30
    V(I)=0             100  V(I)=0
100 CONTINUE                ....
    ....                    ....
```

Figure 1: Old-Style Fortran Do Loops

Replace Old-Style Do-Loops was implemented as a resource refactoring in Photran as follows:
Preconditions: The source code must have at least one do-statement. The terminating statement label for each old-style do-loop must be unique. The terminating statement must be at the same level of the nesting as the do-statement. For example, the terminating statement cannot be inside an if-construct in the loop.

```
....                          ....
DO I=1,30                     DO I=1,30
   V(I)=0                        100  V(I)=0
   100 CONTINUE               END DO
END DO                        ....
```

Figure 2: New-Style Fortran Do Loops

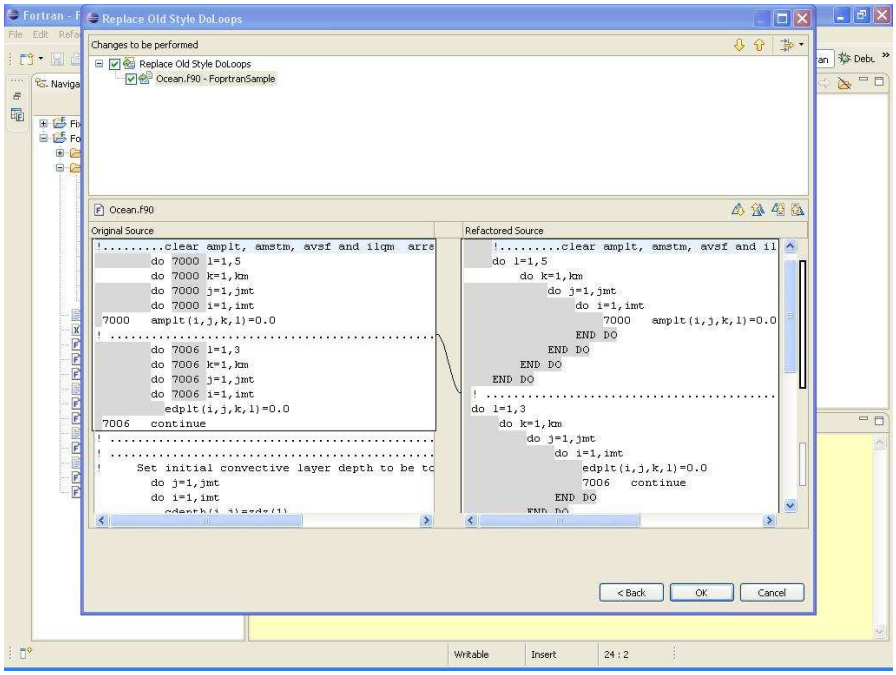

Figure 3: Photran diff view of Replace old-style Do-Loop refactoring.

Transformation: This refactoring transforms all old-style do-loops in the selected files. An END DO statement is inserted immediately following the terminating statement for each old-style do-loop. The statement label is removed from the loop header, and the loop body is re-indented. Figure 3 shows the diff-like preview of an old-style do loop refactoring as implemented in Photran.

The most difficult part of implementing a new refactoring is designing a correct set of preconditions. We believe that Replace Old-Style Do-Loops is a straightforward, syntactic transformation, whereas many other refactorings require much more complicated analyses.

5 Related Work

The concept of code restructuring has existed for many years now, and some transformation tools have being built to apply transformation rules on a complete program in batch mode. An example of this kind of infrastructure is the DMS tool, which allows for reengineering and migration of programs in many different programming languages [4].

In the case of Fortran, the vast amount of existent lines of Fortran code and the investment made on them has encouraged the development of some tools to upgrade legacy Fortran code to new standards. Greenough and Worth have reported a number of software tools currently available that may apply transformations on Fortran programs [9]. There are at least two important reasons of why these tools have not been successful. First, applying some transformation rules in batch mode may help updating the code by replacing outdated constructs (e.g., replacing obsolete operators), but that does not necessarily imply that a developer will gain a better understanding of the structure of code, nor will she be able to clean it, modularize it or remove duplication. That is, legacy code will still be legacy even if it is written in Java but with poor development practices. Second, these transformation tools are not integrated with development environments.

The concept of refactoring as an interactive process performed by an expert programmer while carefully examining the code, in small and safe steps, was defined in Opdyke's thesis many years ago [12]. Since that time, Ralph Johnson's research group at the University of Illinois has promoted refactoring and the development of automated refactoring tools, although it was not until the advent of agile methodologies that refactoring received widespread attention. Specifically for Fortran, Vaishali De's master's thesis [6] enumerates a set of possible Fortran 90 refactorings. Later on, Overbey et al. [13] bring to light the need of refactoring tools integrated with IDEs for Fortran programs and in the High Performance world. Photran is introduced as an integrated development environment that provides the necessary infrastructure for implementing Fortran refactoring [2].In a subsequent work [14], a study founded on the Fortran evolution enumerates outdated language constructs that a refactoring tool could help remove from Fortran code and proposes, more generally, a role that refactoring tools could play in language evolution. As an example, Photran was used to eliminate global variables. Tinetti et al. [17] base their work improving Fortran legacy source for performance optimization on a weather climate model implemented about two decades ago. This work is close to some refactorings proposed in this paper.

6 Conclusions and Future Work

There are some automatic tools for upgrading or migrating Fortran programs, but they have not been successful in removing legacy features of code. We believe that refactoring tools can have a profound impact in this respect. For this reason, we are working on both: the definition of a catalog of Fortran refactorings, classified with the intention of guiding developers to use the right refactoring for their needs, and on the construction of a powerful tool for development and refactoring.

Future work includes implementing more refactorings on Photran and implying it on some case studies to measure the overall improvement. Another important factor is to encourage the scientific world to use Photran, and that will require not only successful stories of the use of Photran in large applications but also providing a formal foundation that ensures behavior preservation.

References

[1] Cray Inc. http://www.cray.com/.

[2] Photran, an Integrated Development Environment and Refactoring Tool for Fortran. http://www.eclipse.org/photran/.

[3] J. Backus. The History of Fortran I, II, and III. *ACM SIGPLAN Notices*, 13(8):165–180, 1978.

[4] I. Baxter, P. Pidgeon, and M. Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the International Conference on Software Engineering, IEEE Press*, 2004.

[5] M. Cohen. Fortran: A few historical details. http://www.nag.co.uk/nagware/np/doc/fhistory.asp, Oct. 2004.

[6] V. De. A Foundation for Refactoring Fortran 90 in Eclipse. Master's thesis, University of Illinois, 2004.

[7] The Eclipse Foundation. Eclipse.org home. http://www.eclipse.org/, 2010.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[9] C. Greenough and D. Worth. The Transformation of Legacy Software: Some Tools and a Process. Technical report, RAL Technical Report TR-2003 012, 2004.

[10] JetBrains. IntelliJ IDEA 9. http://www.jetbrains.com/idea/, 2010.

[11] DB Loveman. High Performance Fortran. *IEEE [see also IEEE Concurrency] Parallel & Distributed Technology: Systems & Applications*, 1(1):25–42, 1993.

[12] W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, 1992.

[13] J. L. Overbey, S. Xanthos, R. Johnson, and B. Foote. Refactorings for Fortran and High-Performance Computing. In *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 37–39, New York, NY, USA, 2005. ACM.

[14] J.L. Overbey, S. Negara, and R.E. Johnson. Refactoring and the Evolution of Fortran. In *2nd International Workshop on Software Engineering for Computational Science and Engineering (SECSE'09)*, 2009.

[15] M. Rieger, B. Van Rompaey, B. Du Bois, K. Meijfroidt, and P. Olievier. Refactoring for Performance: an Experience Report. In *Proc. of the Third Intern. ERCIM Symposium on Software Evolution, co-located with ICSM07*, pages 206–214, 2007.

[16] D. Roberts, J. Brant, and R. Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[17] F. G. Tinetti, P. G. Cajaraville, J. C. Labraga, M. A. López, and M. G. Olguín. Reverse Engineering Applied to Numerical Software: Climate Models (in Spanish). *X Workshop de Investigadores en Ciencias de la Computación*, pages 434–438, 2008. http://hpclinalg.webs.com/hpclinalg_en.html.

[18] F. G. Tinetti, M. A. López, and P. G. Cajaraville. Fortran Legacy Code Performance Optimization: Sequential and Parallel Processing with OpenMP. *World Congress on Computer Science and Information Engineering*, pages 471–475, 2009.