J Supercomput (2013) 64:638–659
DOI 10.1007/s11227-012-0863-x

Restructuring Fortran legacy applications for parallel computing in multiprocessors

Fernando G. Tinetti · Mariano Méndez · Armando De Giusti

Published online: 23 January 2013

Abstract As it is widely known, multi-core computers are broadly used these days, and automatic parallelization of sequential programs is still a challenge. In this context, we propose a set of code transformations to be applied automatically by a tool in order to transform sequential legacy systems into their parallel version. We implement these transformations by applying a lightweight source code analysis based on rewritable AST (Abstract Syntax Tree). Since it is not always possible to automatically parallelize the code, we also implemented some specific analyses in order to report possible changes that would allow specific parallelization. Additionally, we present some examples in which these transformations were conducted and the corresponding performance experiments.

Keywords High performance computing · Parallel computing · Legacy applications · Software restructuring · Fortran

1 Introduction

Change and complexity are deeply rooted in the essence of software [10]. As a consequence, software has to undergo, almost continuously, redesign and evolution. These

F.G. Tinetti (✉) · M. Méndez · A. De Giusti
III-LIDI, Facultad de Informática, UNLP, La Plata, Argentina
e-mail: fernando@info.unlp.edu.ar

M. Méndez
e-mail: marianomendez@gmail.com

A. De Giusti
e-mail: degiusti@lidi.info.unlp.edu.ar

F.G. Tinetti
Comisión de Inv. Científicas de la Prov., Buenos Aires, Argentina

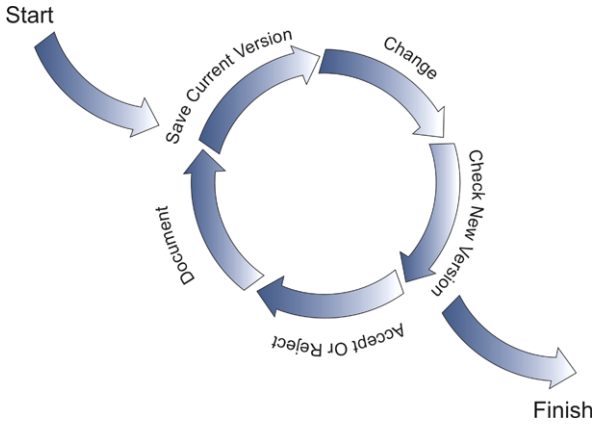

Fig. 1 Five-step process for applying changes to a legacy system

two essential software characteristics make the maintenance process an achievement in itself. This process involves different types of tasks during its instance such as enhancements, corrections, adaptations and preventions [9]. And it is also the most resource-consuming stage of the software development process. Under that point of view, there seems to be more effort being put into the building process of a brand new product than in the maintenance process of an old one. There are programming languages that have been used for more than 30, 40, or 50 years, gathering thousands of lines of source code that are still being used.

Fortran is one of the most representative long-lived programming languages [23] and, also, Fortran is being widely used by scientists these days, with a very large amount of source code being produced ever since it came into existence [13, 26]. Just as an example, in Ref. [22] six climate models have been analyzed, involving more than 1.1M SLOC (source lines of code). Fortran has its own evolutionary process that has been taking place over the last fifty years. Both, programming languages and programs, have overcome diverse stages of computer science evolution such as structured programming, object oriented programming, and so forth. Nowadays, multi/many-core processors are the new challenge that legacy systems must face, at least specifically in the field of HPC (High Performance Computing) [27]. Moreover, the process of how sequential programs are transformed into a parallel version has a preponderant impact in the maintenance stage.

In a previous work, a Legacy Change Process was presented in order to keep complexity under control in the maintenance process [22, 28]. We propose a five-step iterative and incremental process in order to manage, in a controlled way, changes to be applied in an already working program as shown in Fig. 1. This paper focuses specifically on the *Change* step, but taking into account the other steps in the whole process. Furthermore, every step is going to be carried out with a "parallelization point of view," and the *Change* step in particular is the step specifically related to source code transformation. Only changes to be performed in a sequential program in order to transform it automatically in a parallel program for shared memory parallel computers (multiprocessors) will be analyzed. These software changes are proposed as restructuring on the Fortran legacy source code.

Restructuring can be defined as a field of Software Engineering focused on improving existent source code by applying source code transformation. The origins of Restructuring rest on "the modification of software to make it easier to understand and to change, or less susceptible to error when future changes are made" [4]. It is worth mentioning that this definition excludes restructuring for any other purpose, like the improvement of the source code with the aim of performance optimization, the transformation of the code for parallel processing, and so forth. Instead, we specifically propose to include performance improvement by parallel processing via restructuring. Restructuring can be seen as a methodology that can assist in solving the significant problems that arise during the maintenance stage within the life cycle of the software development process.

Software restructuring was born as a necessary tool to be implemented in the maintenance processes because of the essential features of software so as to reduce development costs. It also can serve as a tool to introduce new software functionality. The restructuring objective mainly consists in preserving or increasing software value. Source code restructuring reduces cost of maintenance, as well as increases software re-usability and extends system's life cycle. In this way, internal software value is being increased, too [4]. Another definition of software restructuring can be found in the literature [11]: "Restructuring is the transformation from one representation to another at the same relative abstraction levels, while preserving subject system's external behavior (functionality and semantics)." A restructuring transformation is basically applied on the source code for enhancing some source code's specific feature/property; restructuring is not used to introduce new functionality.

Based on Fig. 1, we propose an iterative process, where transformations are applied by using restructuring as the main technique. We have selected the OpenMP specification [24] to parallelize Fortran legacy source code since it is directly oriented to shared memory parallel computers, which can be considered the standard computing facility since the advent of multi-core multiprocessor computers (even at the desktop). A detailed list of transformations for parallelizing sequential source code will be included. Finally, a set of open source tools are included in an IDE (Integrated Development Environment) to make this process easy to apply. One of the most remarkable aspects about this tool is automation. Nowadays, it is very difficult to find a single-tool integrating source code versioning, development environment (compiler, debugger, etc.), source code transformation automation facilities, and testing capabilities. We will concentrate on Fortran legacy code gathered in the world wide web, in order to perform and/or verify the proposed methodology.

The organization of the rest of the paper is as follows. In Sect. 2 we explain some important issues on Fortran legacy source code, its parallelization, and related work, including those implemented in well-known compilers. Section 3 includes the conceptual (i.e., independent of the implementation) ideas underlying the source code changes we propose. The contents of Sect. 4 can be seen as the detailed implementation of concepts explained in Sect. 3 (from the point of view of augmenting an existing IDE). Section 5 shows the experiments we carried out, explaining actual legacy Fortran code transformation as well as the performance obtained by running the restructured code in multiprocessors hardware. Finally, the conclusions and further work are given in Sect. 6.

2 Fortran legacy applications

Numerical/Scientific/HPC legacy applications are programmed mostly in Fortran for several reasons. Fortran has been and still is one of the most appropriate languages for numerical processing, mostly because numerical processing was about the only one application field by the time Fortran was created [5, 6]. Fortran is one of the first high level programming languages, it is in use and it has been reviewed/updated over the last decades [20, 23], unlike most of the current programming languages, including the most popular ones. Fortran has been the first standardized language and, also, it has several standards reflecting its evolution [1–3, 17–19]. Legacy software/applications and the environments in which they are used have several features that make it difficult to change and/or update them:

- Either the software documentation is outdated, or lost, or does not exist at all.
- Current standard software development methodologies and/or tools have not been used to reach the current version of the software or even the initial version.
- The current version is in fact the result of several not always well documented maintenance/adaptation changes.
- Several developers have been involved, some of them at the initial stages of the development process and others as time and environment progressed. Also, each developer used his/her own coding style.

Numerical applications have several drawbacks which are combined with the previous ones on legacy code:

- Physical/mathematical models are usually coded in large and hard to read programs, due in part to the combination of the low-level programming language abstractions and numerical method/s properties. Most of the numerical problems/properties involved are, in turn, a combination of discrete number representation and numerical method/s used to compute a solution [20].
- The software developers usually have not been trained in software development tools/processes. Thus, the initial software version contains structures and/or coding specifically oriented towards using a specific computer or computing facility instead of solving a numerical problem. Also, hardware dependent code sections are undocumented and difficult to identify in the whole application.

2.1 From sequential to parallel

For transforming sequential programs into a parallel version we must focus on some central aspects of the process:

- What part of the (sequential) code should be changed?
- What changes should be applied?
- How to apply changes?

In order to obtain answers for these questions, we propose two tasks, and both of them are focused on those portions of the source code in which most of the processing time is spent. Our first proposed task entails using a profiler to collect data to find out how the runtime is spent by the program and its function calls. The profile

information can be very useful regarding runtime and at the same time it shows those functions in which transformation could be applied in order to focus the effort and attention on parallelizing the source code. In some way, we are combining the standard HPC approach, which is in some way "profiling dependent" with the restructuring approach of changing portions of code in order to enhance some aspect of the system (specifically performance, in our context). Our second proposed task is a composite approximation to a process for legacy software. It is composed of a profiling stage in which the best candidate functions to be parallelized are gathered. After this initial stage, a well defined process is performed in order to manage source code transformation as explained above.

2.2 Related work

Since many years ago, there has been much work on legacy code as well as on applications parallelization. Some new approaches are directly designed for grid environments [7, 14] with different implementation and performance strategies, and most of them use wrappers as a way to hide and maintain integrity/correctness constraints, such as in Ref. [30] for information systems (with examples involving COBOL and INFORMIX). The work in Ref. [16] proposes or, at least, suggests a similar approach for updating Legacy Fortran code to Fortran 90/95. While our approach is similar to that in Ref. [15] (based on a coordination language), since we want to produce parallelized code from legacy code, we have several important differences:

- Our transformation is based on rewriting program AST; we do not transform the code in processes + communications.
- We do not add a library or new subroutines/processes for handling processes/threads. At least at this stage, parallelization is defined in terms of the well-known and widely used OpenMP specification.
- We do not use Fortran + C language, our approach handles and generates only Fortran code.
- Our approach is considered as part of a complete process for handling legacy code as suggested in Fig. 1 and, as such, our first implementation is already part of an integrated development environment.

Much of the previous work on parallelization has been included in current compilers, mostly as "auto-parallel" or "automatic parallelization" options according to specific compilers terminology. We have experimented with three well-known specific compilers: the PGI (The Portland Group), Intel, and GNU compilers [35–37]. While being successful in parallelizing code, none of the compilers produce/generates new Fortran code, just binaries including some processing in threads. The PGI compiler produces some interesting reports not only on parallelized Fortran `Do` loops but also on problems by which some loops are not possible to be distributed in threads. The Intel compiler generates similar parallelized code, but reports only those `Do` loops actually being parallelized. The GNU approach includes the gfortran front-end, plus some gcc parallelization options (available for/in gfortran), plus, eventually, Graphite, as explained in [35]. The GNU approach is the only one that requires an explicit number of threads to be specified at compile time. None of the compilers generates parallel source code, which is one of our main goals; compilers generate only binary parallel code.

```
....                    ....                !$OMP PARALLEL DO PRIVATE (I)
....                     ....               !$OMP&SHARED(FX,FY,FZ)
   DO 10 I = 1, N        DO  I = 1, N        DO  I = 1, N
   FX(I) = 0.0              FX(I) = 0.0         FX(I) = 0.0
   FY(I) = 0.0              FY(I) = 0.0         FY(I) = 0.0
   FZ(I) = 0.0              FZ(I) = 0.0         FZ(I) = 0.0
10 CONTINUE               END DO              END DO
....                    ....                !$OMP END PARALLEL DO

  Version I               Version II             Version III
```

Fig. 2 Three versions of a Do loop

3 Legacy source code parallelization *for* multiprocessors

In this section we describe a set of source code transformations in order to parallelize sequential Fortran source code with the so-called OpenMP work-sharing directives. Even though this process could be completely performed by hand, it would be highly time consuming and error prone. This set of source code transformations are focused mainly on Fortran `Do` loop statements and are intended to be applied automatically by a tool. In previous work, we developed and performed a number of automated source code transformations in order to upgrade old Fortran source code features or coding practices found in legacy applications, as a previous step to parallelization [28]. Specifically focused in `Do` loops, we expect to have three versions of code, as shown in Fig. 2, where the Version I is the legacy code, almost always in Fortran fixed source form, Version II would be an *actual* Fortran 90/95 code, and Version III is expected to be the parallel version using OpenMP according to the work presented in this paper. Versions II and III would correspond to two full cycles of those shown above in Fig. 1. The most important task to be carried out, however, is finding out the conditions under which an OpenMP directive can be used successfully in terms of:

- Numerical Results: the resulting program should produce the *same* result as the original one. It should be taken into account that when dealing specifically with numerical programs it is likely that some difference could be produced in results due to floating point representation (e.g. different rounding on different order of operations).
- Performance: the resulting program performance should be better than the original one. It is worth noting that including OpenMP directives by itself does not guarantee performance improvements, and the resulting parallelized code has to be analyzed from the point of view of performance.

The rest of the section is devoted to describe specific requirements that `Do` loops should fulfill in order to be considered eligible for OpenMP parallelization, as well as the characteristics of data accesses that have to be taken into account.

3.1 Do loops selectable for parallelization

We start from very restrictive conditions under which `Do` loops are taken into account for parallel analysis, depending on the statements enclosed in the loop (referred to as *do-block* in [18]):

```
      DO 130 I = 2, M
      Z(I)   =  Z(I-1) - Z(I)
      FZ(I)  =  ( ZB(I) - Z(I) )  /  ( Z(I-1) - Z(I) )
 130 CONTINUE
```

Fig. 3 Do loop with data dependencies

```
     DO 130 I = M/2, M
     Z(I)   =  Z(I-M/2) - Z(I)
     FZ(I)  =  ( ZB(I) - Z(I) )  /  ( Z(I-M/2) - Z(I) )
130 CONTINUE
```

Fig. 4 Discarded Do loop without data dependencies (Case I)

```
     DO 130  I = 1,M1
       DBZ(I) =  Z(I+1) - Z(I)
       FZ(I)  =  ( ZB(I) - Z(I) )  /  ( Z(I+1) - Z(I) )
130 CONTINUE
```

Fig. 5 Discarded Do loop without data dependencies (Case II)

– Assignment statements, which of course may involve complex expressions on the right-hand side of the "=".
– Nested `Do` loops, which in turn may include other `Do` loops and/or assignments.

Thus, we have excluded `Do` loops difficult to parallelize or for which the parallelization analysis could become too complex. Furthermore, we have "labeled" `Do` loop statements as not parallelizable if they possess *data dependencies*. Data dependencies are identified using a simple syntactic parsing rule: array accesses have to be made only using the loop control variable (do variable in terms of the Fortran standard terminology, e.g. [18]) as index, and the loop control variable should not appear in arithmetic expressions with other scalar or values used as index(es) in array access(es). Figure 3 shows array accesses preventing `Do` loop parallelization where, in fact, there are data dependencies in the `Do` loop. More specifically, the assignment to `Z(I)` is identified as having a data dependency since array variable `Z` appears to the left and to the right of the assignment and the loop control variable is used at the right side in an expression for obtaining the index at which the same variable `Z` is accessed. This way of identifying data dependencies is over-restrictive, since it provides several "false positives" (or, in fact, false negatives for parallelization, since it leads to *discard* `Do` loop parallelization analysis) as that in Fig. 4, where the assignment to `Z(I)` does not include any data dependency. Furthermore, this over-restrictive parsing rule leads to even more *undesirable* false positives, such as that in Fig. 5. Even when this parsing rule could be considered too restrictive (i.e. discarding `Do` loop parallelization when the index expression for accessing arrays is other than a single scalar variable), we are still interested in the most simple cases for parallelization as a *proof of concept*, in terms of:

– Syntactic analysis: simple cases are easy to identify syntactically, using (statically) the program AST representation, and avoiding complex analysis. One of the complex cases this rule excludes is that of *aliased* variables using the so-called

```
....              ....              ....
DO I =1,30        DO I=1,30         DO I=1,30
  M(I,J)=0          DO J=1,30         DO J=1,30
END DO                M(I,J)=0          DO K=1,30
....                END DO                M(I,J,K)=0
....              END DO                END DO
....              ....                  END DO
....              ....              END DO
....              ....              ....
```

Fig. 6 Simple parallelization loops

`Equivalence` statement, where two variables share storage units as defined in the standard, e.g. [18].
- AST editing: we expect that simple cases are also simple for automatic code/AST transformation, so we can implement the necessary transformations in a software tool (expected to be included into some IDE).
- Preliminary experimentation for performance gain: since performance is the real objective of parallel processing, we expect to obtain good performance results in order to move forward to more complex cases for analysis and parallelization.

The first two items above let us to exclude more complex and time-consuming analysis, such as that of classical data dependency analysis [8, 12, 21]. The next subsections will show that even in the most simple `Do` loops it is possible to find different data accesses/usage that have to be taken into account for parallel processing with OpenMP.

3.2 Do loops with only shared and loop control variables

The most simple (at least from the parallelization point of view) `Do` loop statements have only shared variables (arrays) and loop control variable(s). Since the loop is considered as parallelizable, each assignment statement involves only arrays indexed with loop control variables, as shown in Fig. 6 for three parallelizable `Do` loops. In each case, (a) the most external `Do` loop is parallelizable, (b) array `M` is shared and (c) loop control variables (either `I`, or `I` and `J`, or `I`, `J`, and `K`, respectively) should be made private to each thread in a parallel version with OpenMP. Figure 7 shows the third case as expected to be parallelized automatically, where every variable has been explicitly included in either one of the PARALLEL DO clauses: PRIVATE or SHARED. We have chosen to include explicitly every variable referenced in the `Do` loop in one clause for several reasons:

- Readability: every data is referenced at the beginning of the parallelized `Do` loop with its corresponding so-called data-sharing attribute [24]. Even when OpenMP has a set of rules for variables referenced in a parallel construct and a parallel region (both of which—regions—are defined in the OpenMP specification), we do not force the programmer to remember every definition and specification.
- Further parallelization analysis and performance tuning: parallelization always requires to identify clearly every data access either for shared or distributed memory

Fig. 7 Simple loop parallelized

```
!$OMP PARALLEL DO PRIVATE (I, J, K)
!$OMP&SHARED(M)
DO I=1,30
  DO J=1,30
    DO K=1,30
      M(I,J,K)=0
    END DO
  END DO
END DO
!$OMP END PARALLEL DO
```

```
DO I =1, N
  sometmp = <expr>
  M(I)    = sometmp
END DO

(a)
```

```
!$OMP PARALLEL DO PRIVATE (I, sometmp, ...)
!$OMP&SHARED(M, N, ...)
DO I =1, N
  sometmp = <expr>
  M(I)    = sometmp
END DO
!$OMP END PARALLEL DO
 (b)
```

Fig. 8 Loop with private variable other than the loop control variable

parallel computers. If some further parallelization (or optimization) analysis is required, having every data explicitly defined as private or shared will save much of the time and work required.

Obviously, these simple `Do` loops are not the only ones found in legacy code, and the next subsections will describe some other parallelizable loops.

3.3 Do loops with only shared and private variables

Taking into account the previous `Do` loops characteristics, the next level of complexity in the analysis of `Do` loop parallelization is found when there are variables used in the *do-block* which are necessary mainly for temporary calculations. More specifically, some `Do` loops include variables which are assigned with completely new values in every `Do` iteration, such as that shown in Fig. 8a. Even when variable `sometmp` is as private as the loop control variable `I`, the analysis required for its identification is different (either automatically or by a programmer): private variables are those scalars or array variables accessed using a constant (inside the loop) along every iteration of the `Do` loop. Clearly, the parallel version requires that `sometmp` is used as private in each OpenMP thread, as explicitly defined in the code shown in Fig. 8b.

3.4 Do loops with reduction variables

There are some `Do` loops in which certain forms of recurrence calculations are performed, in these cases the REDUCTION clause is needed, in order to prevent unpredictable results (runtime race conditions). The reduction process is described in [24] as follows, “A private copy of each list item is created, one for each implicit task, as

```
DO 1000 I=1,MOLSA
      VELOCX=H(1,1)*X(1,I,1)+H(1,2)*X(2,I,1)+H(1,3)*X(3,I,1)
      VELOCY=H(2,1)*X(1,I,1)+H(2,2)*X(2,I,1)+H(2,3)*X(3,I,1)
      VELOCZ=H(3,1)*X(1,I,1)+H(3,2)*X(2,I,1)+H(3,3)*X(3,I,1)
      SUMPX=SUMPX+VELOCX
      SUMPY=SUMPY+VELOCY
      SUMPZ=SUMPZ+VELOCZ
1000  CONTINUE
```

Fig. 9 Parallelizable loop with private, shared, and reduction variables

```
!$OMP PARALLEL DO
!$OMP&SHARED(H,X,MOLSA)
!$OMP&PRIVATE(I,VELOCX,VELOCY,VELOCZ)
!$OMP&REDUCTION(+:SUMPX,SUMPY,SUMPZ)
DO I=1,MOLSA
      VELOCX=H(1,1)*X(1,I,1)+H(1,2)*X(2,I,1)+H(1,3)*X(3,I,1)
      VELOCY=H(2,1)*X(1,I,1)+H(2,2)*X(2,I,1)+H(2,3)*X(3,I,1)
      VELOCZ=H(3,1)*X(1,I,1)+H(3,2)*X(2,I,1)+H(3,3)*X(3,I,1)
      SUMPX=SUMPX+VELOCX
      SUMPY=SUMPY+VELOCY
      SUMPZ=SUMPZ+VELOCZ
END DO
!$OMP END PARALLEL DO
```

Fig. 10 Parallelized loop with private, shared, and reduction variables

if the private clause had been used. The private copy is then initialized to the initialization value for the operator, as specified above. At the end of the region for which the reduction clause was specified, the original list item is updated by combining its original value with the final value of each of the private copies, using the operator specified." This kind of Do loop statement is one of most difficult to transform, either by hand or automatically, since we need to identify shared variables, private variables, and reduction variables. Figure 9 shows an actual legacy code which is parallelizable taking into account shared, private, and reduction variables. The corresponding parallel code (using OpenMp) for the Do loop of Fig. 9 is shown in Fig. 10, where every variable in the Do loop is explicitly identified (included in the corresponding clause) as private, shared, or reduction variable. Note that having every variable included in the (PARALLEL DO) directive data clauses provides a single place for analyzing the amount of data (or, at least, variables) accessed in the Do loop. This information could be used in further parallelization analysis, since it is related to the amount of data necessary to be distributed in a distributed memory parallel computer, for example.

4 A restructuring tool for parallelization—a *proof of concept*

We have implemented a software tool so that all of the transformations described in the previous section are applied automatically as restructuring to legacy software. In

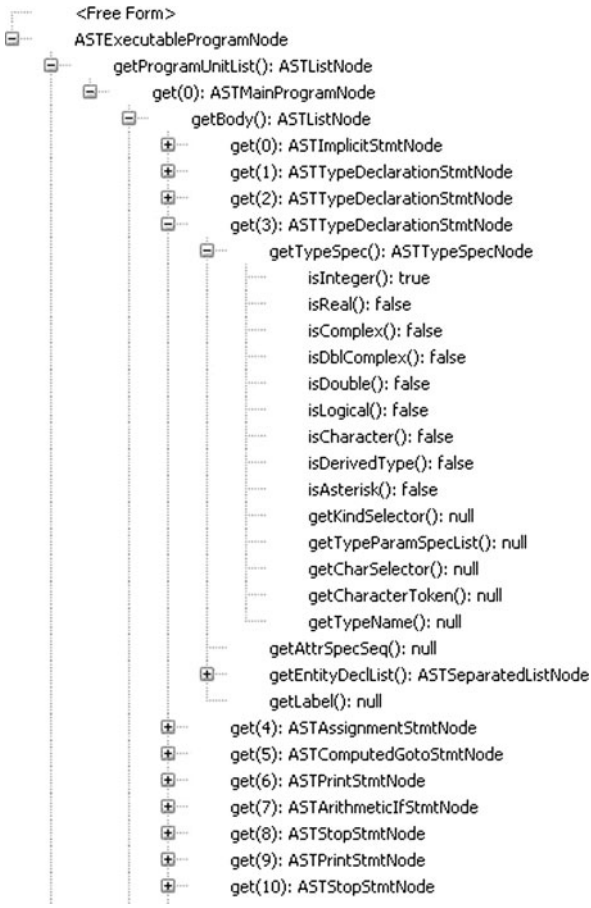

Fig. 11 An example of Photran AST

this context, *automatic* means *made by a tool*, not mandatory or compulsory, i.e., the user/programmer may select a `Do` loop and the restructuring tool either applies the restructuring if the loop is parallelizable or reports it is not possible to parallelize the loop. The tool is aware of index dependencies, identifies local, shared, and reduction variables and generates the source code including OpenMP directives. We base our implementation on AST (Abstract Syntactic Tree) analysis and modification. The current implementation was made as a proof of concept using Photran, an Eclipse plug-in based on the Eclipse CDT (C/C++ Development Tooling) [33, 34]. We are also planning to provide the tool as open source to be easily included in Photran. However, the concept of restructuring for parallelization can be implemented stand-alone, it is not dependent on Photran in particular.

Photran implements Fortran restructuring as Eclipse refactoring, and contains two very important data structures for refactoring implementation. The first one is the AST, which maintains the entire representation of a Fortran program, as in Fig. 11 (a partial example). The AST structure is filled with AST nodes, a set of classes that represent each possible element of the programming language. The second important structure is the VPG (Virtual Program Graph) which facilitates the handling of the

AST and the embedded analysis information, acting as an interface for the AST to the programmer [25]. Thus, the VPG allows refactoring programmers to acquire or release ASTs; it also sets off scope and binding analysis; while allowing the user to obtain variable definitions, and so forth.

Photran divides refactorings into two categories (selectable by the Photran developer): *editor-based refactorings*, which require the user to select part of a Fortran program in a text editor in order to initiate the source code transformation, and *resource refactorings* which apply to entire files. Once selected the kind of code transformation to create, the developer implements the corresponding subclass and configures Photran (via XML). The developer then should add

- The necessary check of initial preconditions, e.g., to verify that the user selected the correct construct in the editor, that the file is not read-only, and so forth.
- User input handling in case it is needed, e.g. for adding a parameter to a subprogram, the user should supply the new parameter's name and type. Some extra preconditions may be added depending on user input, for the transformation to be performed, and to ensure the resulting code will compile, and it is expected to maintain the behavior of the original program.
- The code transformation, which determines what files will be changed, and how.

The XML configuration file and Java reflection facilities are used so that 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. This dialog includes a *diff*-like preview, which allows the user to see what changes will be made before committing it. This preview can be used as a preliminary aid for the *Check New Version* and *Accept or Reject* steps of the whole legacy code transformation process.

The Photran engine provides two classes corresponding to the two source code transformation types (*editor-based* or *resource*). A new code transformation is implemented by extending the FortranEditorRefactoring or FortranResourceRefactoring (see Fig. 12). There are some methods that must be implemented in order to obtain a specific refactoring. Each one of these methods has a precise intent within the Photran structure:

```
public String getName()
protected void doCheckInitialConditions(RefactoringStatus
   status, IProgressMonitor pm) throws PreconditionFailure
protected void doCreateChange(IProgressMonitor pm) throws
   CoreException, OperationCanceledException
protected void doCheckFinalConditions(RefactoringStatus
   status, IProgressMonitor pm) throws PreconditionFailure
```

From the point of view of implementation, code restructuring is carried out in two basic steps: parallelization preconditions are checked in the first step, and the actual code change is made in the second step. The parallelization preconditions checked in the first step include:

1. *Loop selection*: only `Do` loop statements are allowed to be transformed in this transformation. Also, `Do` loop is allowed to include only other `Do` loop statements and assignments in the *do-block*. `Do` loop statements must be ended with the END

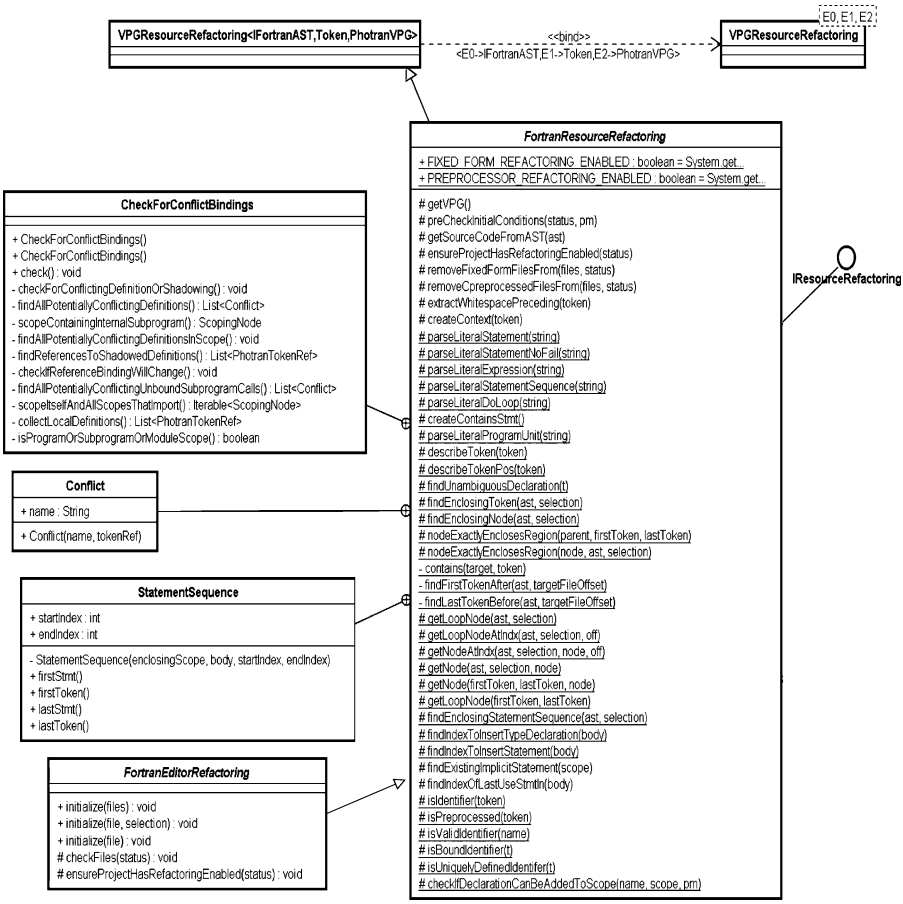

Fig. 12 Photran refactorings class diagram

```
public class StatementsAllowedVisitor extends ASTVisitorWithLoops{
    private boolean allStatemensAllowed = true;
    public                  StatementsAllowedVisitor(){...}
    public boolean          allStatementsAllowed(){...}
    @Override public void   visitIExecutionPartConstruct(... node){...}
    private boolean         allowedStatements(... node){...}
    @Override public void   visitIExecutableConstruct(... node){...}
    @Override public void   visitIActionStmt(... node) {...}
    @Override public void   visitIObsoleteActionStmt(... node){...}
}
```

Fig. 13 StatementsAllowedVisitor Class

DO statement. In order to detect allowed statements inside Do loop as shown in Fig. 13: the Do loop node is traversed in order to check if the statements inside the loop are valid Fortran statements.

2. *Data dependencies*: arithmetic expressions are not allowed as array indexes, as explained above. If some data dependency is found, the loop is considered as non-parallelizable and a report is issued to the programmer (no restructuring/change is made to the source code). Data dependencies where found by using another visitor pattern called DependencyFinderVisitor. If the index variable used in the loop was found inside any binary expression i.e.: `Op1 {+,-,*,/} Op2`, the visitor will assume that dependencies are present inside the source code.

After these conditions are checked and passed, the transformation/code restructuring is carried out as follows:

1. *Get first and last* `Do` *loop statement tokens*: the first and last tokens are gathered in order to get the places where the OpenMp directives will be placed, as well as the *do-block* scope (i.e. the specific statements included in the *do-block*).
2. *Get all variables*: all variables referenced in the `Do` loop statement are collected by visiting the AST `Do` loop node. It is worth noting that the AST representation of programs provides full information on variables as well as all the information needed for compilation.
3. *Get possible reduction assignments*: in order to recognize reduction variables assignments like:

   ```
   <variable> = <variable> op <exp>
   ```

   or

   ```
   <variable> = <exp> op <variable>
   ```

   are identified, if any.
4. *Get private variables*: all scalar variables assigned in the `Do` loop and not identified as a reduction variable are considered as (OpenMP) *private*. Scalar variables inside a `Do` loop are mostly used for containing auxiliary/temporary results and, as such, are private/*local* to each (OpenMP) thread.
5. *Get shared variables*: array variables and read only variables in the `Do` loop are considered as (OpenMP) *shared*. Clearly, the best scenario for parallel computing is that which has most of the variables as shared, but we have to be careful (and conservative, as we have been in the definition of the above rules) in order to avoid race conditions on data assignments, specifically.
6. *Get reduction variables*: all the variables in the possible reduction assignments identified in Step 3 above which are initialized to the proper values depending on the reduction operation involved [24] are considered as OpenMP *reduction variables*.
7. *Build OpenMP directives*: the OpenMP directives are built according to the previous analyses on the `Do` loop. So far, we have been working on `Do` loop statements and, thus, a so-called Combined Parallel Work-Sharing Construct, more specifically a PARALLEL DO is built.
8. *Rewrite AST node*: the `Do` loop node is rewritten with the OpenMp directive and the corresponding data clauses (PRIVATE, SHARED, etc.) built in the previous step. Taking into account the parallelization analysis (applied only to `Do` loops) and the combined PARALLEL DO we build, only a *relatively few* source code lines related

```
    private void makechange(ASTProperLoopConstructNode node){
        final Map<String,OpenMpWraper> Vars
              = new HashMap<String,OpenMpWraper>();
        Token firstToken=node.findFirstToken();
        Token lastToken=node.findLastToken();
        getOpenMpVariables(node,Vars);
        final ArrayList<aSTAssignmentStmtNode> assignements
              = new ArrayList<aSTAssignmentStmtNode>();
        collectPosibleReductionAssignements(assignements);
        getReductionVariables(Vars, assignements);
        //obtain shared variables
        String sharedVarList =makeSharedVariableList(Vars);
        //obtain private variables
        String privateVarList =makePrivateVariableList(Vars);
        //obtain reduction variable
        String ReducionVarList=makeReductionVariableList(Vars);
        //rewriting AST
        firstToken.setWhiteBefore("\n!$OMP PARALLEL DO " +
                                  "\n"+sharedVarList+
                                  " " +privateVarList+
                                  " "+ ReducionVarList + "
\n");
        lastToken.setWhiteAfter("\n
!$OMP END PARALLEL DO\n");
    }
```

Fig. 14 The source code transformation performed in the makechange() method

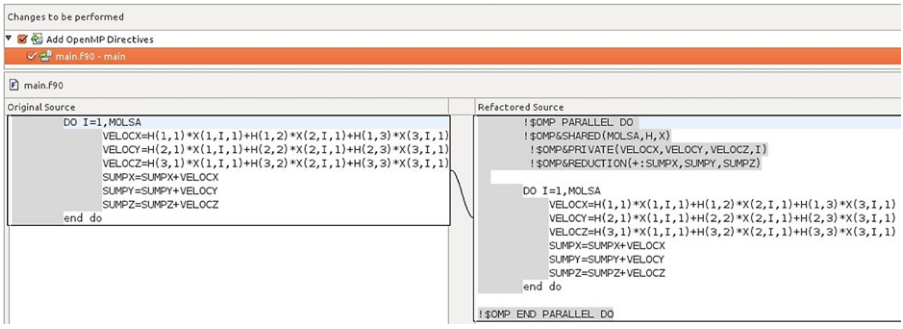

Fig. 15 Example of Do loop parallelization with OpenMP

to Do loops have to be included in the AST, as shown in the makechange() method implementation in Fig. 14.

The new code is ready to be compiled with the corresponding OpenMp compiler option. Figure 15 shows an example of restructuring as it is currently implemented in Photran. On the left of Fig. 15 is the sequential code, in which a Do loop has been selected for parallelization. On the right of Fig. 15 the new code (i.e. the code gener-

ated with OpenMP directives by the tool) is shown, which is considered as the current one after it is explicitly accepted by the user. Highlighting source code differences as shown in Fig. 15 has many advantages, such as: (a) it provides the programmer full knowledge of the source code transformation/s, (b) it can be considered as *part of* or *aid to* the *Check New Version* and *Accept or Reject* source legacy transformation process steps, (c) it enhances source code understanding to the programmer, which is also useful/necessary for the *Document* step of the process.

5 Experiments and results

As part of the proof of concept, it is necessary not only to use and verify the implemented tool, but also to analyze the performance results obtained in real legacy code. We have selected three applications available to download from Internet, each one belonging to a different application area. The first program is used in the electromagnetic scattering by particles and surfaces area, downloaded from [29], and it will be referred to as "Refl Code." The second program belongs to the molecular dynamics area, computing a Lennard–Jones potential, downloaded from [31], which will be referred to as "L–J Code." The third program computes a cumulative Maxwell–Boltzmann distribution function by integrating with the trapezoidal rule, downloaded from [32] and this program will be referred to as "M–B Code." We made several non-parallelization restructuring on this M–B Code, mostly related to data precision, prior to the restructuring we describe below, which is specifically focused on including OpenMP directives and parallelizing the code.

As explained in Sect. 2.1, not all of the source code of each program is selected to change/restructure, and profiling is used to identify the function on which OpenMP directives are going to be included. This decision has two main advantages:

- HPC/scientific programmers' approach to performance optimization almost always has involved profiling and, thus, profiling is considered as part of the optimization/parallelization process. We can take advantage of the usual programmers' knowledge of their own programs and profiling tools.
- Profiling usually provides a reduced set of functions/subprograms on which to concentrate all the effort from two points of view:
  - Parallelization details/complexity is reduced when only a specific function at a time is approached. Also, the set of subprograms usually selected by profiling is small as compared with the total number of subprograms in the application.
  - Programmers' control is considered fundamental for understanding and analysing performance, and these tasks are always better accomplished on a reduced set of functions/subprograms.

We have selected two hardware platforms on which to carry out the performance experiments: a very small-scale parallel environment, and a medium sized shared memory parallel environment. The very small parallel platform (e.g. a desktop computer) has been selected because not all legacy software is focused on large-scale problems and, also, scientists could run scaled down models for preliminary analysis. The medium sized shared memory parallel environment is usually found at the

Table 1 Parallel platforms used in the experiments

	Comp1	Comp2
Processor	Intel i5 M460	2 × Intel Xeon E5405
Total # Cores	2	8
RAM	4 GB	2 GB
OS	Linux 2.6.38 i686	Linux 2.6.31 ×86_64
Compiler	gfortran 4.5.2	gfortran 4.4.2
Compiler options	-O3 -fopenmp	-O3 -fopenmp

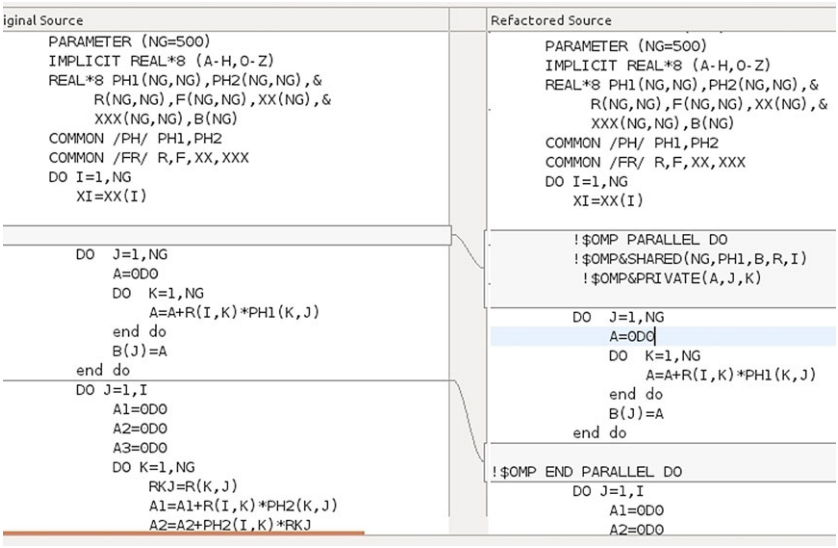

Fig. 16 Automated code transformation for parallelization on Refl Code

basis of clusters used for medium- to high-scale parallel computation. Table 1 shows the main characteristics of the two parallel hardware platforms, referred to as Comp1 for the small-scale and Comp2 for the medium-scale platform, respectively. We have not made specific efforts for obtaining homogeneous environments (same compiler, Linux kernel, etc.) since we do not intend to impose specific environments as constraints to our approach, i.e. we use the computers as they are currently available to scientists. The following sections show the specific results of including OpenMP directives and the performance gains obtained in each of the programs described above.

5.1 Refl Code

Profiling on the Refl Code highlighted that one specific subprogram takes about 77 % of the runtime, thus making this function the clear objective for parallelization. On this subprogram, several `Do` loops were selected for parallelization candidates and, also, we developed our own handmade source code with OpenMP directives, for comparison purposes. Figure 16 shows the suggestion made by the tool on a specific `Do`

Table 2 Refl Code parallel performance—speedup

Comp1	Comp2		
2 Cores	2 Cores	4 Cores	8 Cores
1.8	1.89	3.40	5.40

Fig. 17 Example of code not parallelizable by the tool

```
do j=i+1,n
   xx=x(j)-x(i)
   ...
   call mic(xx,yy,zz,aLx,aLx2)
   dR=xx**2+yy**2+zz**2
   if(dR.lt.cut2) then
      r=dsqrt(dR)
      ...
      if(ir.gt.ngr) stop 'ir>ngr'
      ig(ir)=ig(ir)+1
   end if
end do
```

loop (where the original source code is at the right and the suggested change is at the left of the figure), for which the user should accept or discard the proposed change. The parallel code made by our tool is equivalent to the handmade parallel code version and, also, the source code provided by our tool has every variable accessed in the Do loop identified either as PRIVATE or SHARED data.

Table 2 shows the performance measurements in terms of speedup in the two parallel platforms described above. Parallel performance is slightly reduced taking into account the theoretical maximum speedup for each number of cores, as more cores are involved in the computation. This reduction can be explained as a combination of (a) the *usual* speedup performance reduction as more processors are used, and (b) the parallelized subroutine becoming less time-consuming and other subroutines taking a greater fraction of the total runtime. The second reason leads to a *natural* task to follow the parallelization process: profile the parallel program in order to select the next subroutine/s to parallelize. It can be considered that performance (including scalability) is an important factor for the final decision in the *Accept or Reject* step of the legacy source code transformation process, and this is why we want to specifically show the performance obtained by the new version of the code (it will be also shown in the next examples).

5.2 L–J code

The main program as well as one of the subroutines were clearly identified via profiling as the ones to be parallelized, since more than 97 % of the runtime was spent on these two sections of code. Analyzing the code in the subroutine we found one of the limitations of our own rules for parallelization: Fig. 17 shows that the code in a Do loop of the subroutine involves IF and CALL statements, thus making the (Do) subroutine not parallelizable by our tool. Thus, the OpenMP directives on the code of Fig. 17 where included by hand, but on the main program, the tool provided an

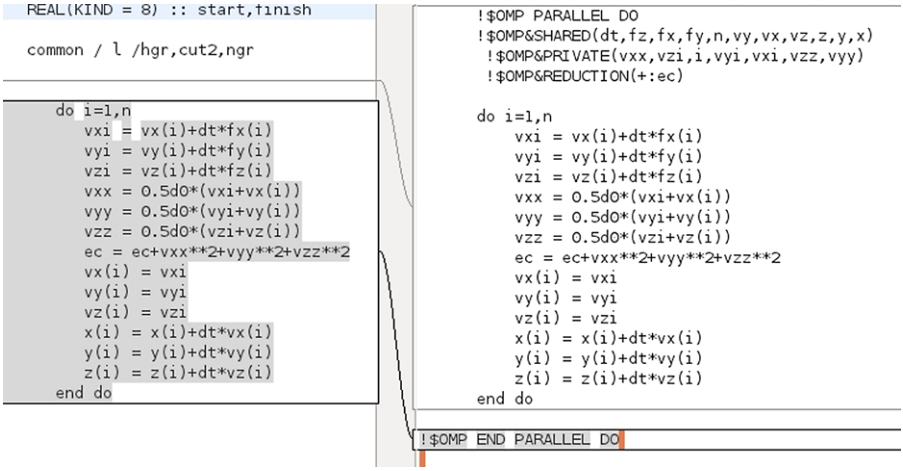

Fig. 18 Automated parallelization transformation on L–J Code

Table 3 L–J code parallel performance—speedup

Comp1	Comp2		
2 Cores	2 Cores	4 Cores	8 Cores
1.21	1.29	2.20	4.05

automated parallelization, as shown in Fig. 18. Thus, for this code, parallelization is partially made by a programmer and partially made by our tool. Even when not all the parallelization could be implemented automatically (by our tool), there was a clear advantage in using an automated approach: the programmer should focus only on those sections of code that could not be solved/approached automatically by the tool. Table 3 shows the performance measurements in terms of speedup in the two parallel platforms described above. Even when performance is not as good as in the previous case, using the tool provides too important results: (1) a parallel version of the legacy code, and (2) information on sections of code on which the programmer should concentrate the efforts for parallelization.

5.3 M–B code

The profile of this code has shown that only one subroutine requires more than 92 % of the runtime, so all the parallelization efforts have been made on this specific subroutine. However, we have found that the code in the subroutine does not fulfill the rules we have defined for automatic parallelization. Since code in this subroutine is basically the only one on which parallelization could be successful, we have implemented a little extension on our tool: information on the reason/s why the automatic parallelization could not be done is provided to the programmer. Thus, the programmer receives two important results: (a) an indication that automatic parallelization is not possible, and (b) the explicit information regarding the rule by which the automatic parallelization has not been made. Specifically the latter information

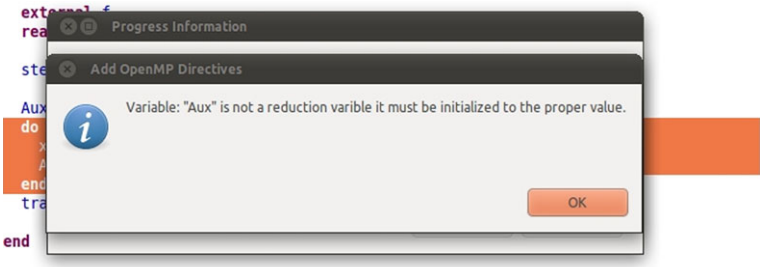

Fig. 19 Non-parallelizable Do loop and tool suggestion/report

Table 4 M–B code parallel performance—speedup

Comp1	Comp2		
2 Cores	2 Cores	4 Cores	8 Cores
1.83	1.97	3.87	7.67

is expected to be useful to the programmer in order to modify the code so that the automatic parallelization could be successfully applied.

Figure 19 shows how the tool provides the information about the reason why it is not possible to parallelize the code. More specifically, the programmer receives the report indicating that variable `Aux` is likely to be defined as an OpenMP reduction variable, but the initial value does not correspond to the specific reduction operation. Once the variable is initialized to the proper value, the parallelization is automatically made by the tool, thus resulting in a parallelization guided by our tool and explicitly made by a programmer. The resulting program is analyzed from the point of view of performance, and Table 4 shows the results in speedup in the parallel platforms described above. Performance results for this specific case are amongst the best obtained and, also, have the characteristics that every scientific programmer hopes for: good performance as well as scalability.

6 Conclusions and further work

Transforming sequential programs into parallel programs is a very arduous task in which different skills are required. There is a vast amount of old sequential (legacy) software being currently used, in most of the cases this software being critical for their owner organizations. We propose a set of automated transformations to be applied in those programs in order to update and take advantage of parallel processing in multiple processors multi-core architectures. Since it is not always possible to provide an automatic parallelization from legacy code, we have implemented several analyses in order to provide the programmer with useful suggestions for parallelization and/or to fulfill the requirements imposed by our tool to parallelize the code.

We have presented three complete examples on which our tool has been tested. As explained in the first section, we have focused the *Change* step defined for transforming legacy code, but several details are useful for other steps, specifically for the

Check New Version (or just *Check*), *Accept or Reject*, and *Document*. As expected, most of the description in this paper has been devoted to explain the *Change* step, since it provides the rationale for the other two steps under consideration. The *Check* step has been successful in all the changes, since the rules by which we decided to change the code are extremely conservative, as explained above. The *Accept or Reject* step has been guided mostly by performance and, eventually, by taking into account that the implemented changes provide a performance enhancement with minimum cost for the programmer without making the code extremely complex. In this context, every change to the code can be considered as restructuring since it affects a relatively small fraction of the source code and does not force the programmer to redesign the whole software project.

We are doing preliminary work on many extensions to the tool, taking into account some parallelization rules' relaxation. In one of the extensions, the tool suggests/implements a parallelization even in the presence of statements other than assignments and `Do` loops. More specifically, we are studying real code in which there are IF, CALL, and function calls, which involve complex analysis and checking rules. A priori, we are experimenting the results of just letting the tool to parallelize the code and issuing warning to the user about the specific "non-safe" parallelization. Thus, the programmer could run some specific testing for these "non-safe" parallel versions in order to find out if the resulting program provides incorrect results. We are also intensively working on the data analysis of the code (e.g. COMMON areas, subroutine dummy arguments, etc.), since they are amongst the most important analyses for shared memory as well as distributed memory parallelization tasks.

References

1. American National Standards Institute, X3. 9-1966 (1996) American National Standards Institute Incorporated, New York
2. American National Standards Institute, X3. 9-1978 (1978) American National Standards Institute, New York
3. American National Standards Institute (1992) American national standard for programming language, FORTRAN—extended: ANSI X3.198-1992: ISO/IEC 1539: 1991. American National Standards Institute
4. Arnold RS (1989) Software restructuring. Proc IEEE 77(4):607–617
5. Backus J (1954) The IBM 701 speedcoding system. J ACM 1(1)
6. Backus J (1978) The history of Fortran I, II, and III. ACM SIGPLAN Not 13(8)
7. Baliś B, Bubak MT, Wegiel M (2008) LGF: a flexible framework for exposing legacy codes as services. Future Gener Comput Syst 24(7)
8. Banerjee U (1997) Dependence analysis. Kluwer Academic, Dordrecht
9. Bennett KH, Rajlich VT (2000) Software maintenance and evolution: a roadmap. In: Proceedings of the conference on the future of software engineering, Limerick, Ireland. June 2000
10. Brooks FP (1987) No silver bullet: essence and accidents of software engineering. IEEE Comput 20(4):10–19
11. Chikofsky EJ, Cross JH II (1990) Reverse engineering and design recovery: a taxonomy. IEEE Softw 7(1):13–17
12. Cooper KD, Torczon L (2005) Engineering a compiler. Morgan Kaufmann, San Mateo
13. Decyk VK, Norton CD, Gardner HJ (2007) Why Fortran? Comput Sci Eng 9(4)
14. Deng Y, Wang F (2011) LAG: achieving transparent access to legacy data by leveraging grid environment. Future Gener Comput Syst 27(1)

15. Everaars CTH, Arbab F, Burger FJ (1996) Restructuring sequential Fortran code into a parallel/distributed application. In: Proc of the 1996 international conference on software maintenance. IEEE Comp Society, Los Alamitos
16. Greenough C, Worth D (2004) The transformation of legacy software: some tools and a process. RAL technical report TR-2003 012
17. ISO, ANSI/ISO/IEC 1539-1:1997 (1997) Information technology–programming languages—Fortran. Part 1. Base language. American National Standards Institute
18. ISO, ANSI/ISO/IEC 1539-1:2004 (2004) Information technology—programming languages—Fortran. Part 1. Base language. International Organization for Standardization
19. ISO, ISO/IEC JTC 1/SC 22/WG 5/N1830 (2010) International Standard ISO/IEC DIS 1539-1, Information technology—programming languages—Fortran. Part 1. Base language, 3rd edn
20. Loh E (2010) The ideal HPC programming language. Maybe it's Fortran. Or maybe it just doesn't matter. Queue 8(6)
21. Maydan DE, Amarasinghe SP, Lam MS (1993) Array-data flow analysis and its use in array privatization. In: Proceedings of the 20th ACM SIGPLAN-SIGACT symp on principles of programming languages, Charleston, South Carolina, USA, March 1993, pp 2–15. https://www.ideals.illinois.edu/handle/2142/16950
22. Méndez M, Tinetti FG (2011) First steps towards a tool for legacy systems. In: XVII congreso Argentino de ciencias de la computación, UNLP, La Plata, Argentina, Oct. 2011. Available at https://lidi.info.unlp.edu.ar/fernando/publis/082.pdf
23. Metcalf M (2011) The seven ages of Fortran. J Comput Sci Technol 11(1):1–8. http://journal.info.unlp.edu.ar/journal/journal30/papers.html
24. OpenMP Architecture Review Board (2011) OpenMP application program Interface—version 3.1. Available at http://openmp.org/wp/
25. Overbey JL, Chen N (2009) Photran 6.0 developer's guide, December
26. Sanders R, Kelly D (2008) Dealing with risk in scientific software development. Software, IEEE Press, New York 25(4)
27. Sutter H (2005) The free lunch is over: a fundamental turn toward concurrency in software. Dr Dobb's J 30(3). http://www.gotw.ca/publications/concurrency-ddj.htm
28. Tinetti FG, Méndez M, Lopez MA, Labraga JC, Cajaraville PG (2011) Update and restructure legacy code for (or before) parallel processing. In: Proceedings of the 2011 international conf on parallel and distributed processing techniques and applications, vol 1, Las Vegas, USA, July 2011. CSREA Press, Las Vegas, pp 652–658 ISBN:1-60132-193-7
29. Mishchenko MI, Zakharova NT FORTRAN codes for the computation of the bidirectional reflection function for flat particulate layers and rough surfaces. Goddard Space Flight Center Sciences and Exploration Directorate, NASA. http://www.giss.nasa.gov/staff/mmishchenko/brf/
30. Thiran P, Hainaut J, Houben G, Benslimane D (2006) Wrapper-based evolution of legacy information systems. ACM Trans Softw Eng Methodol 16(4)
31. Velasco E Prácticas de métodos computacionales en física de materia condensada. II. Facultad de Ciencias, Univ Autónoma de Madrid, Spain. http://www.uam.es/departamentos/ciencias/fisicateoricamateria/especifica/hojas/kike/docto/ejer3/
32. Computer methods in chemical engineering, Maxwell–Boltzmann distribution function. http://terpconnect.umd.edu/~nsw/ench250/boltzman.htm
33. Eclipse—the eclipse foundation open source comm. Website. http://www.eclipse.org/
34. PHORTRAN—an integrated development environment and refactoring tool for Fortran. http://www.eclipse.org/photran/
35. GCC Wiki Automatic parallelization in GCC. http://gcc.gnu.org/wiki/AutoParInGCC
36. Intel Corporation Automatic parallelization with Intel compilers. http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers
37. The Portland Group PGI | products | PGI workstation. http://www.pgroup.com/products/pgiworkstation.htm