Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité de recherche associée au CNRS n° 1398

The Nestor library: a tool for implementing Fortran source to source transformations

Georges-André Silber
Alain Darte

September 1998

Research Report N° 98-42

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.00
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

The Nestor library: a tool for implementing Fortran source to source transformations

Georges-André Silber
Alain Darte

September 1998

Abstract

We describe Nestor, a library to easily manipulate Fortran programs through a high level internal representation based on C++ classes. Nestor is a research tool that can be used to quickly implement source to source transformations. The input of the library is Fortran 77, Fortran 90, and HPF 2.0. Its current output supports the same languages plus some dialects such as Petit, OpenMP, CrayMP. Compared to SUIF 2.0 that is still announced, Nestor is less ambitious, but is light, ready to use (`http://www.ens-lyon.fr/~gsilber/nestor`), fully documented and is better suited for Fortran to Fortran transformations.

Keywords: Library, program transformations, HPF, parallelization, object oriented.

Résumé

Dans ce rapport, nous décrivons Nestor, une bibliothèque pour manipuler facilement des programme Fortran à l'aide d'une représentation interne de haut niveau qui se fonde sur des classes C++. Nestor est un outil de recherche qui peut être utilisé pour implanter rapidement des transformation source à source. Les langages reconnus par la librairie sont Fortran 77, Fortran 90 et HPF 2.0. Les langages disponibles en sortie sont les précédents plus des dialectes de Fortran comme Petit, OpenMP, CrayMP, etc. Comparé à SUIF 2.0 qui est toujours annoncé, Nestor est moins ambitieux, mais il est léger, prêt à être utilisé (`http://www.ens-lyon.fr/~gsilber/nestor`) et complètement documenté. De plus, Nestor est mieux adapté aux transformations source à source de Fortran.

Mots-clés: Bibliothèque, transformation de programmes, HPF, parallélisation, orienté objet.

The Nestor library: a tool for implementing Fortran source to source transformations

Georges-André Silber   Alain Darte*

24th September 1998

Abstract

We describe Nestor, a library to easily manipulate Fortran programs through a high level internal representation based on C++ classes. Nestor is a research tool that can be used to quickly implement source to source transformations. The input of the library is Fortran 77, Fortran 90, and HPF 2.0. Its current output supports the same languages plus some dialects such as Petit, OpenMP, CrayMP. Compared to SUIF 2.0 that is still announced, Nestor is less ambitious, but is light, ready to use (`http://www.ens-lyon.fr/~gsilber/nestor`), fully documented and is better suited for Fortran to Fortran transformations.

1 Introduction and motivations

Several theoretical methods that transform programs to gain parallelism or to improve memory locality have been developed (see [24, 3] for surveys). Unfortunately, there is a gap between the bunch of known parallelism detection and code optimization algorithms, and those implemented in real compilers. Indeed, these algorithms are often difficult to implement, because they use graph manipulations, linear algebra, linear programming, and complex code restructuring (see for example [11, 23, 15, 9] for some parallelism detection algorithms). Consequently, their implementation is a research problem by itself and must be ease by a simple but powerful representation of the input program. This representation should provide all the basic blocks to let the researcher concentrate on algorithmic implementation problems, and hide the classical low level representation of the program (low level abstract syntax tree).

The Nestor library provides such a framework, focusing on tools for source to source transformation of Fortran and its dialects (Fortran 77, 90, HPF 2.0 [12], OpenMP [17], CrayMP, SunMP, etc.). We chose HPF and its variants because it offers useful means to express parallelized codes, such as directives for parallel loops, privatized arrays, data distributions, task parallelism, etc. Moreover, the obtained codes can be executed on a parallel computer after compilation by an HPF (or equivalent) compiler. Furthermore, the parallel code is still readable for the programmer and by Nestor itself. We believe that

*LIP, ENS-Lyon, 69007 Lyon, France. E-mail: {`gsilber`,`darte`}`@ens-lyon.fr`

this high level parallelizing approach (through directives insertion) is important to improve the relationship between the programmer and the compiler, and to enable semi-automatic parallelization (see Figure 1 for the compilation scheme).

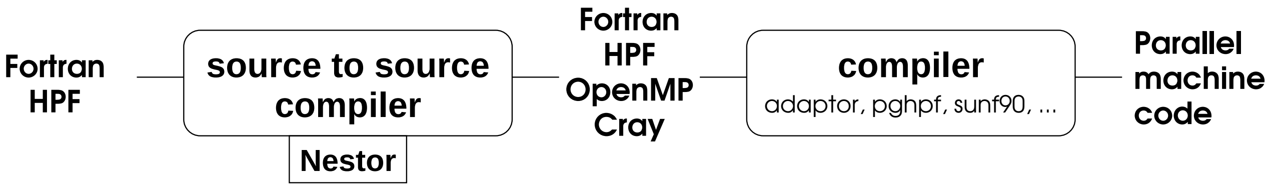

Figure 1: source to source parallelization of sequential Fortran.

Nestor is a C++ library that provides classes and methods to parse a source code in Fortran, to analyze or modify this source code easily with a high level representation, and to unparse this representation into Fortran with parallelization directives. The representation that can be manipulated with Nestor is a kind of AST (Abstract Syntax Tree) where each node is a C++ object that represents a syntactical element of the source code, and that can be easily transformed. The front-end of Nestor is the robust front-end of Adaptor [7], an excellent HPF compiler of the public domain written by Thomas Brandes. This front-end supports HPF 2.0 and Fortran 90.

Nestor aims to be used by researchers that want to implement high level transformations of Fortran source codes. It provides all the features for building a source to source Fortran transformer. The library is easy to use and is portable: it is written in GNU C++ and uses the STL (Standard Template Library) that implements classes for manipulating lists and other container objects. There is a full documentation in postscript and an on-line documentation in HTML and Java that describes all the classes of the library. The programmer has just to click on the hypertext links to follow the class hierarchy in order to find a description of their methods, their attributes, and some useful examples. This documentation can be found at the World Wide Web url: `http://www.ens-lyon.fr/~nestor/doc`.

This paper is organized as follows. In Section 2, we present the differences between Nestor and existing related tools, in particular SUIF. In Section 3, we describe Nestor in details. The section 4 gives examples of its usefulness and its ease of use. Then, we give some concluding remarks and some future work.

2 Related tools

Several research tools implementing program analysis or parallelization algorithms have been developed (Bouclettes [4], Petit [19], LooPo [13]). Their main objectives were to prove that these algorithms could be implemented in a real compiler. For example, Petit demonstrates the power of the Omega test and of the Omega library. Nevertheless, they were not conceived to handle real codes: they all take as input a toy language, usually a subset of Fortran.

But theoretical methods have to be validated also by testing their behaviors on real applications, such as benchmarks or scientific codes. Several more ambitious tools have been developed in the past: PIPS [18] developed at the Ecole des Mines de Paris, Polaris [21] developed at Urbana-Champaign, SUIF [20] developed at Stanford, PAF developed at PRiSM. These tools have been developed over more than ten years for some of them: they are now huge systems (difficult to maintain and extend) and their internal representations begin to be at a too low level for new developers. For example, SUIF 1.0 is currently moving to SUIF 2.0 with deep changes. People from Polaris think of changing their internal representation. We believe that Nestor could be an interesting platform for that.

One could argue that SUIF 2.0 has the same objectives as Nestor concerning the simplicity of use of its internal representation. Compared to SUIF 1.0, the main transformation is indeed to make the representation more object-oriented. Although this feature is already available in Nestor, when SUIF 2.0 is only announced, Nestor should not be seen as a rival tool for SUIF 2.0. First of all, Nestor is devoted to Fortran like languages, whereas SUIF has been designed for manipulating C programs (Fortran is handled through `f2c` and cannot be unparsed). Fortran is a simpler language than C, easier to optimize at a high level (in particular for dependence analysis), and thus it leads to a simpler internal representation.

In addition to this main difference in the input and output languages, Nestor is far less ambitious. It is not a full compiler, but just a kernel for source to source transformations of Fortran programs. Nestor does nothing! It only provides means to do something. But this limited goal gives it some advantages.

- Nestor is small. The library provides only the basic blocks for building source to source transformation systems. Its size allows an easy and quick installation on every system. It is developed and maintained by a single person.

- Nestor is fully documented. We think that this feature is maybe even more important than any other, as far as implementing algorithms is the main issue. There is a postscript documentation describing in details the whole library and each class. There is also an HTML/Java documentation that is very useful when developing with the library. This documentation is automatically generated from the source files and is always up to date, thanks to doc++, a public domain software [25].

- Nestor offers a Fortran90/HPF input and output. All the other tools support only Fortran 77. It is impossible for example to insert parallelizing directives a la HPF as easily as Nestor does (see examples in Section 4).

3 Description of Nestor

3.1 Implementation choices

The first choice was to choose a language to develop the library. This choice is important, because users that are going to use the library will have to write their transformations in this language. For several reasons, we have chosen C++ to develop Nestor.

- C++ is widely used, and a lot of existing libraries are written with this language (or in C). This means that these libraries can be used together with the Nestor library.

- C++ is object-oriented. The internal representation of a Fortran program fits very well in the object-oriented world. For instance, the inheritance principle gives several views of the internal representation. All objects composing it can be seen as a tree organization of NstTree objects, which is the base class of the library (each class inherits from it). But, the programmer can have a higher view of the representation by using the actual type of each object. There are no real NstTree objects, but only child objects of this class (NstStatement, NstDeclaration, etc.). This is explained in Section 3.3.

- C++ offers a lot of useful templates with the Standard Template Library (STL)[1]. This library is now widely available, especially in the GNU C++ compiler that is our compiler of development. This compiler is available on every usual platform. The choice of using the STL is then a reasonable choice, instead of investing a lot of time developing and maintaining new templates for lists, containers, etc.

- C++ offers operator overloading. This feature is useful, especially for writing the unparse: the operator `<<` is overloaded for each class of the library, allowing to write custom unparse and to write to a file or to standard output easily. It is also useful for creating new expressions (see Example 11 in Section 4.4).

- C++ offers virtual methods that are useful to define type methods. For instance, a virtual method `type()` is defined in each class and returns the actual type of the object, even if there is only a subpart of it that is available.

3.2 Writing a source to source transformation pass

When developing Nestor, we wanted to create a development platform for writing a source to source parallelizer. We had a lot of algorithms to implement and evaluate, and we wanted to automatically generate HPF programs starting from Fortran sequential programs. With Nestor, each transformation can be written independently, because the output of one pass can be used by another pass. It is one of the advantages of working at the source level. For instance, one pass can take a Fortran source code, insert HPF directives in front of `DO` loops and unparse the internal representation. One second pass can take the result of this pass and can generate the distributions of the arrays. The main idea is that each programmer can write its own optimization and test it immediately by compiling the result with a Fortran compiler that supports parallelization directives. Moreover, the result is easy to read because it is still written in a high level language instead of a tricky internal representation or in a low level language like C.

Example 1: The code example depicted in Figure 3.2 gives the scheme of a typical transformation pass written with our library. The first statement creates an object from the class `NstComputationUnit` that gives a starting point to the internal representation of the program passed in parameter in the command line. The last statement unparses the internal representation in Fortran on the standard output. This program adds, in front of the

[1] Included in the ANSI/ISO Standard C++ Library, corresponding to Committee Draft 14882 as revised through July 1997.

output, a comment with the name of the source file. It shows the simplicity of writing a transformation pass with the library.

```
#include <libnstbase.H>
void main(int argc, char** argv) {
   if (argc == 2) {
     NstComputationUnit prog(argv[1]);
     cout << "! file:" << argv[1] << endl;
     cout << prog << endl;
   }
}
```

Figure 2: C++ code of Example 1

```
!HPF$ INDEPENDENT
  DO I = 1,N
    A(I) = B(I) + 3
    D(I) = C(I) + 6
  ENDDO
```

Figure 3: Fortran code for Example 2

3.3 A quick look at the internal representation

Usually, compilers use program internal representations that are too low-level for user's manipulation. They do not fully retain the high-level language semantics of a program written in Fortran. One of the advantage of Nestor is that its internal representation is intuitive because there is a one-to-one correspondence between a syntactical element in a Fortran code and the Nestor object representing this element.

Example 2: Consider a `DO` statement in Fortran as given in Figure 3. It is composed of an index variable, a lower bound, an upper bound, possibly a step, and a list of statements. The following source code is an excerpt of the C++ definition of the Nestor class `NstStatementDo`, representing `DO` statements of Fortran.

```
class NstStatementDo : public NstStatement {
public:
  NstVariable* index() const;
  NstVariable* index(NstVariable& new_var);
  NstExpression* lower_bound() const;
  NstExpression* lower_bound(NstExpression& new_exp);
  NstExpression* upper_bound() const;
  NstExpression* upper_bound(NstExpression& new_exp);
  NstStatementList body;
  bool independent;
  ...
};
```

Each element of a `DO` can be accessed or modified by the corresponding method in the class. A call to the method `index()` returns an object representing the index variable of the loop. A call to the method `index(j)` replaces the old index variable by a new object `j`. Each element of the `DO` can be accessed/modified the same way. The statements in the body of the loop are stored in the list of statements `body`. This is a doubly-linked list that can be modified with usual operations on a list: add, delete, traversal, ... The flag `independent` tells if the loop is parallel or not and unparses an `!HPF$ INDEPENDENT` in front of the loop if the flag is set (or the equivalent directive for other dialects).

The class `NstStatementDo` inherits from the class `NstStatement` and then has all the methods and attributes of this class. The class `NstStatement` contains all the informations that are common to all types of statements in a Fortran code: line number in the source code, label of the statement, ... This class inherits from the class `NstTree`, like all the other classes of the Nestor library.

3.4 Type of objects

In the AST created with the Nestor library, each object (i.e., each instance of a class) has a type. Each class inherits (directly or indirectly) from the class `NstTree`. This class defines a virtual method `type()` that is redefined in each subclass. This method is very useful, because it introduces some genericity in the internal representation and in its use. Remember the class `NstStatementDo` (see Example 2). Inside, the attribute `body` is a list of statements. An object representing a statement can be an instance of one of these classes:

```
NstStatementContinue   NstStatementBasic    NstStatementWhere
NstStatementWhile      NstStatementIf       NstStatementNestor
NstStatementDo         NstStatementForall
```

It is difficult to make a list containing objects of different types, this is why a list of statements is a list of `NstStatement` objects. Each statement inherits from this class, the method `type()` will give the actual type of the object.

Example 3: This example is a traversal of a list of statements that prints the number of specified parallel loops. It illustrates the use of the `type()` method, and the use of the lists defined in the Standard Template Library). Lists can be traversed with `iterator` objects.

```
void print_number_of_par(NstStatementList& sl) {
  int num = 0;
  NstStatementList::iterator sl_it;
  NstStatement* current_stat;
  NstStatementDo* do_loop;
  for (sl_it = sl.begin(); sl_it != sl.end(); sl_it++) {
    current_stat = *sl_it;
    if (current_stat->type() == NST_STATEMENT_DO) {
      do_loop = (NstStatementDo*) current_stat;
      if (do_loop->independent) ++num;
    }
  }
  cout << "There are " << num << " parallel loops." << endl;
}
```

If you take a look at the example above, you can see the three lines:

```
    if (current_stat->type() == NST_STATEMENT_DO) {
      do_loop = (NstStatementDo*) current_stat;
      if (do_loop->independent) ++num;
    }
```

Instead of doing a cast from the object `NstStatement` to the object `NstStatementDo`, there are some safe cast methods implemented from each parent class to its child. Here, these three lines can be replaced by:

```
do_loop = current_stat->isNstStatementDo();
if (do_loop && do_loop->independent) ++num;
```

The virtual method `isNstStatementDo()` returns `NULL` if the actual object is not an object of the class `NstStatementDo` and a pointer to the `NstStatementDo` object otherwise.

3.5 Traversals

The class `NstTree` gives a mechanism to write recursive traversals of the AST. The class `NstTree` provides the two virtual methods `init()` and `next()` that gives respectively the first and the next child of a `NstTree` (or a derived class) object.

Example 4: A recursive traversal of an AST that prints the type of each object encountered. The method `class_type()` is defined for each class and returns a string containing the type of the class. This example illustrates the use of the `init()` and `next()` methods. These methods are defined for each class, even for classes representing lists. Consequently, they can be used to traverse a list instead of using an object `iterator`.

```
void print_type(NstTree* t)
{
  cout << t->class_type() << endl;
  NstTree* current = t->init();
  while (current) {
    print_type(current);
    current = t->next();
  }
}
```

The class `NstTree` also provides the method:

```
void traversal(int (*action)(NstTree*, void*), void* extra = NULL);
```

that executes a recursive traversal of the object, executing the function `action` at each node. The recursion stops when `action` returns a nonzero value. There is a class `NstTreeTravel` that can be extended to write more complicated recursive travels.

3.6 The front-end

The front-end of Nestor is a slightly modified version of the front-end of Adaptor [7] that recognizes HPF 2.0 [12][2]. Adaptor has been written by Thomas Brandes and is an excellent public domain HPF compiler. Its front-end is robust, publicly available, and has been written with the GMD compiler toolbox [7] (a high level language to easily describe grammars). We have added the directives of OpenMP and Cray. This front-end allows Nestor to handle real codes, instead of only considering a subset of language or an *ad-hoc* language. This part is also useful because it checks the syntax and the semantics of the code. Once the code has been parsed, the resulting internal representation is a correct Fortran program.

For supporting semi-automatic parallelization, it is useful to define parts of code that have to be parallelized and others that must be ignored. The library offers a mechanism, by the use of new directives, that permits to ignore parts of codes that are known to be sequential and to emphasize parts that must be parallelized. These directives are comments and do not modify the compilation of the code. They begin with the keyword `!NESTOR$`. The directive `!NESTOR$ SINGLE` is to be placed in front of a loop (`DO` or `FORALL`), it sets a flag in the corresponding Nestor object. The second directive (`!NESTOR$ BEGIN`, `!NESTOR$ END`) defines a region of code. The library provides functions to retrieve the marked statements.

There is only one way to parse a code with the Nestor library, with the constructor of the `NstComputationUnit` class. The following code:

```
NstComputationUnit file_parsed("example.f");
```

parses the Fortran file `example.f`, checks if the source code is a correct Fortran program, and creates an object `file_parsed` that contains the Nestor representation of the source code. This internal representation captures all the information of the source code. A table of symbols is created for each subroutine or function of the source code. Figure 6 in Appendix A gives an example of an internal representation for a simple program.

The constructor has some parameters that can be configured, for instance to parse Fortran 90 free code format or to tag all the code as if `BEGIN-END` directives were enclosing all the statements of the code. The object `NstComputationUnit` has also two special tables of symbols that store the externals and the intrinsics of the Fortran source code. By default, the table of intrinsics knows all the Fortran 90 and HPF intrinsics.

3.7 The back-end

The internal representation of Nestor can be unparsed in Fortran 77, Fortran 90, HPF 2.0, OpenMP, CrayMP directives and in the Petit language. Each object has its own unparse methods, one for each of the Fortran dialect. Unparsing recursively an object is a very simple task by the overload of the C++ operator «. The unparsed language can be chosen by a global flag. By default, the unparsed language is HPF 2.0.

[2]Historically, Nestor has been written as a parallelizing pre-phase for Adaptor. Starting from a sequential Fortran program, Nestor+Adaptor can transform it into a parallel program with message passing. The name Nestor comes from the term *loop nest* and the name Adaptor.

Example 5: The example below is extracted from the code of the Nestor library. This is the unparse method that is called when a `NstStatementIf` is unparsed in Fortran. This example shows how the overload of the operator « leads to a simple and clean code.

```
void NstStatementIf::unparse_fortran(ostream& s) const
{
  indent(s); s << "if (" << *condition() << ") then" << endl;
  inc_indent();
  s << then_part << endl;
  dec_indent();
  if (else_part.size() > 0) {
    indent(s); s << "else" << endl;
    inc_indent();
    s << else_part << endl;
    dec_indent();
  }
  indent(s); s << "end if";
}
```

3.8 Dependences and graphs

Dependence analysis is the first step before any optimization that modifies the order of computations in a program. Without a sophisticated dependence analyzer, code transformations such as loop transformations, scalar expansion, array privatization, dead code removal, etc. are impossible. Therefore, any parallelizing tool must contain a dependence analyzer.

Nevertheless, it is well known that the development of a dependence analyzer both powerful and fast is a very hard task. This comes from the fact that the problem is in theory NP-complete, if not undecidable, but that it can be fasten in practice thanks to a pool of ad-hoc methods devoted to frequent cases. For this reason, we decided to rely on a free software tool, named Petit [19], developed by Bill Pugh's team at the University of Maryland. Petit's input is a short program, written in a restricted language, close to - but different than - Fortran 77. Its output is a file that describes pairs of array references involved in a dependence, and this dependence is represented by a (sometimes complicated) relation based on Presburger arithmetic.

A possibility to integrate Petit into Nestor was to plug all Petit's techniques directly into Nestor's abstract syntax trees. However, since Nestor accepts the full Fortran, and Petit only a simple Fortran-like language, it is not so simple to modify the sources of Petit, even for the creators of Petit themselves. Following Bill Pugh's advice, we chose to use Petit as an independent tool through its input and output files. This strategy is not only simpler to implement, it is also more portable: potential bugs in Petit and potential bugs in Nestor are separated, and furthermore updating Petit to new versions will be easier. Two problems still remained: feeding Petit with a correct input, and plugging Petit's output at the right place into the original Fortran code.

The first task was easy to complete thanks to the clean design of Nestor. As mentioned before, Nestor is a C++ library. The unparse function (i.e. the function that transforms an abstract syntax tree into a program in a given language) has been defined simply as the output operator (the C++ operator «). Therefore, we just had to redefine this operator for all C++ classes that have their equivalence in the Petit language. For example, the operator «, applied to the class that corresponds to a Fortran `DO` loop, automatically generates a loop in Petit's format, and recursively applies the operator « to the body of the loop. In Nestor, there is a global flag that determines the output language chosen by the unparse function, and that switches from one to another.

The second task was twofold. First, we had to make the correspondence between array references in Petit and the original array references of the Fortran program. Line numbers are not sufficient because both languages can be formatted in a different manner, and furthermore, only a part of the original code may be sent to Petit. Therefore, we slightly modified Petit's grammar so as to number array references in the same order as they appear in Nestor's abstract syntax trees. Second, we modify the way dependences are represented in Petit's output. Indeed, in most parallelizing algorithms, what we need is an approximation of distance vectors, and not a too complicated Presburger formula. We wrote a small tool, based on the Polylib [22], a library for manipulating polyhedra, developed at IRISA in Patrice Quinton's team (mainly by Doran Wilde and Hervé Le Verge). This tool extracts, from a Presburger formula, a description of dependences by level, direction vector, and polyhedral approximation, the three representations used respectively by the parallelizing algorithms of Allen and Kennedy [1], Wolf and Lam [23], and Darte and Vivien [9].

We point out that we don't need to send the full program to Petit. Indeed, we use Petit only to analyze small portions of codes that we want to parallelize: the unparse function of Nestor builds the corresponding code in Petit's format, and also creates the declaration part of this small program, based on all variables that are used in this portion. For example, if we decide to analyze a single loop, surrounded by an outer loop, then the loop counter of the outer loop becomes a parameter that must be declared in Petit's input. This "local" unparsing technique allows us to manipulate large codes, even if Petit is limited to the analysis of small codes.

Example 6: The very simple example below illustrates the dependence representations that are now available in our Petit implementation. It is the typical case where level and direction vectors are not sufficient to detect parallelism.

```
   DO I=1,n
      DO J=1,n
S:       A(i,j) = A(j,i) + A(i,j-1)
      ENDDO
   ENDDO
```

Our tool detects three dependences: a flow dependence, due to the read `A(i,j-1)`, of level 2, direction vector $(0, 1)$ and whose polyhedral approximation is the singleton $(0, 1)$, and two other dependences, one flow dependence and one anti dependence, both due to the read `A(i,j)`, and of level 1, direction vector $(1+, 1-)$ and whose polyhedral approximation is the polyhedron with one vertex $(1, -1)$ and one ray $(1, -1)$.

Building a dependence graph is a very easy task and is completed by a call to the constructor of the class `NstRDGVar`. This class contains the list of edges and the list of vertices of the dependence graph. The constructor builds the Petit input, calls Petit and retrieves the output of Petit to build the dependence graph. Each vertex is linked to the corresponding variable access in the AST of Nestor. Classical graph manipulation algorithms, such as computations of connected components, of strongly connected components, topological ordering, etc. are provided.

3.9 Automatic parallelization

Nestor already implements two algorithms for parallelism detection. These algorithms are very simple and were implemented to validate the internal representation of Nestor and to check its ease of use. The first algorithm only detects if the loop is parallel without any modification (see Example 9 of Section 4 for a description). The second is a modified version of the Allen-Kennedy algorithm [1]. Our goal with this modified version was to have at least one robust algorithm, able to handle complex loops with conditionals and possibly non constant loop bounds, in other words structured codes that may contain control dependences. Many extensions of the Allen-Kennedy algorithm have been proposed in the literature that are able to handle control dependences. All of them rely on the creation of "execution variables" (scalar or array variables) that are used to pre-compute and store the conditionals, and on the conversion of control dependences into data dependences [2, 14, 16].

While implementing such an algorithm, we found out that it was difficult, in general, to determine the size of these new arrays, especially in parameterized codes. Furthermore, in the context of High Performance Fortran and distributed memory systems, the problem of aligning and distributing these new arrays arises. To avoid these two problems, it may be better to re-compute the conditionals (when it is possible) instead of using a stored value. It may also be better to manipulate privatized arrays or scalars than to manipulate distributed arrays. We therefore tried to understand how these two new constraints – the control of the new array dimensions, and the re-computations of conditionals – can be handled, since no previously proposed algorithm can take them into account.

For that, we explored a new strategy for taking control dependences into account. The technique is to pre-process the dependence graph, and once this process is achieved, any version of the Allen-Kennedy algorithm can be used: the dimensions of the new arrays are guaranteed to satisfy the desired constraints. To make things simpler, the automatic version that is currently implemented in Nestor is a version that guarantees that all new variables are at worst privatized scalar variables (thus, with no size to declare). A semi-automatic version offers to the user the choice of the array dimension he tolerates for his program.

On Figure 5, we give a parallel version of the code of Figure 4, obtained with our algorithm. The important thing here is the new array `nst_0` introduced to hold the values of the loop upper bound. This code is correct only because this array is privatized. In the implementation of our algorithm, this array has been privatized by adding it to the list of privatized variables in the `NstStatementDo` class, with the instructions:

```
NstIdentifier new_id;
NstDeclarationVariable new_array(program, new_id, array_type);
```

```
do I = 1,N
  do J = 2,N
    do K = 2, A(I,J+1,1)
      A(I,J,K) = B(I,J,K)+A(I,J-1,K)
      C(I,J,K) = A(I,J-1,K-1)+D(I,J,K)
    enddo
  enddo
enddo
```

Figure 4: Original code.

```
!HPF$ INDEPENDENT, NEW(nst_0)
  do I = 1,N
!HPF$ INDEPENDENT
    do J = 2,N
      nst_0(J) = A(I,J+1,1)
    end do
    do J = 2,N
!HPF$ INDEPENDENT
      do K = 2,nst_0(J)
        A(I,J,K) = B(I,J,K)+A(I,J-1,K)
      end do
    end do
!HPF$ INDEPENDENT
    do J = 2,N
!HPF$ INDEPENDENT
      do K = 2,nst_0(J)
        C(I,J,K) = A(I,J-1,K-1)+D(I,J,K)
      end do
    end do
  end do
```

Figure 5: Code after parallelization.

```
do_loop.new_variables.push_back(&new_array);
```

During the unparse, the array `nst_0` appears in the list of all variables that are declared `NEW` in the `!HPF$ INDEPENDENT` directive.

4 Examples

We now illustrate some features of Nestor through examples.

4.1 Printing unit names

Example 7: This example prints on the standard output the name of all the units in the Fortran file passed as parameter in the command line. It illustrates the use of the `traversal` methods. The parameter of this method is a function, called for each node during the traversal, passing the node as parameter of the function. Note the use of the cast method `isNstUnit()` that returns a pointer to a `NstUnit` object if the `NstTree` object is a unit.

```
# include <libnstbase.H>

int print_unit_name(NstTree* t, void* ignored) {
  NstUnit* unit = t->isNstUnit();
```

```
  if (unit) cout << *unit->name() << endl;
  return 0;
}

void main(int argc, char** argv)
{
  if (argc == 2) {
      NstComputationUnit file(argv[1]);
      file.traversal(print_unit_name);
  }
}
```

4.2 Renaming a variable

Example 8: This example renames all the variables of a unit by giving them a unique identifier that is not present anywhere in the entire internal representation. This could be the first pass of the inlining of a function call. This example illustrates the fact that each access to a variable refers to the object stored in the symbol table of the unit, so it is the only place where we have to change the identifier. The call to the constructor `NstIdentifier()` creates a unique identifier by the use of a hash table that stores all the identifiers of the internal representation. Consequently, Nestor can quickly checks if the identifier is already used.

```
void rename_all_identifiers(NstUnitSubroutine& us)
{
  NstSymbolTable sb = us.object()->symbols();
  NstTree* current;
  NstIdentifier* name;

  current = sb.init();
  while (current) {
    if (current->type() == NST_OBJECT_VARIABLE) {
      name = new NstIdentifier();
      ((NstObject*)current)->identifier(*name);
    }
    current = sb.next();
  }
}
```

4.3 Checking if a loop is parallel

Example 9: This example builds a dependence graph from a statement and checks if there are no dependences carried by the loop (in this case, the loop is parallel). Note that this simple example is a parallelizer from Fortran to HPF for a shared memory machine (no distributions are generated). This example illustrates the use of a dependence graph that contains a list of edges labeled with dependences and a list of vertices representing statements. The function `check_level` checks if there is a dependence carried by the loop.

```
int check_level(NstRDGVar& dg, int level)
{
  NstEdgeList::iterator el_it;
  NstDependence* dep;

  for (el_it = dg.edges.begin(); el_it != dg.edges.end(); el_it++) {
    dep = (NstDependence*) (*el_it);
    if (dep->level == level) return 1;
  }
  return 0;
}

int parallel_loop(NstTree* t, void* ignored)
{
  NstStatementDo* stdo = NULL;
  if (t->type() == NST_STATEMENT_DO) stdo = (NstStatementDo*) t;
  if (stdo) {
    NstRDGVar dep_graph(*stdo);
    if (dep_graph.built()) { // graph successfully built
      stdo->independent = !check_level(dep_graph, 1);
    }
  }
  return 0;
}

void main(int argc, char** argv)
{
  if (argc == 2) {
    NstComputationUnit file(argv[1]);
    file.traversal(parallel_loop);
    cout << file << endl;
  }
}
```

4.4 Creating a program

Example 10: In the previous examples, the internal represention is created with a call to the `NstComputationUnit` constructor. In this example, we illustrate the fact that we can create new objects and create easily an internal representation. This last example creates the following program from scratch (without any parsing):

```
PROGRAM ESSAI
INTEGER*4 I
I = 3
END PROGRAM ESSAI
```

The corresponding internal representation is represented by Figure 6 in Appendix A. Here is the C++ code that creates and unparses the previous Fortran code:

```
#include <libnstbase.H>
void main()
{
  // Create the program
  NstIdentifier id_essai("essai");
  NstUnitProgram prog_essai(id_essai);

  // Declares the variable I in prog_essai
  NstIdentifier id_i("i");
  NstDeclarationVariable decl_i(prog_essai, id_i, nst_integer_type);

  // Creates an assignment instruction
  NstVariableUsed use_i(*decl_i.object());
  NstInteger three(3);
  NstSimpleAssign assignment(use_i, three);

  // Creates the statement and add to the program
  NstStatementBasic stat_assign(assignment);
  prog_essai.statements.push_back(&stat_assign);

  cout << endl << prog_essai << endl;
}
```

Example 11: This example illustrates the fact that some operators are overloaded to create new expressions. The object `new_exp` contains the expression `i*5+j*4`. All the usual operators are overloaded.

```
NstExpression* create_exp(NstVariableUsed& i, NstVariableUsed& j)
{
  NstExpression* new_exp = i * 5 + j * 4;
  return new_exp;
}
```

5 Conclusion and future work

This paper provides a description of the Nestor library. We think that this library is very useful for the researcher who wants to implement and test new source to source transformations. Our library has a front-end and a back-end that totally supports Fortran and its dialects, and an object-oriented internal representation that eases the process of implementing new algorithms. Furthermore, it is fully documented, small, robust, and easy to install on every system.

Several researchers are already interested by Nestor, especially by the fact that it is both light and practical. We hope that Nestor is going to be effectively widely used by researchers for implementing new parallelization strategies. For the time being, Nestor is used at LIP by researchers involved in automatic parallelization and high level transformations. It is used in the project Alasca for automatic insertion of HPF data redistributions, it is used for inserting automatically low overhead communication and computation subroutines in Fortran codes [10], it is used in high level loop transformations before compilation to VHDL, and it is used in the project HPFIT [5, 6] to implement parallelization algorithms. Nestor is now publicly available with its source code and its documentation at the address

`http://www.ens-lyon.fr/~gsilber/nestor`

We are implementing new parallelization algorithms [8] into it. These parallelization algorithms could be included in the base Nestor package and then transform it into a more powerful source to source automatic parallelization kernel.

References

[1] John Randy Allen and Ken Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[2] John Randy Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of Control Dependence to Data Dependence. In *Conference Record of the Tenth Annual ACM Symposium on the Principles of Programming Language*, Austin, Texas, January 1983.

[3] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4), 1994.

[4] Pierre Boulet. Bouclettes: A fortran loop parallelizer. In *HPCN 96*, pages 784–791, Bruxelles, Belgium, June 1996. Springer Verlag Lecture Notes in Computer Science.

[5] T. Brandes, S. Chaumette, M.-C. Counilh, A. Darte, J.C. Mignot, F. Desprez, and J. Roman. HPFIT: A Set of Integrated Tools for the Parallelization of Applications Using High Performance Fortran: Part I: HPFIT and the TransTOOL Environment. *Parallel Computing*, 23(1-2):71–87, 1997.

[6] T. Brandes, S. Chaumette, M.-C. Counilh, A. Darte, J.C. Mignot, F. Desprez, and J. Roman. HPFIT: A Set of Integrated Tools for the Parallelization of Applications Using High Performance Fortran: Part II: Data-structure Visualization and HPF extensions for Irregular Problems. *Parallel Computing*, 23(1-2):89–105, 1997.

[7] Thomas Brandes. ADAPTOR, High Performance Fortran Compilation System. World Wide Web document, `http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html`.

[8] Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling. *Parallel Processing Letters*, 7(4):379–392, 1997.

[9] Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *International Journal of Parallel Programming*, 25(6):447–497, 1997.

[10] F. Desprez and B. Tourancheau. LOCCS: Low Overhead Communication and Computation Subroutines. *Future Generation Computer Systems*, 10(2&3):279–284, June 1994.

[11] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Int. J. Parallel Programming*, 21(6):389–420, December 1992.

[12] High Performance Fortran Forum. High Performance Fortran Language Specification. Technical Report 2.0, Rice University, January 1997.

[13] The group of Pr. Lengauer. The loopo project. World Wide Web document, `http://brahms.fmi.uni-passau.de/cl/loopo/index.html`.

[14] Ken Kennedy and Kathryn S. McKinley. Loop Distribution with Arbitrary Control Flow. In *Supercomputing'90*, August 1990.

[15] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, January 1997.

[16] Kathryn S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Department of Computer Science, Rice University, 1992.

[17] OpenMP Standard for Shared-memory parallel directives. World Wide Web document, `http://www.openmp.org`.

[18] PIPS Team. PIPS (Interprocedural Parallelizer for Scientific Programs). World Wide Web document, `http://www.cri.ensmp.fr/~pips/index.html`.

[19] William Pugh. Release 1.10 of Petit. World Wide Web document, `http://www.cs.umd.edu/projects/omega/`.

[20] Stanford Compiler Group. The SUIF Compiler System. World Wide Web document, `http://suif.stanford.edu/suif/suif.html`.

[21] The Polaris Group. Polaris Project Home Page. World Wide Web document, `http://polaris.cs.uiuc.edu/polaris/polaris.html`.

[22] Doran K. Wilde. A library for doing polyhedral operations. Master's thesis, Oregon State University, Corvallis, Oregon, Dec 1993. Also published in IRISA technical report PI 785, Rennes, France; Dec, 1993.

[23] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distributed Systems*, 2(4):452–471, October 1991.

[24] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[25] Roland Wunderling and Malte Zöckler. A documentation system for C/C++ and Java. World Wide Web document, `http://www.zib.de/Visual/software/doc++/index.html`.

A Example of an internal representation

Figure 6 gives the example of the internal representation representing the following program:

```
! file essai.f
PROGRAM ESSAI
INTEGER*4 I
I = 3
END PROGRAM ESSAI
```

This internal representation is obtained after a call to the constructor:

```
NstComputationUnit essai("essai.f");
```

or by the execution of the program of Example 10.

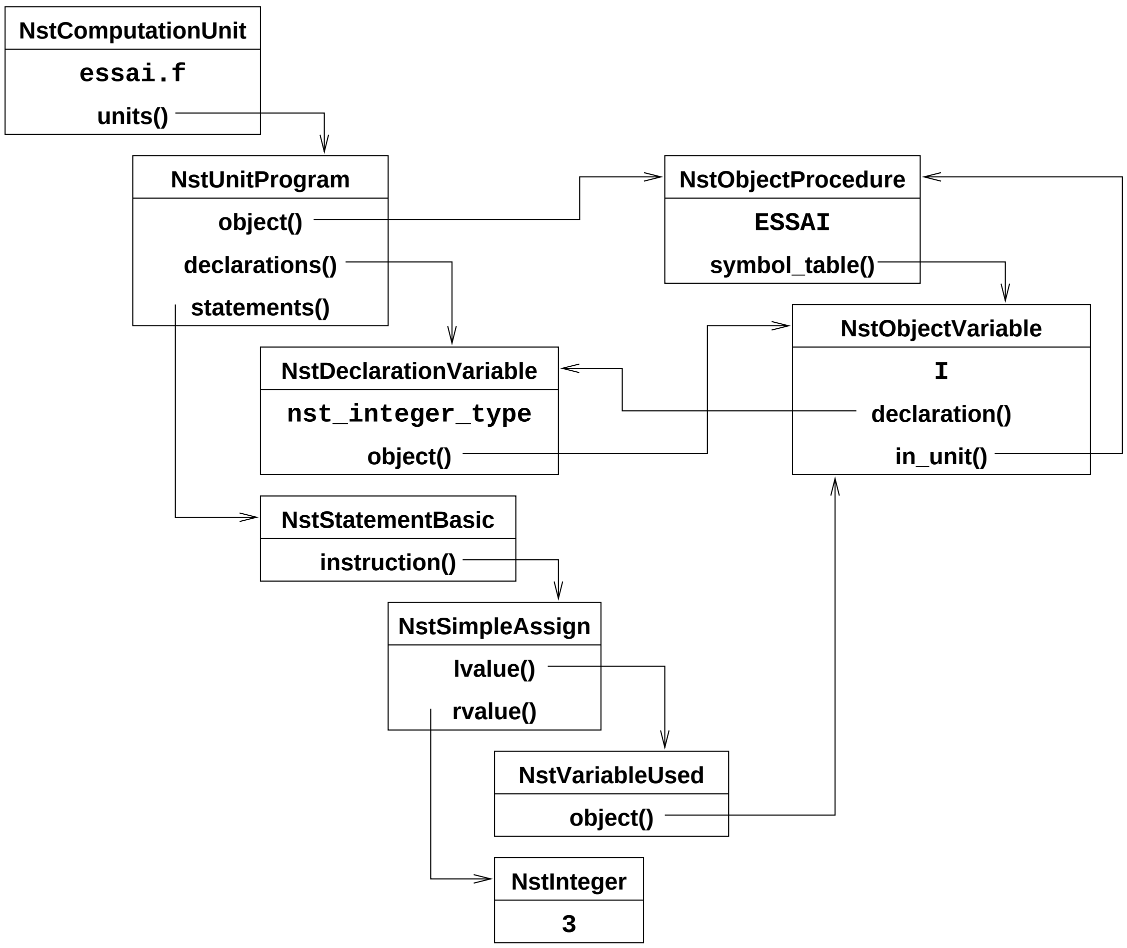

Figure 6: internal representation of the Fortran program ESSAI.