MÉMOIRE

en vue de l'obtention d'une

HABILITATION A DIRIGER DES RECHERCHES
spécialité: informatique

devant l'École Doctorale STIC de
l'Université de Nice Sophia-Antipolis

par

Laurent HASCOËT

Analyses statiques et transformations de programmes: de la parallélisation à la différentiation

Soutenue le 28 janvier 2005 devant le jury composé de:

M. Jacques BLUM	président
M. Christian BISCHOF	rapporteur
M. Andreas GRIEWANK	rapporteur
M. Francois IRIGOIN	rapporteur
M. Mike GILES	
M. Gilles KAHN	
M. Mohamed MASMOUDI	
M. Vittorio SELMIN	

Static Analyses and Transformations of Programs: from Parallelization to Differentiation

Laurent Hascoët

January 28, 2005

Abstract

We present a synthesis of research on tools for automatic program analysis and transformation. We give basic principles of static analysis of programs, and focus on two specific transformations: Automatic Parallelization and Automatic Differentiation (AD). We show how the formalism developped for parallelization is applied for a more efficient AD. We also present research on the usage in scientific computing of derivatives obtained through AD. We describe in detail the software tools that we developed for parallelization and AD, and we show perspectives of future research in AD.

Keywords: Compilation, Static Analysis, Data Flow Analysis, Program Transformation, Parallelization, Loop Nest Parallelization, Program Optimization, Data Dependency, Parallelization Tool, SPMD parallelization, Automatic Differentiation, Automatic Differentiation Tool, Scientific Computing, Optimal Shape Design, Optimization, Inverse Problems, Data Assimilation, Adjoint Models.

Résumé

Nous faisons une synthèse de recherches dans le domaine des outils d'analyse et transformation automatiques de programmes. Nous décrivons les principes des analyses statiques de programmes, et plus particulièrement la Parallélisation et la Différentiation Automatique (DA). Nous soulignons comment le formalisme développé pour la parallélisation peut être utilisé pour améliorer les performances de la DA. Nous présentons aussi des recherches sur l'utilisation efficace en calcul scientifique des dérivées calculées grâce à la DA. Nous décrivons en détail les outils logiciels que nous avons développés pour la parallélisation et la différentiation, et nous donnons des pistes pour des recherches futures en DA.

Mots-clés: Compilation, Analyse Statique, Analyse de Flot de Données, Transformation de Programme, Parallélisation, Nids de boucles, Optimisation de Programmes, Dépendances des Données, Outil de Parallélisation, Parallélisation SPMD, Différentiation Automatique, Outil de Différentiation Automatique, Calcul Scientifique, Optimisation de Formes, Problèmes Inverses, Assimilation des Données, Modèles Adjoints.

Contents

List of Figures

Chapter 1

Introduction

– Should we leave computer programming to humans ?

The answer seems obvious:

– Of course we should ! Who would do it otherwise ?

At second thought, the answer is not so clear. All computer users know that most codes are imperfect and contain errors. All programmers know that this comes from programming errors made by fellow programmers. After some maturation, most experienced programmers will even admit that they can't avoid errors in their own code ! So human-made programs are unsafe.

Also, almost anyone in charge of a big code development project will complain about the time it takes to write a code. Delays are short and success is uncertain. Several studies found an incredibly low average production of a dozen lines of code per programmer per day. So human-made programs are expensive.

Third, program design is certainly more interesting and more creative than program writing. Designing algorithms and data structures can be thrilling, implementing a small prototype can be fun, but writing the complete code can be a bore. Making modifications and updates into a large existing code rank among the most tedious activities with computers. So computer programming is most often boring.

– After all, maybe we don't want to leave programming to humans !

Indeed, it is true that humans are progressively relieved from programming tasks. Almost no one writes machine code nor assembly language any more. Programming languages are more abstract, and the effective computer programming task is done by a compiler, which is one of the most complex programs on a computer. The advantages are clear: writing a program in an elaborate programming language is faster, more interesting, and translation to machine code introduces virtually no error.

This evolution doesn't stop at compilers. On one hand, computer architectures offer more power, at the cost of increased complexity. Compilers can take this into accout very decently, but to take full profit of multi-level cache or parallel architectures often requires specialized code analyzers. These tools, with the help of the programmer, can transform a program into an equivalent program that will perform better on the target architecture. Programmers are relieved from this code optimization task, and their input comes at a more abstract level.

Similarly, programmers need to update or to adapt increasingly large existing codes. Code understanding, sometimes called reverse engineering, is crucial. This is not done only by hand but also with tools that analyze existing codes. These tools can even find subtle errors, especially when the original programming language is abstract enough to embed some specification of the desired results. We see here that there is a mutual benefit to program at higher abstraction levels, both for initial code developpers and for following code adapters. These analyses are called *static* or *compile-time*, opposed to *dynamic* or *run-time*, because their result must be valid for every possible program input. This constraint is strong, but static analyses take up this challenge.

An ultimate objective is to replace *programming* languages by *specification* languages. Technology has taken several steps in this direction. For instance *declarative* languages like PROLOG describe what the program result must be, rather than how it must be computed. It is the task of a separate tool to decide how this result will actually be computed. The declarative program can be used as an abstract specification [9], on which some proofs can be made, with the help of other tools. This can lead to *certified* programs. Similarly, one can check that the compiler that builds the running program actually builds a program which has the same semantics as the original specification. This leads to *certified* compilers.

Programs are growing more and more ambitious, especially in Scientific Computing. Complete generation of a scientific application from specifications is still out of reach. But there are several ways to generate some *parts* of the application. One classical example is symbolic computation environments, such as MAPLE, where equations can be translated into small program pieces. We shall give special attention to Automatic Differentiation (AD), where the input specification is a program part that evaluates some function, and the generated program part computes some mathematical derivatives of the function. AD has very promising applications in Scientific Computing, especially for inverse problems and optimization. We claim that AD is just another program transformation, which will take profit from existing compiler and parallelizer technology.

1.1 What this report is about

This report presents a synthesis of work done in the areas of program static analysis and transformation. We use classical compiler theory, along with some specific extensions, to study tools that generate new programs from existing programs. This generation is semi-automatic: on one hand it depends on information obtained by static analysis of the given program, but on the other hand it uses information provided by the end-user, because static analyses are limited and cannot find everything. We study in detail two program transformations: **Automatic Parallelization** and **Automatic Differentiation (AD)**. AD is more recent, and therefore it was very interesting to transpose the techniques and tools developed for parallelization to AD.

Essentially, these compiler techniques are based on graph theory: the internal representation of programs combines trees (*abstract syntax trees*) and several sorts of directed cyclic graphs (*call graphs, flow graphs, dependence graphs*). Analyses and transformations boil down either to computation of attributes on trees, or graph algorithms such as topological sorting, loop detection or acyclic condensation. This graph representation is also adequate to prove essential properties, such as algorithm termination. These analyses are implemented by programs and run on programs, therefore most of them are undecidable. Informally, this means that the analysis result will often be approximate, or even unknown. Other approximations come from cost/benefit tradeoffs in the analysis, such as coarse approximations on pointers or arrays. When arrays are concerned, analysis of indexes is often approximate. In favourable cases, array index analysis can be done using linear programming techniques on integers, yielding precise information. After analysis comes the actual program transformation, where finding the optimal choice is often a NP-hard question. Therefore, heuristics must be developed. A typical example is finding a good storage/recomputation tradeoff to speed up the computation of derivatives for a large expression.

In our work, we give a particular importance to the actual implementation of usable tools. We think this is necessary to validate our techniques on real programs, and also to get feedback from industrial users. As a consequence, we can't restrict our prototypes to mini-languages or clean academic programs. The three application tools that we describe in sections 3.4, 3.5, and 4.6 are meant to be real tools, working on full FORTRAN77 and FORTRAN95. This implies some constraints on the architecture, such as a clear separation between a language-specific front-end and back-end, and a more language-independent middle-end that does the analysis and transformation. In this middle-end, many fundamental static analyses are common, and can be shared so that a tool can be kept as a platform to develop another program transformation tool. In order to adress industrial programs, it also requires that analyses perform well not only on clean, structured, academic programs, but also do their best on deprecated features such as unstructured jumps or `GOTO`'s, or `COMMON-EQUIVALENCE` declarations. We believe this is also interesting from a research point of view. For example in section 3.4.4, we show an extension of a loop parallelization algorithm that can extract parallelism from "dirty" loops with jumps and old-fashioned style.

1.2 How to read this report

We organized this report to reflect this transposition process, from software technology developped for compilation and parallelization, towards new program transformations such as Automatic Differentiation, which is our current research focus. Therefore, after general comments on static analyses and transformations of programs in chapter 2, the two following chapters 3 and 4 highlight this transposition in their structure. Figure 1.1 displays the correspondence between chapter 3 devoted to Parallelization, and chapter 4 devoted to Automatic Differentiation. Obviously, AD is less famous than Parallelization, and therefore an introductory section 4.1 is necessary, as well as some motivating applications for this technology in section 4.2. Apart from that, the structure is similar. We identify three major aspects of compiler technology, namely the internal representation of analyzed pro-

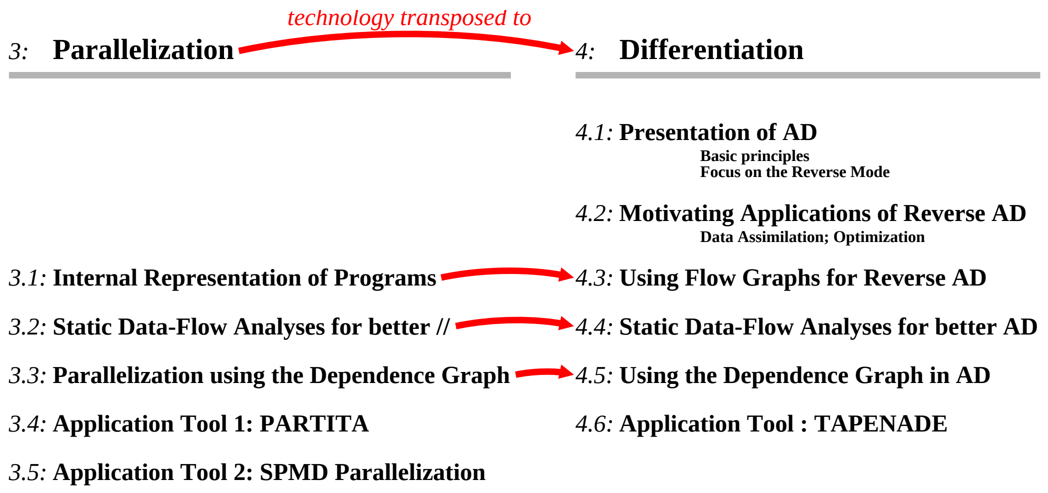

technology transposed to
3: Parallelization
4: Differentiation
4.1: Presentation of AD
Basic principles
Focus on the Reverse Mode
4.2: Motivating Applications of Reverse AD
Data Assimilation; Optimization
3.1: Internal Representation of Programs
4.3: Using Flow Graphs for Reverse AD
3.2: Static Data-Flow Analyses for better //
4.4: Static Data-Flow Analyses for better AD
3.3: Parallelization using the Dependence Graph
4.5: Using the Dependence Graph in AD
3.4: Application Tool 1: PARTITA
4.6: Application Tool : TAPENADE
3.5: Application Tool 2: SPMD Parallelization

Figure 1.1: *Compared structures of chapters 3 and 4*

grams, the static data flow analyses, and the dependence graph. Each aspect is presented in a section of chapter 3, because it originates from this compilation/parallelization domain. In front of this, corresponding sections of chapter 4 show how we transpose these aspects into AD, and what benefits come from that. Both chapters terminate with a section devoted to the application tools we have developped. These tools show the interest of program analysis technology on real-world, large applications.

Maybe one doesn't need to read this report completely, according to one's main interests.

- If interested in concepts and algorithms for code analysis, transformation, and optimization, then look at chapter 2 and sections 3.2, 3.3, 4.3, 4.4, and 4.5. There are also very specific algorithms in the application sections 3.4 and 3.5.

- If interested in a description of automatic parallelization, look at section 3.3, followed by 3.4 for a focus on loop-level parallelisation or by 3.5 for a focus on coarse-grain SPMD parallelization.

- If interested in AD principles and the use of AD in Scientific Computing, then look at sections 4.1 and 4.2. Section 4.6 may also prove interesting.

- If interested in common points between parallelization and AD, then look at sections 2.2 and 4.3 together, or sections 3.2 and 4.4, or sections 3.3 and 4.5.

- If interested in what functionalities a program transformation tool may offer, look at sections 3.4, 3.5, and 4.6.

- If interested in AD research problems and their links with general static analysis of programs, look at sections 4.3 through 4.5, and to the summary of open problems in chapter 5.

Chapter 2

Program Static Analyses and Transformations

This chapter presents general remarks on program static analyses and transformations, which are common to parallelization, discussed in chapter 3, and to differentiation, discussed in chapter 4. We first define and discuss static analyses in section 2.1, then section 2.2 discusses the internal representation of programs that we think is best adapted. Section 2.3 focuses on data flow static analyses. Finally, section 2.4 focuses on classical practical difficulties arising in all data flow analyses of imperative programs, namely aliases, pointers, and arrays.

2.1 Static *vs.* Dynamic

An analysis or transformation on a program is *static*, as opposed to *dynamic*, when it doesn't know the actual run-time inputs nor the run-time behavior of the program. Therefore, the actual value of most variables is unknown, as are the results of the switches that determine the flow of control. Actually, since a program can usually run on a very large set of possible different inputs, it is natural to assume this input is arbitrary, and therefore perform static analyses. However, there exist dynamic analyses, such as execution traces, any statistic analysis of a particular run, profiling, etc. Detection of cache misses, blocking communications, processor idle time, are typical dynamic analysis, whose results are of great help to improve the program.

The frontier between static and dynamic analyses is not even clear-cut.

For example partial evaluation, that we consider a static analysis, uses knowledge of a part of the run-time input to specialize and simplify the program (*cf* section 2.1.4). Also, most static analyses need extra, dynamic information on the run-time values to give better results. This information may be provided by the user, e.g. with directives, or can be found by a previous dynamic analysis.

Basically, the art of static analysis is to extract information, not from the run-time arguments which are hidden, but from the structure of the program itself. This gives abstract information on the run-time values, which in turn is useful for program transformations.

2.1.1 Undecidability

There is a theoretical limit to the precision of static analyses, called *undecidability*. It states that is is impossible to design a program which, given any program as an argument, will determine whether this argument program terminates or not. There exist a variety of corollaries. For instance, one can't write a software tool that finds out if two programs are equivalent, giving the same results in response to the same inputs. A program can't even decide that any two expressions of the analyzed program always evaluate to the same result.

The consequence is that most static analyses that must answer to a "boolean" question on the given program will give one of the three answers "*yes*", "*no*", or "*I can't tell*". Obviously, if the answer is "*I can't tell*", the tool that asked this question must be prepared to take the good decision. This decision must be *conservative*, i.e. it must produce a transformed program or an answer which is correct, whatever the run-time arguments. In other words, the tool that asked this question and got the "*I can't tell*" answer must take no chances: it must continue with its work considering that the run-time answer can be "*yes*" or "*no*" (there is no other possibility at run-time).

Notice that undecidability is an absolute but very far barrier. In practice other barriers will arise much sooner than undecidability, which have little relation with it. For example comparison of expressions in array indexes can become very complex and tools generally give up when the expressions are not linear with respect to the loop index. Array indexes can also involve indirection arrays, i.e. other arrays with unknown values. Comparison would require an enumeration of all possible cases, which is of course out of reach practically. Also, deciding whether two pointers reference the same memory location would require to enumerate all possible flows of control, and this is also practically out of reach.

Undecidability should not discourage anyone from implementing analyzers: in a sense, it just means that a given analyzer can always be improved, asymptotically!

2.1.2 Dynamic arguments and values

Run-time values of variables are simple elements of well-known domains, such as character strings or real numbers. Static information on these values must represent all possible run-values, and therefore belong to more complex domains, known as *abstract domains.* It can be intervals for example, or any kind of enumeration. The abstract domain may be more complex if one wants to capture possible relations between different values. For example, one can take simplexes or polyhedra in multi-dimensional spaces to represent inequalities between run-time values. A theoretical framework is *Abstract Interpretation* [13], that considers a static analysis as a special execution of the program that computes not values, but elements of the chosen abstract domains. Choice of the abstract domain determines the precision of the static analysis, i.e. the moment when approximations will become necessary. It also determines the computational cost of the static analysis.

Suppose we choose the abstract domain of intervals for values that are real numbers at run-time. Then we need to redefine the standard operations so that they run on intervals, and return intervals too. This is already more expensive than basic operations on real numbers. Now if we choose the abstract domain of polyhedra, the precision of the analysis is better, because it can capture and propagate relative information such as $a \geq b$, but one can be frightened by the cost of the standard operations defined on polyhedra.

2.1.3 Dynamic control

A static analysis must consider what happens during each possible run. This implies that each piece of information found is attached to one or many possible runs. Therefore we need a way to represent these runs abstractly. Maybe the most general representation is *instruction instances* (*cf* [10]), an algebraic notation that captures the history of all successive program switches and loops that lead to one particular run-time execution of an instruction.

There is a natural order between instruction instances, that reflects the execution order. Thus, for each textual instruction of a program, a static analysis chooses a partition of all possible instances of this instruction, and for each subset in this partition, it holds a piece of information relevant to this particular case. When the static analysis studies a *relation* between different run-time imstructions, it may need two or more instruction instances for each piece of information. Consider for example data-dependence analysis, that looks for every pair of run-time instructions such that the first one must always be executed before the second one. Each data-dependence must be attached to a pair of instruction instances, one for each textual instruction. The cost of this can be enormous.

The notion of instruction instances is essential in the theory of static analyses. In particular, it allows for algebraic manipulations such as proofs of correctness of transformations based on these analyses. A long-term goal would be certified implementations of static analyses and transformations, based on a formal specification. However, for nowadays complex static analyses running on large applications, drastic simplifications are necessary. Most often, all possible instances of an instruction are considered at the same time. Only in carefully selected cases, the set of instruction instances is partitioned into a few subsets. This may happen for a crucial conditional instruction, or for the body of some subroutine. In the case of subroutines, this question is also know as *specialization vs. generalization*: should one generalize, i.e. analyze the subroutine only once, in a context which is the union of all possible instances of all calls to the subroutine, or should one specialize, i.e. analyze the subroutine many times, generally once for each call site, in a context which is the union of all possible instances of this call? The rule of thumb is to stick to generalization as much as possible, because a piece of code that is called many times is generally used in a uniform manner. Conversely, two really different operations are generally performed by distinct pieces of code. In general, we might say that "syntax gives hints". Exceptions to this rule often come from specific properties of the program, only known to the end-user. Therefore a good strategy for a static analysis tool would be to let the user force specialization when appropriate, from the user-interface.

This partition of all instruction instances into a small number of more generic instances has a strong impact on the way static analyses run. If each instruction instance is analyzed separately, then for this instance the previous instructions form a straight-line code, which is easier to analyze. But this is impractical because there are so many instruction instances, even often an infinity. At the other extreme, if all instances are analyzed jointly, static analysis of a conditional goes into each of the two branches, and the resulting abstract values at the end of the two branches must be merged to get the abstract values at the end of the conditional. Similar things happen for every syntactic construct. For loops, a fixpoint analysis is generally necessary (*cf* section 2.3.2). This merging probably looses a part of the information, but the good side is that the analysis can follow the nested syntactical structure of the program, and this decreases complexity even more (*cf* section 2.2.4).

For analyses that look for a relation between two different instruction instances, it is often better to generalize all instruction instances and only specify the *distance* between each pair of corresponding instances. This *distance* indicates difference between two instances, and an example is in section 3.3 about data dependences.

2.1.4 An example: Partial Evaluation

Partial Evaluation is a transformation technique that specializes a program for a fixed value of a part of its inputs. Suppose we formally split the inputs of program $\mathtt{P}$ into two parts x and y. Partial evaluation of $\mathtt{P}(x, y)$ for the fixed input $x = \mathtt{x}_0$ is a new program $\mathtt{P}_{\mathtt{x}_0}$ such that for all y,

$$\mathtt{P}_{\mathtt{x}_0}(y) = \mathtt{P}(\mathtt{x}_0, y)$$

Usually, one hopes that a good constraint propagation (which is a static analysis) will help make the specialized program $\mathtt{P}_{\mathtt{x}_0}$ smaller and quicker than $\mathtt{P}$.

The choice of the abstract domain for variables is crucial. For example, polyhedra allow us to capture and propagate an inequality between two variables. For variables with discrete values, a simple abstract domain is the sets of possible values. Constraints add a precious level of precision: when constraint propagation proves that some variable has only one possible value, then one may simplify the operations it is involved in. Later on in the program, when a conditional uses a test, say $a \geq b$, the abstract values of a and b may prove that the test is always true or always false, and specialization may just remove the test and the other branch or the conditional.

The question of when to specialize a subroutine for a given subset of its inputs is crucial. In Partial Evaluation, the answer is given by heuristics. Partial Evaluation, in essence, relies on *specialization*, but we saw that this may lead to combinatorial explosion. The heuristics limits *specialization* to cases when it is probably harmless. Notice that both specialization and generalization return correct programs, but one may be faster while the other may be smaller in code space. In imperative languages, each specialization of a procedure creates an additional procedure in the specialized program, which is therefore heavier. On the other hand, this additional procedure can be automatically simplified, and thus probably runs faster. Choice of a good heuristic really is at the heart of Partial Evaluation systems.

To illustrate the power of Partial Evaluation, here is a rather pretty example that we studied when we were developping a Partial Evaluation system for inference rules [26, 27]. Suppose `P` is a set of inference rules that specify that a given regular expression accepts a given string of atoms. Classically, `P` can be written using rewrite rules: when a prefix atom can be accepted by the regular expression, this expression rewrites into a new regular expression and the tail of the string must be accepted by the rewritten regular expression. If we denote

$$E \xrightarrow{a} E'$$

the predicate saying that regular expression E rewrites into E' after accepting atom a, the core of `P` consists of the following inference rules:

$$\frac{\mathbf{accepts}(E_1, \emptyset) \qquad E_1 \xrightarrow{a} E_1' \qquad E_2 \xrightarrow{a} E_2' \qquad \mathbf{and}(E_1', E_2, R_1) \qquad \mathbf{or}(R_1, E_2', R)}{E_1.E_2 \xrightarrow{a} R}$$

$$\frac{\neg\,\mathbf{accepts}(E_1, \emptyset) \qquad E_1 \xrightarrow{a} E_1' \qquad \mathbf{and}(E_1', E_2, R)}{E_1.E_2 \xrightarrow{a} R}$$

$$\frac{E_1 \xrightarrow{a} E_1' \qquad E_2 \xrightarrow{a} E_2' \qquad \mathbf{or}(E_1', E_2', R)}{E_1 + E_2 \xrightarrow{a} R}$$

$$\frac{E \xrightarrow{a} E' \qquad \mathbf{and}(E', E^*, R)}{E^* \xrightarrow{a} R}$$

$$\frac{}{\mathbf{atom}\ a \xrightarrow{a} \mathbf{empty}} \qquad\qquad \frac{a' \neq a}{\mathbf{atom}\ a' \xrightarrow{a} \mathbf{stop}}$$

where $\mathbf{accepts}(E, \emptyset)$ checks that regular expression E may accept the empty string, **and** and **or** respectively rebuild a normal form of the concatenation "." and the alternative "+" of their two argument regular expressions, and **stop** denotes that the the regular expression failed to accept the given prefix.

We can do partial evaluation on this program `P`, fixing the given regular expression E, for example $E = 6.(2.3+4^*)^*.5$, and letting the string vary. Obviously what we get is a specialized program that checks that a given string matches $6.(2.3+4^*)^*.5$. Actually we need to perform partial evaluation on the solving process itself, i.e. the program that combines inference rules to get the answer. Therefore the abstract domain we choose is the set of *proof trees*, that are built by logical combination of inference rules. We also introduce in the abstract domain constraints that hold information on the variables that appear in the proof trees. Specialization versus generalization is decided by a heuristic that accepts specialization only when the number of possible proof trees doesn't grow too much and no proof tree becomes too large. Each specialization of the $\rightarrow$ predicate to a given "state" of the regular expression will be denoted here as $\mathbf{S}_n \rightarrow$, where $\mathbf{S}_n$ is a fixed name. What we end up with is the following set of specialized inference rules:

$$\frac{}{\mathbf{S}_0 \xrightarrow{\emptyset} \textit{“reject”}} \qquad \frac{\mathbf{S}_1 \xrightarrow{i} r}{\mathbf{S}_0 \xrightarrow{6\bullet i} r}$$

$$\frac{}{\mathbf{S}_1 \xrightarrow{\emptyset} \textit{“reject”}} \qquad \frac{\mathbf{S}_2 \xrightarrow{i} r}{\mathbf{S}_1 \xrightarrow{2\bullet i} r} \qquad \frac{\mathbf{S}_3 \xrightarrow{i} r}{\mathbf{S}_1 \xrightarrow{4\bullet i} r} \qquad \frac{}{\mathbf{S}_1 \xrightarrow{5\bullet i} \textit{“accept”}}$$

$$\frac{}{\mathbf{S}_2 \xrightarrow{\emptyset} \textit{“reject”}} \qquad \frac{\mathbf{S}_1 \xrightarrow{i} r}{\mathbf{S}_2 \xrightarrow{3\bullet i} r}$$

$$\frac{}{\mathbf{S}_3 \xrightarrow{\emptyset} \textit{“reject”}} \qquad \frac{\mathbf{S}_3 \xrightarrow{i} r}{\mathbf{S}_3 \xrightarrow{4\bullet i} r} \qquad \frac{\mathbf{S}_2 \xrightarrow{i} r}{\mathbf{S}_3 \xrightarrow{2\bullet i} r} \qquad \frac{}{\mathbf{S}_3 \xrightarrow{5\bullet i} \textit{“accept”}}$$

which implements the finite state automaton shown on figure 2.1. Notice that this finite state automaton is not minimal, because this Partial Evaluation algorithm could not find that the two regular expressions

$$4^*.(2.3+4^*)^*.5 \qquad \text{and} \qquad (2.3+4^*)^*.5$$

are equivalent. Therefore nodes $\mathbf{S}_1$ and $\mathbf{S}_3$ are not merged. Yet partial evaluation actually compiled the regular expression into a finite automaton. This classical operation is usually done by ad-hoc algorithms, such as the "first and follow" algorithm [5]. Here it is a mere consequence of the basic principles of Partial Evaluation!

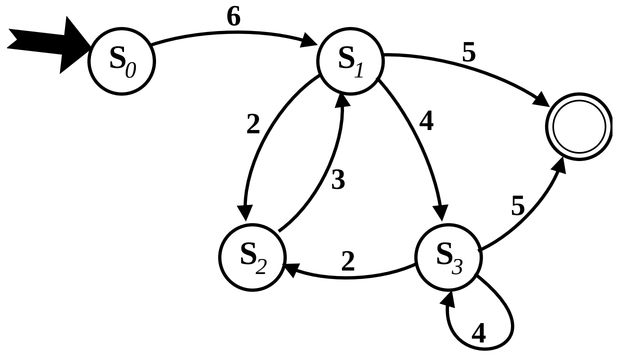


Figure 2.1: *Specialized finite automaton built by Partial Evaluation*

2.2 Internal Representation of Programs

The internal representation of programs is essential for good static analyses. It must be easy to follow each thread of control. To this end, we advocate Call Graphs and Structured Flow Graphs rather than plain Abstract Syntax Trees. Information on variables, which is required very often, should be available quickly. It is not surprising that the answer to this is a Symbol Table. Also, anticipating on section 2.2.4, a structured representation helps to keep complexity reasonably low.

2.2.1 Symbol Tables and Memory

In the program's statements, variables are identified by their name, which is a string. From this name, we must get the type, the size, and many other pieces of information. This is classically done with a *Symbol Table* which, given a key (a string) returns whatever is stored for this key. Symbol Tables use *hash coding* for a quicker retrieval of the key.

Symbol Tables must provide nesting to implement *scoping*. When a new scope is opened, e.g. when a local variable is declared, a new Symbol Table is built on top of the existing Symbol Table. It can see all variables in deeper Symbol Tables, but new declarations at this level will hide deeper declarations. When this scope ends, the initial Symbol Table is visible again.

Nesting Symbol Tables is somewhat more complex for Object-Oriented languages. When a class `C` inherits from a parent class `P`, the Symbol Table of `C` is built on top of the `P`. Depending on access properties, `C` is sometimes built only on top of the *public* Symbol Table of `P`. If `C` inherits from several parent classes, then its Symbol Table is built on a sort of union of its parents' Symbol Tables. The langugage standard specifies the order in which a string is searched in the parents' Symbol Tables.

All this is valid for properties of variables that do not change at runtime. For a property that evolves, such as "is the variable initialized or not", it must be stored in distinct structures, probably one for each program's instruction. Instead of creating a new hash table mechanism, it is a good practice to simply create arrays, with one entry per variable. The index allocated to represent a variable is a fixed property of this variable, and thus can be stored in the Symbol Table.

This correspondence from variables to indices forms a map of a sort of virtual memory. We call the indices *zones*. At one extreme, one may choose to attach only one zone to each array variable. However there are cases where it is best to create several zones for one array. For example it may happen that two arrays partly overlap in memory, due to *aliasing* or in FORTRAN using the `EQUIVALENCE` declaration. This is captured esily by creating different zones for the different overlapping array regions (*cf* figure 2.2). At the other

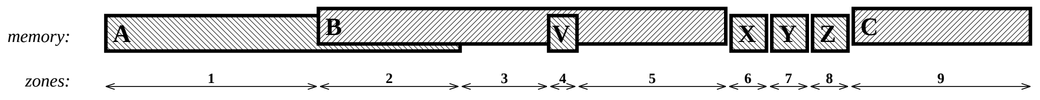


Figure 2.2: *Creating different memory zones captures memory overlapping*

extreme, one may wish to assign one different zone to each array cell. This is not always possible, though, because some arrays have a dynamic size, and therefore one can't tell how many cells they will need. Moreover it is probably a poor strategy, especially if accesses to the array are uniform. Here also, "syntax gives hints!". If an array is always referenced with a generic loop index, such as `A(i)` or even `A(f(i))`, it is most likely that these references are uniform and all array cells are somewhat equivalent. One zone is enough for `A`. On the other hand, if the program refers explicitly to `A(1)` or `A(37)`, the corresponding memory cells may be best analyzed individually, like ordinary scalar variables.

2.2.2 Flow Graphs

For the record, *Abstract syntax Trees* (AST's) are the best way to see through the *concrete syntax*, getting rid once and for all of tedious tasks such as counting white spaces or checking precedence in expressions. The nodes of the syntax tree are operators that represent constructs of the programming language, such as arithmetic or logic operations, assignment, array indexing, or structured control. The leaves of the syntax tree are constant litterals or references to the name of a variable. Syntax trees are nearly everywhere: for readability, in the remainder of this report we shall often display syntax trees as pieces of source program text instead of a "real" tree. This is when there is no ambiguity. For example on figure 2.3, the individual instructions in the Flow Graph are indeed syntax trees shown as natural text.

Atomic instructions are best represented as AST's Structured instructions, such as loops and conditionals can be either kept as AST's or as Flow Graph's. In either case, it is easy to follow the thread of control. Then comes the problem of non-structured control (e.g. `GOTO`'s). Let's face it, real programs sometimes use constructs that are not absolutely clean. When a language offers such instructions, we must be prepared to analyze them. Many academic research simply considers small languages, where non-structured control is forbidden. We can't afford to do that, because we want to develop tools that apply to real programs. For non-structured control, the only representation that captures the thread of control is the Flow Graph.

The advantage of Flow Graphs is that they handle any kind of control in a uniform manner. Analyses based on Flow Graphs are therefore more general. A Flow Graph is a directed graph, whose nodes are *basic blocks*, i.e. sequences of instructions. Precisely, a basic block contains a list of instructions which are always executed in sequence, from first to last, with the same Symbol Table context. Thus the current Symbol Table is attached to the basic Block. Arrows in the Flow Graph represent the flow of control, i.e. the possible destinations of the execution pointer after completion of a basic block. At run-time, a test located at the end of the basic block decides on the actual next basic block to execute. Figure 2.3 shows the Flow Graph of a small procedure. Notice the special basic block at the beginning of the procedure, called the *Entry Block*. Similarly, the *Exit Block* represents the end and return from the procedure. There are some other special basic blocks that hold loop headers: this is a simplified way to introduce syntactic structure back into the flow graph. A more general way, *Structured Flow*

```
          SUBROUTINE small(A,B)
          INTEGER A,B,i,n
          DIMENSION A(100), B(100)
C
          n = 0
          DO 100 i=1,100
            IF (A(i).ge.0) THEN
              A(i) = n
              n = n + 1
            ELSE
              A(i) = B(i)
              GOTO 200
              n = n + 3
            END IF
            B(i) = 0
 100      CONTINUE
 200      print *,n
          end
```

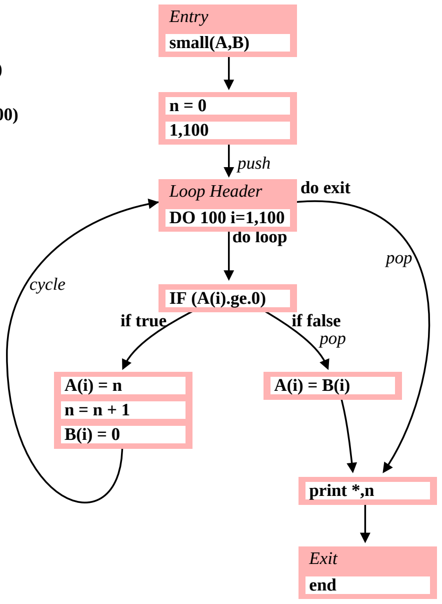


Figure 2.3: *The Flow Graph of a small procedure*

Graphs, is described below. Notice also that the arrows must hold some extra information in the following cases:

- when several arrows leave from a basic block, they must indicate which arrow is taken in which case at run time. For example the arrow labelled **do exit** is chosen when the iteration control decides that the loop is terminated.

- when an arrow doesn't remain in the same loop syntactic structure, this is denoted by a *push* when it jumps into a loop, a *pop* when it jumps out of the loop, or a *cycle* when it just goes to the loop's next iteration.

Generally, two special Symbol Tables are built for a procedure:

- The *public* Symbol Table holds all symbols that can be seen by the outside context. This includes the formal parameters, most other procedures, and all sorts of global variables.

- The *private* Symbol Table holds all local symbols, whose scope ends when the procedure returns. This includes local variables and procedures.

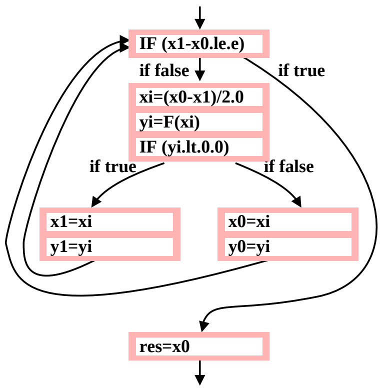
Of course the private Symbol Table is built on top of the public one, because the procedure instructions can see both private and pubic symbols, whereas the calling context can only access to the public symbols.

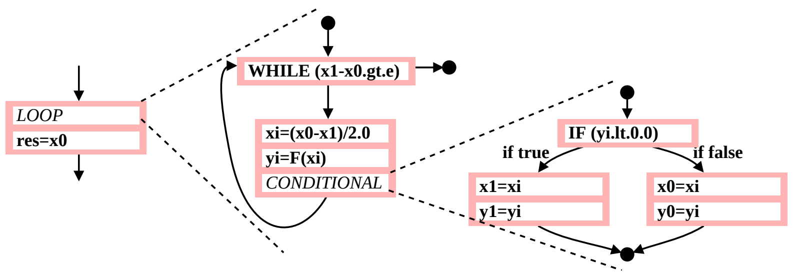
Flow Graphs have a bad reputation in academic research, because they alledgedly loose the control structure, and because one cannot keep nice algebraic notations such as instructions instances [10]. It is true that structured programs are preferable to "spaghetti code", and plain Flow Graphs do not distinguish a well-structured program from a "spaghetti" program! However, we find this judgement unfair. Flow Graphs can actually capture structure: not surprisingly, these are known as "*Structured Flow Graphs*"! Structured Flow Graphs [2, section 10.10] represent procedures as a Flow Graph whose instructions can be atomic instructions, or recursively Structured Flow Graphs. The "dirty" effect of a `GOTO`'s is therefore limited to the inside of its enclosing Structured Flow Graph. It is well known [4] how to transform any Flow Graph into an optimal Structured Flow Graph, which is the good base to then regenerate a cleaner source program. This is done in particular by *restructuring* tools. We end up with this general scheme, whatever the structuration of the initial procedures:

1. build the Flow Graph, therefore temporarily loosing existing structure, then
2. using "dominator" analysis, (re)build a structure on Blocks, and
3. run static analyses recursively on this structured Flow Graph;
4. if regenerating a procedure from the Flow Graph, use this structure to build an equivalent structured code.

Figure 2.4 shows an example of restructuration based on Structured Flow Graph. The input on the left is some kind of old-style FORTRAN program. The result on the right meets the newest standards. The internal representation (bottom left) is style-independent and static analyses and transformations apply to it easily.

```
100  IF (x1-x0.le.e) GOTO 300
     xi=(x0-x1)/2.0
     yi=F(xi)
     IF (yi.lt.0.0) GOTO 200
     x0=xi
     y0=yi
     GOTO 100
200  x1=xi
     y1=yi
     GOTO 100
     res=0.0
300  res=x0
```





```
DO WHILE (x1-x0.gt.e)
   xi=(x0-x1)/2.0
   yi=F(xi)
   IF (yi.lt.0.0) THEN
      x1=xi
      y1=yi
   ELSE
      x0=xi
      y0=yi
   ENDIF
ENDDO
res=x0
```

Figure 2.4: *Restructuration using a Structured Flow Graph*

2.2.3 Call Graphs

The Call Graph is the topmost level of program representation. The Call Graph is a directed and possibly cyclic graph. Each node of the Call Graph represents one procedure, and there is an arc from procedure `P1` to procedure `P2` if `P1` may make a call to `P2`. A cycle in the Call Graph means that the program is recursive. Since a node represents a proedure, it indeed contains a Flow Graph, except in special situations, like for external, library, or intrinsic procedures, whose source code is not given. Arrows in the Call Graph can be built only after the static type-checking phase, because many languages use the types to determine which procedure is called. This happens with *overloading*, or in object-oriented programs.

The Call Graph is a static representation for static analysis. In particular, an arrow from procedure `P1` to procedure `P2` only implies that `P1` may call `P2`, but it may also *not* call `P2` in some contexts. Conversely there is only one node for `P2`, even if it may be called by many different procedures in many different contexts. This corresponds to our choice to stick to *generalization* as much as possible: when `P2` can be called in many contexts, first merge these into a single general context, and analyze `P2` only once in this general context. As we said earlier, this choice of *generalization* leads to some loss of precision during static analyses, but in general the loss is very limited.

Static analyses on Call Graphs fall into two principal categories: *top-down* and *bottom-up*.

- **top-down:** Assuming that there is a `TOP` procedure (e.g. the `main` procedure), a *top-down* analysis propagates information initially known for `TOP`, down into all procedures possibly called under `TOP`. The generalization principle tells us that a given procedure `P` can be analyzed only when all its calling contexts are known, and therefore only when all procedures that may call `P` have been analyzed. Thus, procedures must be analyzed in a well-chosen order. When the Call Graph is acyclic, this order is the topological sorting of nodes below `TOP`. When the Call Graph is cyclic, there is no such order, and the analysis must be run repeatedly until fixpoint. But the *Depth-First Spanning Tree* algorithm [2, section 10.9] gives a very good order that keeps the number of iterations low.

- **bottom-up:** Starting from the bottom nodes of the Call Graph, i.e. procedures that call no other procedures, a *bottom-up* analysis synthesizes properties of procedures, independent of their particular calling context. This process eventually stops at the `TOP` procedure. A given procedure `P` can be analyzed only when all the procedures it calls have been analyzed. Again, recursive programs that have a cyclic Call Graph require an iterative process. In contrast with *top-down* analysis, there is no need to remember and accumulate calling contexts. Symmetrically, after the analysis of each procedure `P`, a summary of the synthesized information for `P` must be built and stored, to be used by the callers of `P`.

Let us mention an extra difficulty that arises in object-oriented programs, or when run-time overloading is used. On a given `CALL` to a procedure, the actually called procedure may be unknown at static analysis time. It only ranges in a – hopefully finite! – set of procedures. When running a *bottom-up* static analysis, one cannot just read the analysis result for the called subroutine. Instead, one must take the results for each procedure possibly called, and build a conservative union of these results. This is yet another cause for imprecision. This is one of the reasons why most static transformations of programs, and especially Automatic Differentiation, still focus on "classical" languages and delay analysis of object-oriented languages, which is still an open problem.

2.2.4 Complexity issues

This hierarchical representation (*Instructions*, *Flow Graphs* possibly with structured Blocks, *Call Graph*) allows us to control combinatorial explosion during analyses.

Careless implementation of a program static analysis would probably start from the first line of the main subroutine, and then follow the "execution pointer". If a procedure `S` is called many times in the program, it would be analysed many times, once for each call site. Moreover, procedures called by `S` would also be analyzed many times with `S`. Most likely, information on variables carried along analysis would concern all variables of calling procedures. This leads to unacceptably inefficient analyzers, and we need a better strategy, based on the nested levels of our hierarchical representation of section 2.2..

In general one can identify a computationnally expensive part of the computed information, which can be synthesized. This means it is computed bottom-up, starting on the lowest (and finest) levels of the program representation, and then recursively combined at the upper (and coarsest) levels. Consequently, this synthesized information must be made independent from the context, i.e. the rest of the program. When the synthesized information is built, it is used in a final pass, essentially top-down and context dependent, that propagates information from the "ends" of the program (the main procedure's beginning and end), to each particular procedure, basic Block, or instruction. We refer the interested reader to the classics of Compiler Theory, and in particular [2].

At the Call Graph level, bottom-up analyses build a context-independent signature of each procedure, which is then used during analysis of each `call` instruction of a calling procedure. Conversely, top-down analyses on a `call` instruction of a calling procedure accumulate their calling context into the general calling context of the called procedure. This general context is then retrieved when in turn analyzing the called procedure. Recursive programs require an extra fixpoint at this level in the algorithm.

At the Flow Graph level, analyses most often require a fixpoint because Flow Graphs are seldom acyclic. The number of iterations required to reach the fixpoint is generally about the number of loops in the Flow Graph, provided each iteration visits the basic block in a good enough order, usually found by the *Depth-First Spanning Tree* algorithm. Yet this fixpoint mechanism can be expensive. Let us present two approaches to reduce its cost:

- **Basic blocks:** iterations repeatedly call for propagation of a piece of information through a basic block. Instead of doing this by sequential propagation through each successive instruction, one can build a summary of the "effect" of the entiere basic block on the computed information, and apply this effect each time propagation across the basic block is required.

- **Structured Flow Graphs:** similarly, if the Flow Graph is structured, one structured sub-flow-graph is just seen as one instructution at the level above. One can compute the effect of this structured instruction once, and then use it during propagation of computed information. Moreover, when the structured sub-flow-graph is a loop, the fixpoint mechanism is limited to this sub-flow-graph, and no fixpoint is needed at the other levels.

Examples in sections 2.3.3 and 4.4 illustrate these techniques.

2.3 Data Flow Analyses

Apart from rare exceptions, static analyses needed by program transformations find information on variables, on their run-time value, and on their relationship. These are called *data flow analyses.* It is important to specify these analyses formally, to make sure that they terminate, of course, and also to implement them with algorithms of smallest possible complexity.

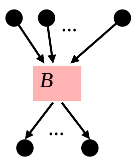
2.3.1 Set-based specification of Data Flow Analyses

At the Flow Graph level, each Data Flow analysis is described concisely by so-called *Data Flow equations.* In their most general form, these equations apply to *unstructured* Flow Graphs, because real programs have unstructured Flow Graphs in general. On the other hand, these general equations can be specialized to *structured* Flow Graphs, i.e. cleanly nested loops and conditionals, yielding *structured* Data Flow equations. The latter may be applicable to a smaller class of programs, but are usually more efficient, and also more illustrative.

Assume that the information computed by the data flow analysis can be represented as a set of variables. This can be, for example, the set of all initialized variables at a given location in the program. Data Flow equations are constraints that relate these sets *before* and *after* each individual instruction. Because of loops, these constraints form in general a system that defines the sets *implicitly.* In other words the system must be solved, e.g. by a fixpoint approach, to obtain the desired sets. The constraints themselves express the nature of the data flow information seached. The fixpoint propagation mechanism is just a mechanical step, independent of the nature of the data flow problem. Therefore data flow equations completely specify the data flow analysis.

Consider for example the problem of detecting variables used before initialized. The set S we need to propagate is in fact the set IV of initialized variables. On a general, unstructured Flow Graph, it needs only two Data Flow equations, shown on figure 2.5. They just state that $OutIV$, the IV set of initialized variables just after a basic block B, is equal to $InIV$, the IV set just before B, which in turn is the union of the $OutIV$ sets of all incoming basic blocks, augmented with $Written(B)$, the set of all variables that are overwritten by the basic block itself.

In the case of structured flow graphs, data flow equations can be special-

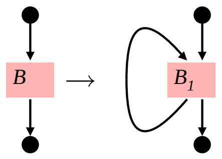


$$InIV(B) = \bigcup_{P \text{ predecessor of } B} OutIV(P)$$

$$OutIV(B) = InIV(B) \cup Written(B)$$

Figure 2.5: *General data flow equations for initialized variables*

ized for well-known structures, giving more efficient rules. Considering again the variable initialization problem, the general rules of figure 2.5 give the specialized rules for a loop structure shown on figure 2.6. Notice that since repeated unions with a constant set don't change the result, the fixpoint could be solved statically and therefore the structured data flow equations are more efficient than the unstructured.



$$InIV(B_1) = InIV(B) \cup Written(B_1)$$

$$OutIV(B) = InIV(B_1)$$

Figure 2.6: *Structured data flow equations for initialized variables, in the loop case*

Assuming again that the analysis propagates sets of variables, it is advisable not to represent these sets as a collection of names, i.e. of character strings, because comparing strings is expensive. Since the space of all names is known a priori, t is far better to assign a unique integer index to each possible name, as discussed in section 2.2.1. The sets are thus represented as "bitsets", on which the classical set operations can be implemented very efficiently.

2.3.2 Termination issues

It is necessary to make sure that the fixpoint resolution eventually terminates. Abstract interpretation gives the general framework for this. When the analysis is defined by set-based data flow equations, we can give a helpful lemma. Suppose we compute set "*Set*". The iterative resolution algorithm starts with an initial state $(InSet_0, OutSet_0)$, and iteratively computes new states $(InSet_i, OutSet_i)$, for increasing i's. Each state is built of two mappings from each block B to $\mathit{InSet}_i(B)$ the value of *Set* before B, and to $\mathit{OutSet}_i(B)$ the value of *Set* after B. Each iteration i applies the data flow equations on each block B (excluding the entry and exit blocks) to build $\mathit{InSet}_i(B)$ and $\mathit{OutSet}_i(B)$ using the previous state (InSet_{i-1}, OutSet_{i-1}). Then we have:

Lemma : *Iterative resolution of data flow equations for set Set reaches a fix-point in a finite number of iterations if the set of possible values of Set is finite and* $\forall i > 0, \forall B, \mathit{InSet}_{i-1}(B) \subseteq \mathit{InSet}_i(B)$ *and* $\mathit{OutSet}_{i-1}(B) \subseteq \mathit{OutSet}_i(B)$.

Proof :
Since there is only a finite number of possible values of *Set*, there is a finite number of possible states. Set inclusion induces a partial order $\subseteq$ on states and the hypothesis just says that $(\mathit{InSet}_{i-1}, \mathit{OutSet}_{i-1}) \subseteq (\mathit{InSet}_i, \mathit{OutSet}_i)$ for each iteration $i > 0$. If the iterative resolution does not reach a fixed point, all the inclusions are strict, and thus the successive states form an infinite set of states. This is impossible since there is only a finite number of different states. □

There are cases where the set of possible values is infinite. For example when the data-flow information consists of intervals of possible values of variables. Then the analysis must provide a mechanism to prevent infinite sets of successive states. One method is to allow each interval bound to grow only a fixed number of times, and then the bound is pushed to infinity. This classical method is called *widening*.

2.3.3 An example: Read-Written data flow analysis

The Read-Written data flow analysis finds, for each procedure, and for each argument of this procedure, what the effect of the procedure on the argument is. This can be one of:

- *"rw":* the initial argument value is read and later it is overwritten
- *"r":* the initial argument value is read and not iverwritten
- *"w":* the initial argument value is not read, but it is overwritten
- *"n":* the argument value is not read nor overwritten

This analysis is bottom-up on the Call Graph.

Essentially because of arrays, things are not so simple. We choose to consider any array as an atomic variable. Thus a variable, if it is an array, can have a part which is read (*resp.* written), and another part which is not. Also, since the control is not known, a variable can be read (*resp.* writen) for one control, and not for another. So, at a given location in the program, we represent the Read-Written information as a collection of four sets RW, R, W, and N. The set RW (*resp.* R, W, N) is the set of all variables for which there exists a possible flow of control from the procedure entry to the current location along which some part of the variable may be **initially read and later overwritten** (*resp.* **only read**, **only overwritten**, **not accessed**). Now, a given variable can belong to one or many of these sets! Naturally, the sets are computed following the direction of the Flow Graph arrows, because all sets have a simple initial value on the entry block, whereas the values of the sets on the exit block is exactly the desired result. At any location, the sets that we would naturally use, "*Read*" and "*Written*", are nothing but shortcuts defined as

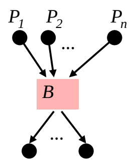
$$Read = R \cup RW$$

$$Written = W \cup RW$$

Notice also that the set of all variables whose input value is completely overwritten by a piece of code B, known as the *Kill* set, can be defined as:

$$Kill = Written \setminus (R \cup N)$$

Figure 2.7 gives the general data flow equations that specify Read-Written analysis. The first four equations are straightforward: they say that there exists a flow of control from the procedure entry to B's entry along which a variable is *"rw"*, *"r"*, *"w"*, or *"n"* if there exists at least one such flow of control from the procedure entry to the exit of some predecessor block P_i. The last four equations combine the incoming sets with the sets $RW(B)$, $R(B)$, $W(B)$, and $N(B)$, which hold local, context-independent information about the block B. For example a variable is in $RW(B)$ if its value when entering B may have been at least partly read and overwritten when reaching the exit of B. Let us read the equation that gives $OutR(B)$: a variable is "only read" from the procedure entry to the exit of B if it is only read till the entry of B and then it is only read or not accessed inside B, or else if it is not accessed till the entry of B and only read inside B. The next equation, that gives $OutW(B)$ says that a variable is "only overwritten" from the procedure



$$InRW(B) = \bigcup_i InRW(P_i)$$
$$InR(B) = \bigcup_i InR(P_i)$$
$$InW(B) = \bigcup_i InW(P_i)$$
$$InN(B) = \bigcup_i InN(P_i)$$

$$OutRW(B) = InRW(B) \cup (InN(B) \cap RW(B)) \cup (InR(B) \cap (RW(B) \cup W(B)))$$

$$OutR(B) = (InR(B) \cap (R(B) \cup N(B))) \cup (inN(B) \cap R(B))$$

$$OutW(B) = InW(B) \cup (InN(B) \cap W(B))$$

$$OutN(B) = InN(B) \cap N(B)$$

Figure 2.7: *General data flow equations for Read-Written analysis*

entry to the exit of B if it is "only overwritten" on some control path till the entry of B (and what happens in B doesn't matter because the initial value is lost anyway), or else if it is not accessed till the entry of B and only overwritten inside B.

Naturally, in the general case where the Flow Graph is cyclic, resolution of the data flow equations is iterative. The initial situation sets every *In* and *Out* set to $\emptyset$, except for the *OutN* set of the entry block, which must contain all variables in the current scope. This represents the initial step of any execution of the procedure: every variable is brand new, neither read nor written yet. On the other hand when iterative resolution reaches the fixed point, the exit block contains the desired result, whihc is the effect of the procedure on each of its arguments.

It is very easy to check that after each iterative sweep on the flow graph, all sets strictly grow. Moreover, the sets can have a finite number of values. Using the above lemma, this ensures that the iterative process terminates.

Let us show now how structured flow graph can improve efficiency. Consider for instance a structured loop (figure 2.8), and let's see how the general data flow equations can be specialized to this situation. Similar specialization work for conditionals and other structured patterns. The fixed point can be solved statically and the specialized equations need no fixpoint any more. Consider first the equations for the *InN* and *OutN* sets. The general data flow equations tell us that

$$\mathit{InN}(B_1) = \mathit{InN}(B) \cup \mathit{OutN}(B_1)$$

and

$$\mathit{OutN}(B_1) = \mathit{InN}(B_1) \cap N(B_1)$$

from which we easily find the fixpoint equation:

$$\mathit{InN}(B_1) = \mathit{InN}(B) \cup (\mathit{InN}(B_1) \cap N(B_1))$$

The set $\mathit{InN}(B)$ is fixed here, and the set $N(B_1)$ is constant during all the resolution. $\mathit{InN}(B_1)$ is initialized to $\emptyset$, and therefore the fixed point solution is

$$\mathit{InN}(B_1) = \mathit{InN}(B)$$

Let's consider now the *InW* and *OutW* sets. The general data flow equations give:

$$\mathit{InW}(B_1) = \mathit{InW}(B) \cup \mathit{OutW}(B_1)$$

and

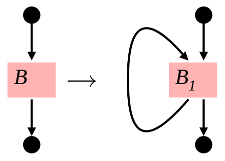
$$\mathit{OutW}(B_1) = \mathit{InW}(B_1) \cup (\mathit{InN}(B_1) \cap W(B_1))$$

Again $\mathit{InW}(B)$, $W(B_1)$ are constants, and we can use the value of $\mathit{InN}(B_1)$ we found above. Therefore the fixed point is obvious:

$$\mathit{OutW}(B_1) = \mathit{InW}(B) \cup (\mathit{InN}(B) \cap W(B_1))$$

Similar reasoning leads to the specialized rules on figure 2.8, and one can check that no more fixed point resolution is needed, and thus resolution is faster.

To terminate this section, let us show how the use of bitsets can simplify the whole implementation. Obviously, all sets *InRW*, *OutRW*, *RW*, etc can be represented as bitsets, as mentioned in section 2.3.1. Classical set operations $\cup$ and $\cap$ are implemented as the classical bitwise-or and bitwise-and operations of C or JAVA. We can even use this to ease the technical task of translating the Read-Written signature of a procedure `P`, from the local name space of `P` to the name space of the calling procedure `CP`. This is of course essential to the bottom-up Read-Written analysis on the Call Graph.



$$InRW(B_1) = InRW(B) \cup (InR(B) \cap (RW(B_1) \cup W(B_1))) \cup (InN(B) \cap (RW(B_1) \cup (R(B_1) \cap W(B_1))))$$

$$InR(B_1) = InR(B) \cup (InN(B) \cap R(B_1))$$

$$InW(B_1) = InW(B) \cup (InN(B) \cap W(B_1))$$

$$InN(B_1) = InN(B)$$

$$OutRW(B) = OutRW(B_1) = InRW(B_1)$$

$$OutR(B) = OutR(B_1) = (InR(B) \cap (R(B_1) \cup N(B_1))) \cup (inN(B) \cap R(B_1))$$

$$OutW(B) = OutW(B_1) = InW(B_1)$$

$$OutN(B) = OutN(B_1) = InN(B) \cap N(B_1)$$

Figure 2.8: *Structured loop data flow equations for Read-Written analysis*

The Read-Written signature of `P` is written in `P`'s local name space. There is a convention that assigns each variable of `P` to one "zone", i.e. to one index in the bitsets. In the calling subroutine `CP`, actual parameters have different names than formal parameters of `P`, and therefore the Read-Written signature $S_{\texttt{P}}$ of `P` must be translated into $S_{\texttt{CP}}$, so that it uses the zone number conventions of `CP`. This is done by multiplying $S_{\texttt{P}}$ by a fixed matrix of booleans, which is a constant characteristic of this particular call site, and which we call the *translation matrix*. For example, suppose `P` actually has three formal parameters, plus two global parameters in a `COMMON` block `/C1/`. Suppose they have received zone numbers 1 to 3 and 4 to 5 respectively. On the calling site, `P` is called with actual parameters that correspond to `CP` zones 4, 2 and 5 respectively, and the `COMMON` block `/C1/` was split differently, as a single array whose zone is numbered 6. The translation matrix of this call to P inside CP is thus:

$$T_{\mathtt{CP}\leftarrow\mathtt{P}} = \begin{pmatrix} . & . & . & . & . \\ . & 1 & . & . & . \\ . & . & . & . & . \\ 1 & . & . & . & . \\ . & . & 1 & . & . \\ . & . & . & 1 & 1 \end{pmatrix}$$

Suppose that the $R_{\mathtt{P}}$ effect of P on its arguments is that arguments 1 and 2 and the second variable in the common are possibly partly read (bitvector (. 1 1 . 1)), then the effect $R_{\mathtt{CP}}$ of this call of P inside CP is obtained simply by:

$$R_{\mathtt{CP}} = T_{\mathtt{CP}\leftarrow\mathtt{P}}.R_{\mathtt{P}} = \begin{pmatrix} . & . & . & . & . \\ . & 1 & . & . & . \\ . & . & . & . & . \\ 1 & . & . & . & . \\ . & . & 1 & . & . \\ . & . & . & 1 & 1 \end{pmatrix} \times \begin{pmatrix} . \\ 1 \\ 1 \\ . \\ 1 \end{pmatrix} = \begin{pmatrix} . \\ 1 \\ . \\ . \\ 1 \\ 1 \end{pmatrix}$$

Which can be implemented in a few machine operations.

2.4 Some classical, practical, difficulties

2.4.1 Arrays

The value of an array index generally isn't known at run time, except when it is an immediate constant. In general, it is an expression, often depending on the index of surrounding loops, and involving other variables. The question whether one array reference at one instruction instance is identical to another reference at another instruction instance is probably undecidable fro one thing, and in practice problems arise much before that: for example, the array indices may depend on a variable whose value we didn't care to analyze. In other words, we cannot hope to manipulate arrays as if they were just a collection of isolated scalar variables.

Therefore most static analysis will have to make conservative approximations for arrays. This problem has received a lot of attention, for instance in [15, 14, 44]. The key to these approximations is to choose which "regions" of the arrays the analyzer will be able to handle.

- One extreme is to consider all cells of the array are equivalent. In other words, one never knows for sure if one reference to an array overlaps in memory or not, with another reference to the same array. All information on one array cell is immediately diluted to all array cells, leading to a great deal of “blurring”. Approximations are thus very conservative, and very little can be known for sure on the array. On the other hand the analysis requires very little space.

- At the other extreme, one could consider each array cell as a separate scalar variable, for which information must be built individually. This would be extremely expensive in memory space during analysis, but no approximation would be necessary. However, this extreme is impossible, because array dimensions are often unknown at static analysis time, and it is not always possible to compute the value of array indices at any instruction instance.

In practice, we look for a compromise between these extremes. This is a matter of taste, and depends on the way the arrays are referenced to by the program. Again, “syntax gives hints!”. Existing accesses to arrays follow a certain pattern, and the array regions must be able to capture this pattern. Conversely when regenerating pieces of program, the chosen array regions must be easily manipulated by the program’s language.

Here are some popular choices: Consider a two-dimensional array `A`: When indices of `A` are linear functions of surrounding loop counters, this indicates that array regions could be regular patterns in the array. When arrays are used in loops whose bounds depend on surrounding loop indexes, then the array regions could be (convex) polyhedra (or *“polytopes”*) that define “parts” of `A`’s index space. This can be a rectangle, or a triangle... Researchers also think of a combination of a regular pattern somehow “clipped” by a polytope, or of a given polytope regularly repeated following a pattern.

One advantage of the above regions is that intersection needs no approximation: the intersection of two regular patterns is a regular pattern, and the intersection on two convex polyhedra is a convex polyhedron. Unfortunately, some static analyses occasionaly require the union of two such regions, and this is not defined exactly. For example the union of two convex polyhedra is not convex any more. Usually, the conservative approximation requires to take the convex envelope.

Let us mention last the compromise that is made in PARTITA and TAPENADE: we only capture array overlap, due to `EQUIVALENCE`, `COMMON` blocks or array parameter passing (*cf* figure 2.2). In practice, most arrays are considered atomic, with only one region, except for overlapping arrays which may have two or three. We also investigated creating special regions for array cells accessed explicitly with immediate indices, like in `A(2) = x*A(1)`. In our opinion, the partial failure of `HPF-fortran`, which was specialized for systematic regular access to arrays, shows that real applications have far more complex array indexes, that cannot be captured by the array regions above. Since this was not our main research focus, we remained with very coarse-grain regions that only distinguish overlapping arrays.

2.4.2 Aliases

Suppose a subroutine has many arguments (i.e. formal parameters, globals, etc). At each call site, each of these formal parameters is given an actual parameter. Aliasing happens when the same actual parameter is given to two different formal parameters, and one of these two formal parameters is overwritten by the subroutine. More generally, aliasing happens when two actual parameters overlap in memory, and at least one may be overwritten. Note that this aliasing does not conform to the FORTRAN standard [38, section 5.7.2 page 91], but is often found in real codes.

Aliasing makes many static analyses and transformations fail, because they assume there is no aliasing. For example, if the called subroutine is

```
subroutine COPYINTO(A,B,n)
   integer n,i
   real A,B

   do i=1,n
      B(i) = A(i)
   enddo
end
```

a parallelizer will probably declare the loop as `PARALLEL`. However, if the call is

```
call  COPYINTO(T(150),T(100),200)
```

then `A` and `B` actually overlap, and since the parallell loop can be executed in any order, the result is unpredictable and the program is wrong.

2.4.3 Pointers

Pointers add an extra level of uncertainly, about which variable is accessed by a given reference. Statically, one cannot tell where a pointer points to. Therefore, the conservative approximation “blurs” the information available for each possible target of the pointer.

This can be improved partly if one knows the possible targets of a pointer for one given instruction instance. This is the purpose of *pointer analysis*, implemented for example in PARTITA. This analysis improves the situation when the set of targets is smaller than the set of all existing variables, because fewer variables will get a “blurred” piece of information.

Pointer analysis is similar to *dependency* analysis (*cf* section 4.4.1). It propagates, for each pointer, the set of its possible targets and updates this set each time an instruction modifies a pointer. Yet, this anlaysis is expensive for a limited benefit: it is very often that the possible targets are very many. We think that probably, the best way to cope for that is to ask for user knowledge of the application, with directives. Notice this is similar to what FORTRAN95 does when requiring the `TARGET` attribute be put on possible array targets.

Chapter 3

Automatic Parallelization: Techniques, Tools, and Applications

This chapter is devoted to Program Parallelization, which is just one particular static analysis and transformation of programs. We emphasize the fundamental concepts that are common to the numerous flavors of parallelization. This chapter summarizes research that we made in the 1990's, when we developped the PARTITA semi-automatic parallelization tool [28], which was incorporated into the commercial environment FORESYS [46]. In this chapter, our main goal if to describe the influence of the techniques for representing and analyzing imperative programs. Parallelization tools are a proeminent application domain for these techniques, and provide ample justification for them. We shall especially focus on

- representing programs by a Call Graph and Flow Graphs (in section 3.1),
- extracting Data Flow information (in section 3.2),
- using Data Dependences to optimize scheduling of instructions (in section 3.3).

Specifically for parallelization, these techniques were implemented in the PARTITA tool for FORTRAN parallelization, and we describe two main concrete applications:

- the initial application to parallelization of loop nests (in section 3.4),

- a more recent SPMD parallelization of mesh-based applications [29, 31] (in section 3.5).

Many other program transformations can take advantage of these techniques. The following chapter 4 describes how they can be extended and applied to Automatic Differentiation, with what benefits.

3.1 Internal Representation of Programs

The internal representation of programs has a strong impact on the quality of the static analyses that can be run on them. It also impacts the possible transformations of these programs. An example for this is Automatic Differentiation in "reverse" mode, described later in section 4.3. Possible internal representation are discussed in section 2.2 and in the classics of compiler theory, and in particular in [2]. Of course everything starts with a *parsing* step, that builds an abstract *syntax tree* for the whole program. But this must not be the final internal representation. The most appropriate program representation for static analyses and transformations appears to be composed of the following three nested levels, from the finest to the coarsest:

- The individual, "atomic" instructions are represented as abstract syntax trees.
- The Flow Graphs represent the scheduling of instructions inside each given procedure. Each instruction has access to a *symbol table* holding the static properties of variables and other symbols.
- The Call Graph represents the procedures that compose the program.

The Call Graph captures the structure of the program as a set of procedures that call each other. Each procedure is a node in the Call Graph, and there is an arrow from procedure `A` to `B` if `A` contains a call to `B`. In object-oriented languages, or when pointers to functions are used, it is sometimes unclear statically whether `A` calls `B`. In this case, there is an arrow from `A` to `B` if `A` might call `B`. The Call Graph can contain cycles, which indicate recursion. There is generally one particular node representing the main procedure. There may be several Call Graphs involved in an analysis, for example when the program uses a separate library.

After internal representation is built and analyzed, transformation tools must build a new (parallelized, differentiated, *etc*) program. As shown on figure 3.1, we naturally build it in internal form, then go back through each step that led to the internal representation, to come up with a result in the natural, textual form. Downwards arrows on figure 3.1 show the successive

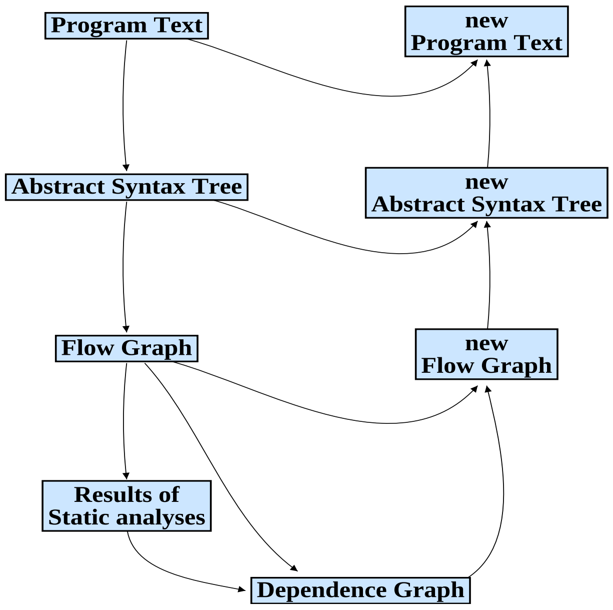


Figure 3.1: *Analysis and Regeneration pattern*

abstraction levels, from program text to syntax tree and then to Call Graph of Flow Graphs. At each level downwards, representation is more abstract, and probably also more language-independent. Then static analyses are run. For parallelization, their principal output is the *dependence graph*. The results of analyses are then used to build a new program in internal form. Upwards arrows show the final steps that lead to the new textual program. Notice the extra rightwards arrows: at each level, some information that was abstracted away on the way down may be reused on the way up.

3.2 Static Data Flow Analyses for a better Parallelization

Parallelization needs precise data flow analyses. Anticipating on section 3.3, a good parallelization relies on a good data dependence analysis, which in turn requires precise knowledge of the values involved in array indices.

As shown in section 2.1, conservative over-approximations must be made at some point, and they strongly impact the rest of the analysis particularly in the case of *arrays*. One can use *array regions* (*cf* section 2.4.1). Notice however that not all applications access arrays with regular indexes, and therefore other algorithmic developments may get a higher priority. Actually, it is often enough to consider a coarse over-approximation, in which each whole array is considered atomic, therefore generalizing the data flow properties of each array cell to the whole array.

For parallelization, as well as for most program transformations, the following static analyses prove particularly useful for the ultimate goal of a minimal data dependence graph.

- A *pointer analysis* builds, for each place in the program, the list of a pointer's possible destinations. This helps reduce the number of dependences.

- Constraints on the value of variables involved in array indexes help prove independence. These constraints may be detected automatically, or they may be provided by the end-user through the use of *directives*. For instance, such a constraint can be the lower and upper bounds of the value of variables. The *variable bounds* analysis propagates these bounds through the program. For other program transformations (e.g. *Partial Evaluation*), this anaysis is crucial and one needs to propagate more precise information than just variable bounds.

- Parts of the program that will never be reached will never create dependences. For example, *variable bounds* can tell us that some tests never return a given value. We shall call this *reach analysis*.

- Some arguments of a procedure are not read or not overwritten by the procedure. This may therefore remove *data dependences* to and from each call. For each procedure, this information is computed by the *Read-Written* analysis.

- When an array is overwritten at some location, it is important to know whether all cells of the array are overwritten, or only some cells. We call this *killed arrays* analysis. More generally, it would be good to know which "part" of the array is overwritten. This is the purpose of the *array region analysis* [14], not available in PARTITA.

- Finding *induction variables* helps. Those are variables that are linear functions of the normalized loop index. When an induction variable appears in an array index, the data dependence analysis is more accurate.

These analyses often depend on one another. Figure 3.2 shows the relative order between those implemented in PARTITA. Sometimes, different analyses may benefit circularly from each other. Dotted arrows on figure 3.2 illustrate just some of these circularities. We may ignore these circularities, and therefore loose some precision in the results. Alternately, we may run the analyses many times, until some fixpoint is reached, but this may be costly.

3.3 Parallelization of Programs, using the Dependence Graph

This section presents our definition of parallelization, organized around the central notion of *Dependence Graph*. After giving the motivation for *Data Dependences*, we give the structure of the *Dependence Graph*, whose arrows are the data dependences. We discuss some aspects of the effective construction of the dependence graph, which takes profit of the static analyses from previous section 3.2. We then show how the dependence graph is used for parallelization of loop nests. The next two sections 3.4 and 3.5 show concrete applications inside parallelization tools that we developped. The first is fine-grain parallelization of loop nests, and the second is coarse-grain SPMD parallelization of mesh-based computations.

3.3.1 Execution Order and Data Dependences

Any imperative programming language explicitly defines its execution order. The *execution order* of a program is the order in which all operations specified in the program text actually take place at run time. This order is

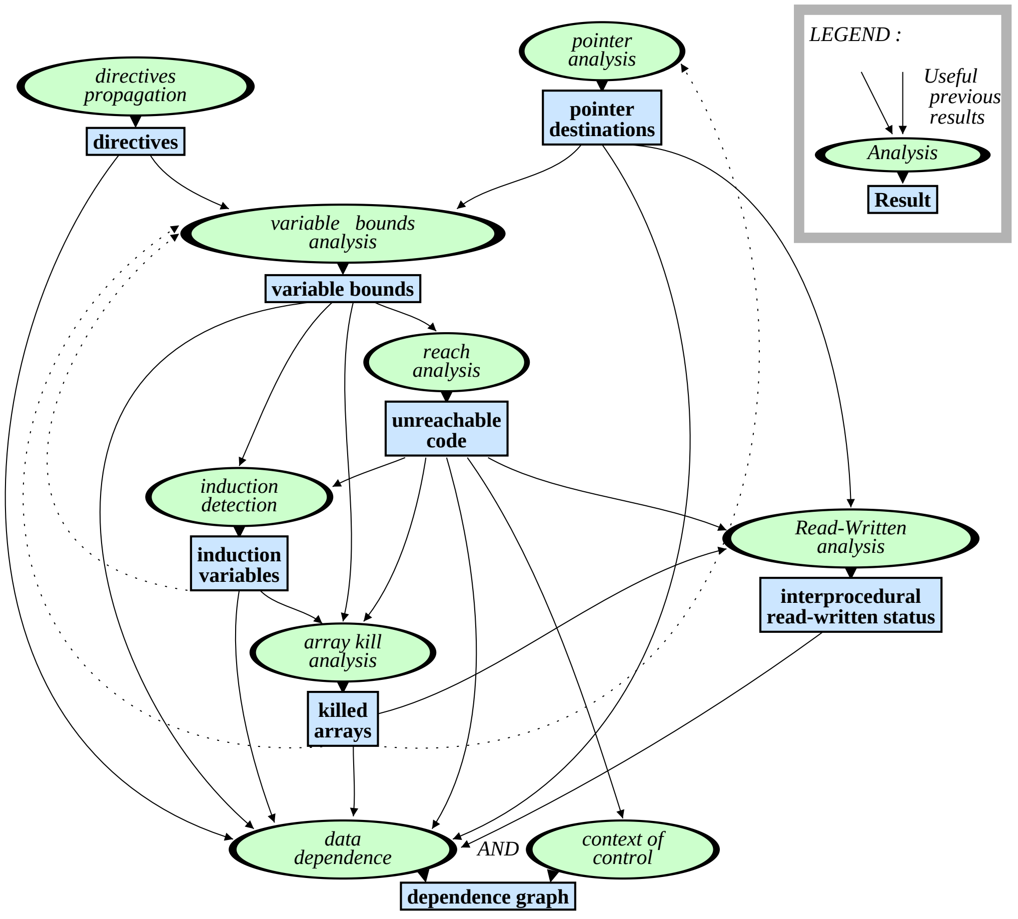


Figure 3.2: *Ordering of static data flow analyses in* PARTITA

fundamental, as the programmer relies on it to implement algorithms. For example in the following program fragment:

```
X = X * 4
SQ = SQRT(X)
```

the programmer assumes that instructions are executed in sequence. This assumption is granted by the definition of the language (here the FORTRAN standard). The program results would probably be different if this execution order was modified, because the value read for variable `X` would be different.

Notice however that the execution order is generally not specified *completely*. The definition of the language may leave some parts of it undefined. Let us call the order specified by the language definition the *language order*. The compiler is then free to choose any execution order *compatible* with the language order. For example, still in FORTRAN, the evaluation order of the actual parameters of a procedure call, or of the operands of an arithmetic operation, is not specified. This can sometimes lead to *nondeterminism*. For example the following program fragment:

```
J = I + INCR(I)
```

where function `INCR` increments and returns its argument, is not *deterministic* and is therefore illegal, because its result depends on the order in which operands of `+` are evaluated.

Supercomputers offer improved performances because some operations can be done concurrently. To take profit of this architecture, *parallel languages* define a weaker, more open language order, that allows the compiler to select a compatible execution order which is the most efficient on the target parallel machine. Figure 3.3 shows the language orders between the instances of instructions in a loop, defined by sequential, vectorial, and parallel loops. Of course, such a *parallel program* must be “correct”, whatever it means. If there is no reference specification available, we must at least check that it is *deterministic* i.e. the result does not depend on the particular execution order. This is a more convincing case of *nondeterminism* because it depends not only on the choices made by the compiler, but also on the run-time state of the “parallel processors”, which may very well change at next execution for a number of reasons.

Correctness is clearer in the case of *parallelization*, because we have an original sequential program which serves as a reference. Let us assume that

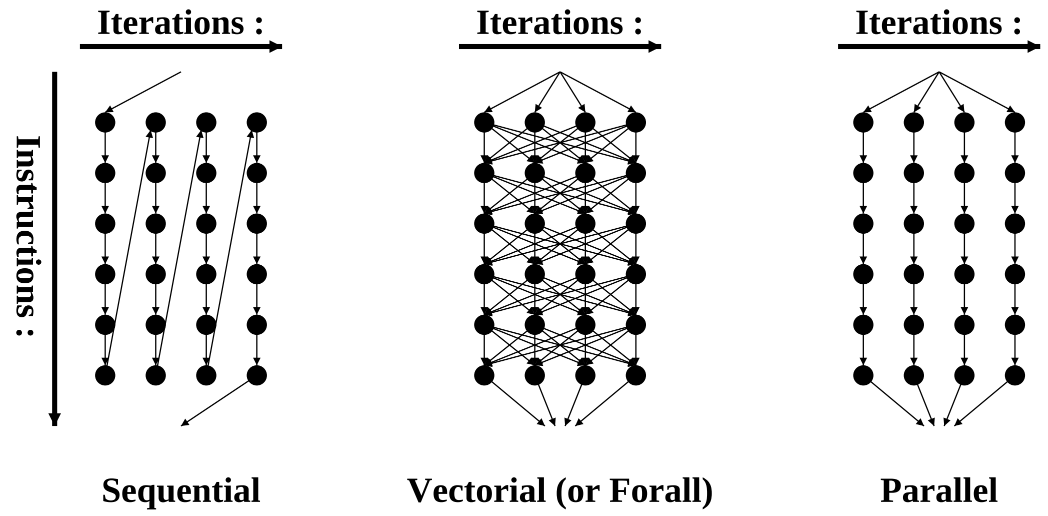


Figure 3.3: *Language Order of different kinds of loops*

parallelization does not change the operations done by the program, but only their *language order*. Parallelization transforms a deterministic sequential program P_1 (in language L_1) into a parallel program P_2 (in "target" language L_2). Parallelization is correct if for any execution order compatible with the L_2 language order of P_2, the result is the same as the result of P_1.

Now how could this result be different? Remember that we assume that the very same operations are done, operating on the same variables. Only their relative order may change. The result can be different only when some variables contain different values when they are read, and the only reason for that is that the execution order between *reads* and *writes* of the same memory cell has changed. This order due to *reads* and *writes* is called *data dependences*. In other words, *data dependences* are the subset of the *language order* where the origin and destination instructions contain at least one pair of depending *read-write* operations. Thus, parallelization is correct if *data dependences* are respected in any execution order of the parallelized program, or equivalently if *data dependences* are enforced by the language order of the parallelized program. As for vocabulary, *data dependences* fall into three categories (see figure 3.4 for how we draw them):

- **true** or **flow** dependences go from a *write* of a value into a variable to a subsequent *read* that may return this value.

- **anti** dependences go from a *read* of a value from a variable to a subsequent *write* that may overwrite this value.
- **output** dependences go from a *write* of a value into a variable to another subsequent *write* that may overwrite this value.

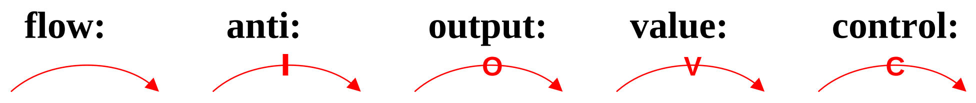


Figure 3.4: *Graphical notation for data dependences*

To summarize: from program P_1 we can extract the data dependences, that form a subset of the L_1 *language order* of P_1, and parallelization to P_2 is correct if these data dependences also form a subset of the L_2 language order of P_2. Figure 3.5 illustrates this: an original program (sketched on the left) which is a loop, might be modified into a fully parallel loop (middle), or into a parallel loop followed by a sequential loop (right). Thin arrows indicate the respective language orders. Thick arrows indicate data dependences, probably found by data dependence analysis on the original program. It turns out that transformation into the middle program is invalid, because the new language order does not enforce all data dependences. On the other hand, the program on the right preserves all data dependences, and is therefore valid.

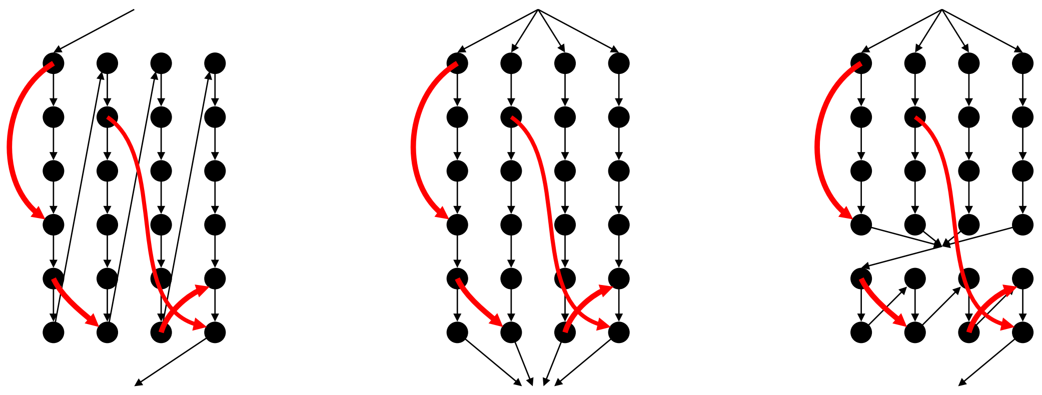


Figure 3.5: *An invalid and a valid parallelization of a loop*

Data dependences are constraints that forbid some (incorrect) transformations. The fewer data dependences, the better. But extraction of data dependences is a static analysis, undecidable in general. Sometimes, conservative assumptions must be made, that lead to unnecessary *"false"* data dependences. False dependences are a nuisance and should be avoided whenever possible. This explains why good static analyses are essential to a good data dependence analysis.

3.3.2 Memory management and communications

The previous section only deals with scheduling of operations. But most parallel environments not only distribute operations on many processors, but also distribute memory. Coherence between distributed memory spaces must be maintained, and data dependences are also the fundamental tool for this. Memory issues also arise in monoprocessor architectures, for example to improve data locality and cache management. Here also data dependences are fundamental.

Suppose again that we start from a given sequential program which serves as a reference, and that we want to check correctness of one parallelized version. We must check that every value read from a variable is the same in the two versions. If the value was written into one distributed piece of the memory, and the processor that does the read sees another piece of the memory, some communication must take place. Most often, to avoid repeated communication of the same value, the second piece of memory holds a local copy of the variable, which is updated by this communication.

In other words for each **flow** dependence, if the origin *write* and the destination *read* affect different pieces of the distributed memory, a communication must be inserted to propagate the written value towards the memory from where it will be read. Conversely, **anti** and **output** dependences between different pieces of memory can be neglected, because memory distribution has indeed created copies of variables, so that these so-called *artificial* dependences just vanish.

When evaluating the cost of a given memory distribution, **flow** data dependences indicate the amount of communication that this distribution requires. Section 3.5 describes an application of this strategy to the case of SPMD parallelization. Similarly, when splitting a piece of program into a number of parallel *tasks*, the **flow** dependences between operations sent to different tasks indicate communications and synchronizations between tasks.

3.3.3 The structure of the Dependence Graph

We said Data Dependences define a partial order between the program's run-time operations. Since we are here talking about static analyses (i.e. compile-time tools), we cannot list the exact set of run-time operations. Therefore we are obliged to consider *static* operations, i.e. the textual program's operations. When a textual operation is enclosed in control structures such as loops or tests, it represents all its run-time instances, which may be many, or none... The nodes of our *Dependence Graph* are the static operations, and not the run-time operations. Data dependences naturally project on static operations, and this induces the arrows of the *Dependence Graph*. But now this graph may be *cyclic*, because static operations may represent many run-time operations, folded into one single node. Consider for example two instructions `A` and `B` in a loop, `A` textually before `B`. Consider run-time instances a_i of `A` at loop iteration i, and b_j of `B` at loop iteration j. If $i > j$, b_j is executed before a_i. If there is a data dependence from b_j to a_i, it induces an arrow in the Dependence Graph, from node `B` to node `A`. Another consequence is that dependences in the *Dependence Graph* are now equipped with a *distance*, that represents the relation between the iterations of the enclosing loops, between the origin and destination run-time operations. A distance is a vector, with one element per common enclosing loop, topmost first. On the above example, the distance is $(i - j)$. Each element of this vector is either an integer, or some set of integers, such as an interval.

According to our needs, we may choose various granularities of the nodes of the *Dependence Graph*. For example, we may create one static operation for each atomic instruction in the program's text. Alternatively in PARTITA, we want to be able to split instructions to extract more parallelism. Therefore we create one static operation for each textual reference to a variable and one for each arithmetic operation. In that case, the language order between operands and elementary operations introduces additional dependences, that we call **value** dependences. Finally, and because controlling instructions must be executed before controlled instructions, we introduce the **control** dependences, that go from each loop counter or test result, to each operation that actually depends on it. See figure 3.4 for how we draw them.

Figure 3.6 shows the *Dependence Graph* for the original loop of figure 3.5. Here we chose a granularity of one node per atomic instruction, and therefore there are no **value** dependences. There are **control** dependences from the node that represents the loop counter to each node in the loop. The other data dependences are equipped with a distance. As one can expect, the *Dependence Graph* has lost some precision from the original data dependences, in the sense that it implicitly adds new, *false*, data dependences. Most parallelization techniques can live with this problem. Otherwise, the dependence distance must probably be refined.

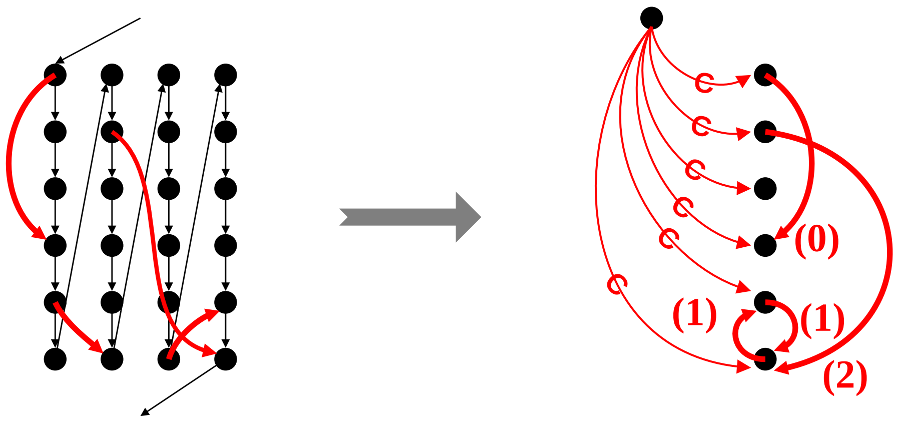


Figure 3.6: *A simple loop's Dependence Graph*

3.3.4 Effective construction of the Dependence Graph

Given two static operations, we want to detect whether there is a memory cell which is accessed by both operations, with one access being a write. This question easily boils down to the following: "given two array references, can they access the same memory cell, and at which conditions?" This question has motivated a great deal of research, and is still a hot problem. However a general strategy is commonly accepted, and we implemented it in PARTITA. Consider for example the following FORTRAN program:

```
subroutine test(T,g,k0,m,n)
integer i,j,k,k0,m,n,g,T
dimension T(-20:1000,300)

if (g.ge.7) then
  k = k0
  do i=0,n
     do j=1,m,2
        T(j,k) = ...
        ...  = ...  T(j+g,k+6) ...
     enddo
     k = k - 3
  enddo
endif
end
```

Let us look for the dependences from the static write of array T to the static read of T, i.e. from any write of T to any read of T, possibly after a number of iterations of the enclosing loops. If both operations access the same memory cell, this implies that the array indices are equal. On the example, this writes:

$$\mathtt{j} = \mathtt{j}' + \mathtt{g}'$$
$$\mathtt{k} = \mathtt{k}' + 6$$

where the $'$ sign distinguishes the values at the destination time (iteration context) from the values at origin time. Now we take profit of the static analyses done beforehand. In particular, detection of induction variables tells us that j and k are linearly related to the enclosing loop counters, $\mathtt{lc}_1$ for the outer loop and $\mathtt{lc}_2$ for the inner loop, by:

$$\mathtt{j} = 2*\mathtt{lc}_2 + 1$$
$$\mathtt{k} = -3*\mathtt{lc}_1 + \mathtt{k0}$$

Loop counters are always positive by definition. Therefore, the two initial equations become respectively:

$$-3*\mathtt{lc}_1 + 3*\mathtt{lc}_1' = 6$$
$$2*\mathtt{lc}_2 - 2*\mathtt{lc}_2' - \mathtt{g}' = 0$$

The bounds of the enclosing loops also imply constraints on $\mathtt{lc}_1$ and $\mathtt{lc}_2$, and we also know that $\mathtt{lc}_1' >= \mathtt{lc}_1$ because origin time must be *before* destination time. Read-Written analysis has shown that m, n, and g are loop constants, and variable bounds analysis tells us that inside the loop, we always have `g` >= 7. This constraint may be helpful for our problem. We end up with the following set of linear (in)equations:

$\texttt{lc}_1$	$\texttt{lc}'_1$	$\texttt{lc}_2$	$\texttt{lc}'_2$	$\texttt{m}$	$\texttt{n}$	$\texttt{g}$		
-3	3						$=$	6
		2	-2			-1	$=$	0
1					-1		$\leq$	0
1							$\geq$	0
	1				-1		$\leq$	0
	1						$\geq$	0
		2		-1			$\leq$	-1
		1					$\geq$	0
			2	-1			$\leq$	-1
			1				$\geq$	0
-1	1						$\geq$	0
						1	$\geq$	7

There are many techniques to solve this system, with different tradeoffs between quickness and precision. The simplest are the *gcd* and *separability* test, Among the most refined are the *Omega* test [43], or *parametric integer programming* [41, 42] or *simplexes* [48]. We used JANUS, a library developped by Jean-Claude Sogno [47]. It uses classical methods, such as the *gcd* test or the Fourier-Motzkin elimination. The tactiques that decide when these methods should be applied are internal to JANUS, and external users such as PARTITA need not worry about these choices. This is natural because these choices mostly depend on the actual linear system. On this example, JANUS finds that this system has integer solutions, and therefore there exists a dependence. Moreover, it returns necessary constraints on the dependance distance vector:

$$\begin{array}{c} 2 \leq \texttt{lc}'_1 - \texttt{lc}_1 \leq 2 \\ -\infty < \texttt{lc}'_2 - \texttt{lc}_2 \leq -4 \end{array}$$

so that the distance vector writes, with topmost iteration first: $(2,]-\infty, -4])$

3.4 Application 1: Loop Nests Parallelization with PARTITA

This section describes the actual use of the Dependence Graph in the PARTITA tool, which is the fine-grain loop parallelizer component of the commercial environment FORESYS. The people who contributed to the development of PARTITA are Laurent Angeli, Didier Austry, Bernard Dion, Laurent Hill, Nicole Rostaing, Jean-Claude Sogno, Guillaume Viland, Dominique Vilard, and myself. Development of PARTITA was also supported by several European projects, the last one about extension to OpenMP [1].

Based on the objectives stated in section 3.4.1, we first define a parallelization tactic for a single loop, in section 3.4.2, then we define in section 3.4.3 a general representation for dependences inside arbitrary loop nests, and show in section 3.4.4 how the loops transformations we are targetting at, express nicely on this representation. These tansformations are then combined into a tactic, and its result is shown on examples in section 3.4.5. We terminate with a glimpse of PARTITA's user interface in section 3.4.6.

3.4.1 Objectives of the PARTITA tool

The goal of PARTITA is to transform FORTRAN77 programs into equivalent programs in various parallel "dialects" of FORTRAN, such as HPF, FORTRAN95, OpenMP, *etc.* PARTITA must extract as much parallelism as possible from the loops found in real programs. Therefore, we don't want to put restrictions on the style of the input program. Also PARTITA can decide to split expressions when they contain a sizeable parallel part. Also, we want to apply most classical program improvements, even when they are not directly related to parallelization. All these transformation have in common the fact that they are best expressed on the Dependence Graph. Therefore, among the jungle of parallelizing program transformations, here are the loop transformations that PARTITA targets at:

- **dead code elimination:** removing code that is either never reached, or which has no influence on the program's outputs.
- **invariant code motion:** taking code out of loops when it executes identically at each iteration.

- **variable expansion:** expanding local variables of a loop into arrays, therefore removing some anti-dependences and allowing for more vectorization.
- **loop distribution, fission, fusion:** splitting or merging loops on the same iteration domain, to maximize the size of those resulting loops which turn out to be parallel or vectorial.
- **reduction detection:** finding reduction operations, such as global sums or products of some array's elements, in order to use a parallel reduction primitive.
- **loop rotation:** peeling off some instructions from the first and last iterations of a loop, in order to reorder its instructions and find more loop-local variables.
- **loop exchange:** exchanging two nested loops, in order to push the "parallel" level inwards, or to pull it outwards, according to the target parallel language.

There are more sophisticated loop transformations, also based on the Dependence Graph, that take into account several levels of nested loops. These are

- **loop unimodular transformations:** modifying the traversal order of the iteration domain of nested loops, to exhibit new parallel dimensions.
- **loop sectioning, strip mining, tiling:** changing a single loop into two (possibly nested) loops, one of which is parallel.

These sophisticated transformations did not fit well in the framework we have designed, so we chose to subcontract them to another tool, BOUCLETTE developped by the LIP laboratory in Lyon [7]. An interface was built to couple the tools, so that PARTITA could benefit from the transformations provided by BOUCLETTE. On the large basis of application programs that PARTITA was used on, experience shows that loops that can benefit from BOUCLETTE are very rare, but for these loops the improvement is obvious. We shall not consider these sophisticated transformations in the sequel.

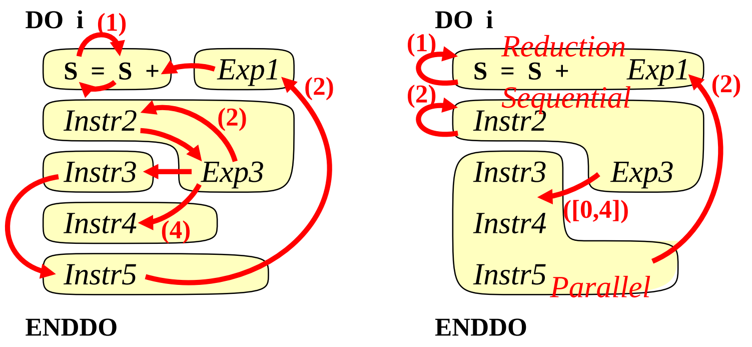
3.4.2 Parallelization of a single loop using Acyclic Condensation

To detect whether a given loop is in fact **parallel**, or **vectorial**, or not (we shall say **sequential**), one just has to look at the Dependence Graph. The rules are simple:

- If the Dependence Graph for the loop contains a cycle, then the loop is not parallel. However there are interesting special cases:
  - If the cycle is only due to a variable which, at each step, is modified by adding or multiplying to its previous value, then the cycle corresponds to a **reduction**, which can be done efficiently on supercomputers.
  - If the cycle is due to a variable which is overwritten at each iteration of the loop, it may be profitable to perform **variable expansion** or equivalently **localization**. The cost is to introduce an array where there was a scalar. The benefit is to remove some **anti** and **output** dependences, which may break the cycle.
  - Otherwise the loop must remain **sequential**.
- If there is no cycle in the loop's Dependence Graph, then the loop can probably be rewritten efficiently for the target architecture:
  - If the instructions are simple enough (e.g. no procedure calls, array indices always in the same order,...) then it can be rewritten vith **vectorial** syntax (FORTRAN95 so-called *array* notation).
  - If all **flow** dependences in the loop's Dependence Graph have distance (0), then it can be rewritten as a **parallel** loop.
  - Otherwise, unfortunately the loop must remain **sequential**.

Furthermore, it is well known that any loop can be split if there is no dependence cycle between the two split parts of the loop. This leads us to a general tactic for parallelization of a given loop, which is called *acyclic condensation.* It begins building the Dependence Graph between all the static operations in the loop. This means we choose a fine granularity, with every single variable reference and arithmetic operation yielding one node in the Dependence Graph. Then these nodes are gathered into groups. Firstly all nodes involved in a cycle must go to the same group, which will be labelled as **sequential**, or when appropriate as **reduction**. The other nodes go into one singleton group each, which is labelled as **vectorial**, **parallel**, or **sequential**, following the rules above. Each group represents a separate new loop (or a vectorial instruction) in the future parallelized program. By construction, the dependences between groups are now **acyclic**, and define the relative order of these future loops.

Then grouping (**condensation**) goes on, to further reduce the number of groups. This will reduce the cost of loop overhead and the number of temporary variables due to split instructions. But there are opposite constraints: *(a)* two groups can be merged only when there is no third group caught between them in the Dependence Graph, and *(b)* when two groups with different labels are merged, the resulting label is of course the worst, e.g **sequential** merged with **vectorial** gives **sequential**. To implement this tradeoff, PARTITA defines an evaluation of the cost of each group, based on atomic costs of each operation and loop construct. We choose among possibilities according to their cost.



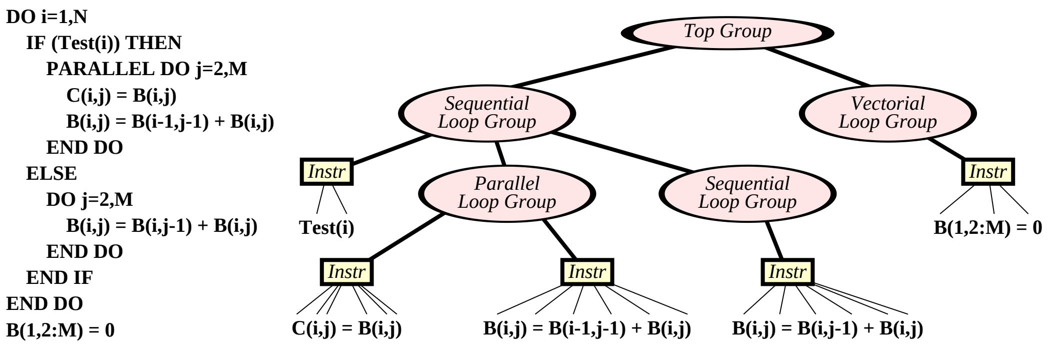
```
DO i
    Instr2
    tmp(i) = Exp3
ENDDO

PARALLEL DO i
    Instr3...tmp(i)...
    Instr4
    Instr5
END PARALLEL DO

S = S + SUM(Exp1)
```

Figure 3.7: *Acyclic Condensation on a single loop*

Figure 3.7 illustrates this tactic. On the left is the original loop with its Dependence Graph. Dependence distances are shown only when different from (0). For readability, we didn't push granularity to its finest: only a couple of expressions are shown split, therefore introducing a couple of **value** dependences. The boxes below show the original acyclic condensation. As condensation goes on, we reach the situation shown in the middle, where the parallel parts have been merged. The cost of introducing a temporary array "`tmp`" for the third instruction has been judged acceptable, compared to the benefit of keeping the left half of the instruction parallel. The right part of the figure sketches what the resulting parallelized program might look like.

3.4.3 Representing Nested Loops with Nested Groups

In the above we supposed there was only one loop, but in general loops can be nested. To cope for that, we introduce a tree structure for groups, that we call the *group tree*. The leaves of this tree are simple instructions from the original procedure, possibly split. Each node of this tree is a group, representing the code for the sons of this node, wrapped into a new loop structure. The type of this loop (sequential, parallel, vectorial, *etc*) is determined by the type of the group. In short the group tree summarizes the structure of nested loops. For example, figure 3.8 shows the *group tree* of the small code fragment on the left, that features various types of loops, nested. The leaves are single instructions. The groups above represent an *iteration* of their enclosed subgroups, of type specified by the group. When such a group derives from a loop of the original program, it retains the iteration domain of this original loop. The top group does not iterate: it mainly gathers the toplevel of the procedure. The Group Tree is a convenient structure for loop trans-



```
DO i=1,N
  IF (Test(i)) THEN
    PARALLEL DO j=2,M
      C(i,j) = B(i,j)
      B(i,j) = B(i-1,j-1) + B(i,j)
    END DO
  ELSE
    DO j=2,M
      B(i,j) = B(i,j-1) + B(i,j)
    END DO
  END IF
END DO
B(1,2:M) = 0
```

Figure 3.8: *The Group Tree of a small code fragment*

formations. All the loop transformations implemented in PARTITA amount to simple modifications on it. This allows us to concentrate on the essence of the transformation, forgetting all the syntactic details that would come up if transforming the program itself. Transformations of the Group Tree are driven by Data Dependences between sub-groups of a given group. Such a *group dependence* between two groups specifies their relative order with respect to the parent group. Its distance is the iteration distance with respect to the parent group loop. Group dependences are obtained by simple projection of the data dependences. Here are the rules governing this projection: If two groups have different parent groups, their order is guaranteed by the order between the parents. Therefore, a data dependence will be projected between two groups only if these groups have the same parent group. For the same reason, when a dependence projects on the parent group with a distance that cannot be zero, it will not be projected between the groups underneath, because this dependence will be respected as soon as the father group respects it. Notice that a dependence of distance zero from a group to itself is meaningless, because it is systematically respected. Formally, consider two (possibly identical) groups g_1 and g_2, their ancestor groups (top-down) $Top = a_0$, a_1, ...,a_n, and a dependence of distance $(d_1, d_2, ..., d_n, ...)$ from an operation in some leaf of g_1 to an operation in some leaf of g_2. The d_i are integers or integer variables ranging in an interval. The dependence will be projected between groups g_1 and g_2 if

- For all i such that $0 < i < n$, d_i is or may be equal to zero.
- Either $g_1 \neq g_2$, or d_n is or may be different from zero.

When it exists, the projected *group dependence* will receive distance d_n. By definition of the top group, for $n = 0$, $d_0 = 0$. The projected dependence retains the kind (**flow**, **anti**, **output**, **value**, or **control**) of the original dependence.

Figure 3.9 shows this projection for some example group trees, for the same data dependence of distance $(0, 1)$. Figure 3.10 shows that the projections are quite different when the distance of the original data dependence is $(1, 0)$. One can see that projections vary according to the distance of dependence and the configuration of the group tree. This corresponds to the idea, expressed for example by Kennedy and Allen in [3], that each dependence has a *nesting level*, such that the dependence does not exist in loops nested deeper than this level. In other words, if the distance of a dependence supposes some iteration of an outer loop, then for a fixed iteration of this outer loop, this dependence vanishes.

When the Group Tree is modified, dependence projection must be updated, which is cheap, but the Dependence Graph need not be recomputed, which would be expensive. We believe this is the main advantage of this

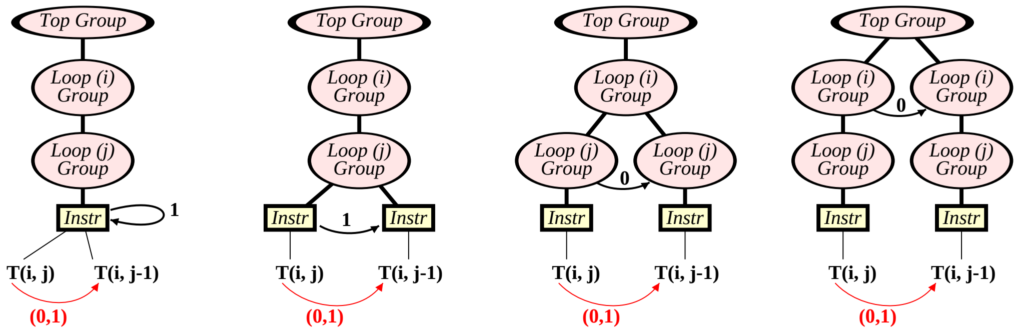


Figure 3.9: *Projections of a data dependence of distance (0,1)*

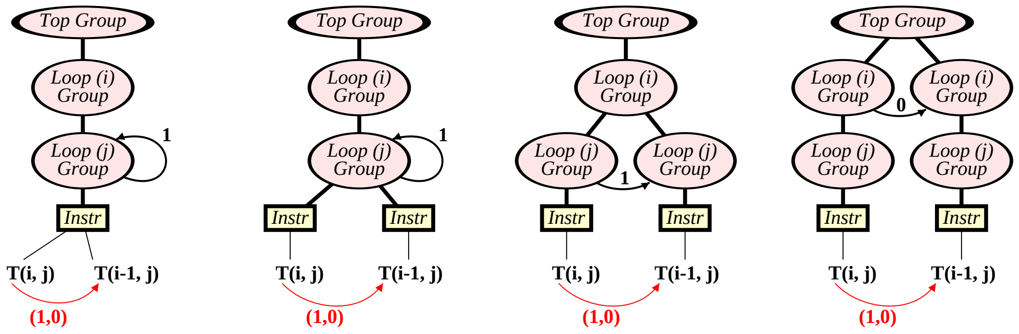


Figure 3.10: *Projections of a data dependence of distance (1,0)*

representation compared to other tools which repeatedly regenerate a new program after each transformation, and analyze it again before the next transformation.

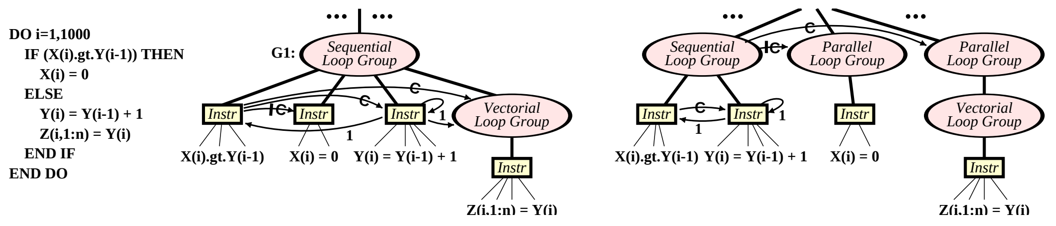
The group tree is not executable. It has lost the information about branches of conditionals. For parallelization analysis, this has no consequence. This just keeps the data structure simple and manageable. For code regeneration, however, this control information is necessary, and it will be extracted from the original Flow Graph.

3.4.4 Expressing Elementary Parallelizing Transformations on Group Trees

In this section, we present some of the most representative parallelizing transformations available in PARTITA. For each of them, we show how applicability is checked on the group tree, and how it is then actually performed on the group tree.

Loop Distribution

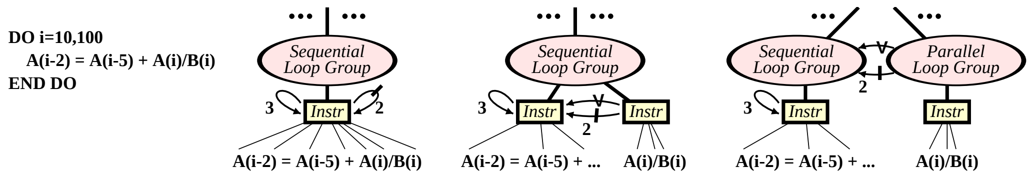
Loop Distribution is probably the most classical parallelizing transformation. What we are talking about here is indeep called *loop fission*, because it preserves the meaning of the program. Consider a *sequential* loop group G1, such as G1 on figure 3.11. Loop distribution first finds the strongly connected components of the dependence graph projected between all immediate descendents of G1. This uses the classical *Tarjan algorithm* [45]. Each cycle may be transformed into a separate *sequential* loop group. Other components (not cyclic), become separate *parallel* loop groups. After the transformation, the group G1 disappears and is replaced by the new *sequential* and *parallel* groups. Then data dependences are then re-projected.



```
DO i=1,1000
  IF (X(i).gt.Y(i-1)) THEN
    X(i) = 0
  ELSE
    Y(i) = Y(i-1) + 1
    Z(i,1:n) = Y(i)
  END IF
END DO
```

Figure 3.11: *Loop Distribution on the group tree*

Notice that a leaf *Instr group* can always be distributed, because since an *Instr group* expresses no iteration, there is no risk of finding a dependence cycle Distributing an *Instr group* means splitting one instruction into many, using temporary intermediate storage, in a *three-address code* fashion. The advantage is the possibility to exhibit parallelism , later, for just a part of an instruction. Figure 3.12 shows a case where distribution of a *Instr group*, followed by a loop distribution, exhibits a parallel part of the instruction. Notice that the *value* dependence between the two final loop groups implies



```
DO i=10,100
   A(i-2) = A(i-5) + A(i)/B(i)
END DO
```

Figure 3.12: *Instr Distribution followed by Loop Distribution*

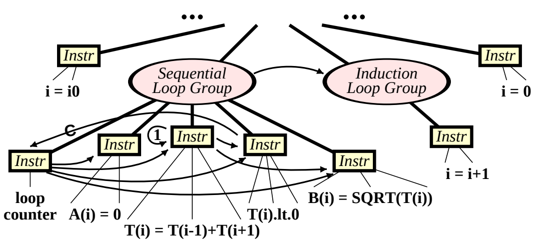
a temporary storage, to communicate the value from a loop to the other. The size of this storage is determined by the nesting level of the two groups. Here, a one-dimensional array is necessary. The cost of that is taken into account by PARTITA when evaluating the cost of this configuration of the group tree. A similar remark applies to the *control* dependence between the final loop groups of figure 3.11.

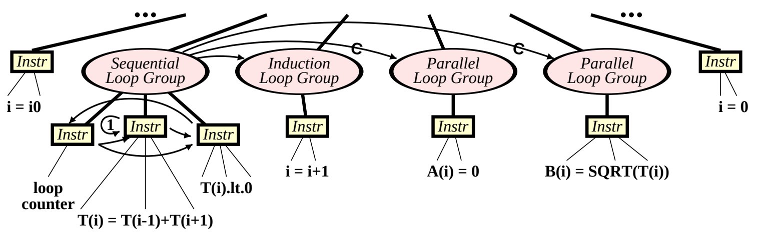
Distribution of Natural Loops

The above *loop distribution* is powerful enough to work on natural loops. We just need to introduce a new atomic operation, that we call the *loop counter*. There is one loop counter for each loop. This "operation" is in charge of giving the value of all induction variables of the loop. In particular, it gives the value of the `DO` loop index. For a normal `DO` loop, the loop counter immediately knows the number of iterations of the loop and all the induction values. So far, for clarity, we did not represent this loop counter on the figures.

Now suppose that for a given loop, loop exit is not known when the loop starts. This is the case when a `DO` loop contains an exit, of for `WHILE` loops, or for natural loops. This is the case on figure 3.13. We claim one should not abandon hope of parallelization yet. The loop counter now has dependences towards all uses of the induction variables, as usual, but also receives control dependences from the places where “exit” decision is taken. Therefore, loop distribution will leave it in a *sequential loop group*, and instructions involved in the dependence cycle will remain in this group. Other instructions may, as in figure 3.13, be out of the cycle, because they are not involved in the “exit’ decision. Therefore they might be parallelized. This is just plain application of our *loop distribution* on groups. Here again, the *control* dependences from

```
        i = i0
400     A(i) = 0
        T(i) = T(i-1)+T(i+1)
        IF (T(i).lt.0) THEN
           GOTO 500
        ELSE
           B(i) = SQRT(T(i))
        ENDIF
        i = i+1
        GOTO 400
500     i = 0
```





```
        i = i0
        lc = 0
400     T(i) = T(i-1)+T(i+1)
        IF (T(i).lt.0) THEN
           GOTO 500
        ENDIF
        i = i+1
        lc = lc+1
        GOTO 400
500     PARALLEL DO i=i0,i0+lc-1
           A(i) = 0
           B(i) = SQRT(T(i))
        ENDDO
        A(i0+lc) = 0
        i = 0
```

Figure 3.13: *Loop Distribution applied to an arbitrary Natural Loop*

the loop counter will require a temporary storage. Typically, one must store the actual rank of the last iteration. Figure 3.13 shows the resulting program generated by PARTITA.

Variable Expansion and Localization

It is well known that artificial dependences (i.e. *anti* and *output* dependences) can be removed by rewriting the program in a single-assignment fashion. This amounts to creating new variables, to avoid variable overwriting. In particular, in loops, local variables are overwritten at the beginning of each iteration. Variable expansion replaces the local variable by an array of variables, one for each iteration. Equivalently, localization tells the compiler to create one variable per parallel processor.

Variable expansion or localization may be applied, with profit, to a variable in a *sequential group* if cutting the artificial data dependences due to this variable actually breaks the dependence cycle between the subgroups of this group. This is the main condition, although minor restrictions apply.

When each iteration of the loop does not start with an overwrite of the expansion candidate variable, PARTITA is able to perform another transformation, **loop rotation**, to try to move the overwrites at the beginning of each iteration. If this succeeds, then PARTITA may apply **variable expansion**. We shall not describe this in greater detail.

We talk of **variable expansion** rather than **scalar expansion** because all the above applies also to arrays. Since PARTITA is able to detect when arrays are completely overwritten, it can perform *array* expansion or localization.

When variable expansion is chosen, the effect on the group tree is to annotate the *read* and *write* of the variable, and the artificial data dependences between them, as "expanded" with respect to the loop. As a consequence, expanded dependences will not be projected at this loop level in the group tree. Then ordinary *loop distribution* may be applied to the *sequential loop group* again, yielding new *parallel loop groups*. The other effect, which is the actual expansion or localization of the variable, will be postponed until program regeneration time. The advantage is that, if for some reason, variable expansion is decided not profitable, it may be undone at very low cost, simply removing the corresponding "expanded" annotations. Figure 3.14 shows an example of variable expansion of scalar **x** and the effect on the group tree. The little star represents the "expanded" annotation.

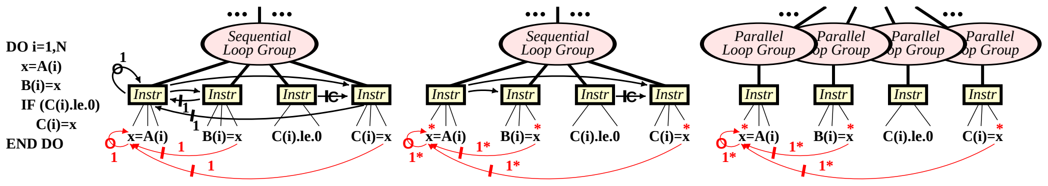


Figure 3.14: *Scalar Expansion followed by Loop Distribution*

Detection of Reductions

Before any specific analysis, reduction operations appear as sequential groups. This is because the same variable is repeatedly read then written at each iteration. However, most target architectures implement reductions far more efficiently. Therefore we need to detect them. A sequential loop group corresponds to a reduction if the following conditions hold:

- Group dependences follow a standard pattern (e.g. all **anti** dependences have distance 0).
- sub-groups are either *Instr* or other *reduction* groups.
- The involved operation is a reduction (e.g. sum or product)

In this case, the sequential group may be replaced by a *reduction* group. Like for variable expansion, some data dependences will be annotated, so that they will not be projected as long as the group type remains *reduction*. Typically this is the case for the **output** dependences on the reduction variable. Figure 3.15 shows an example of reduction detection.

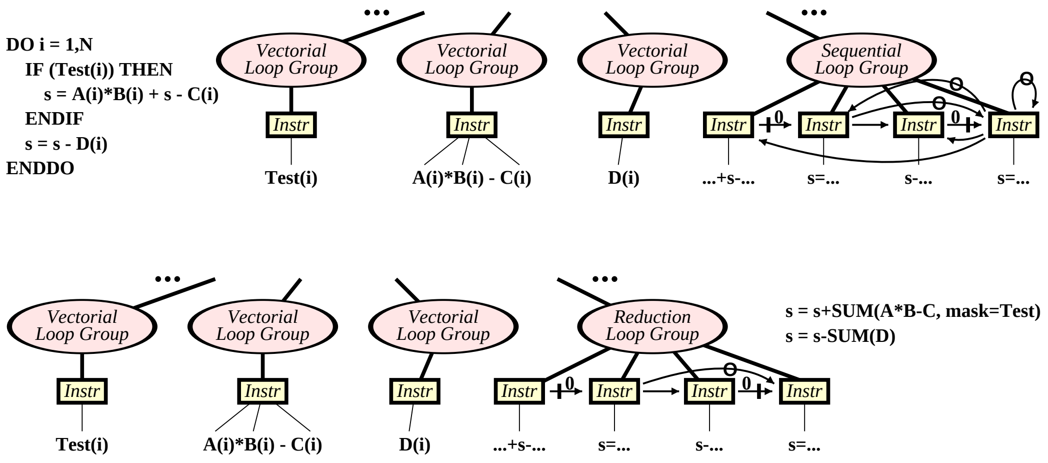
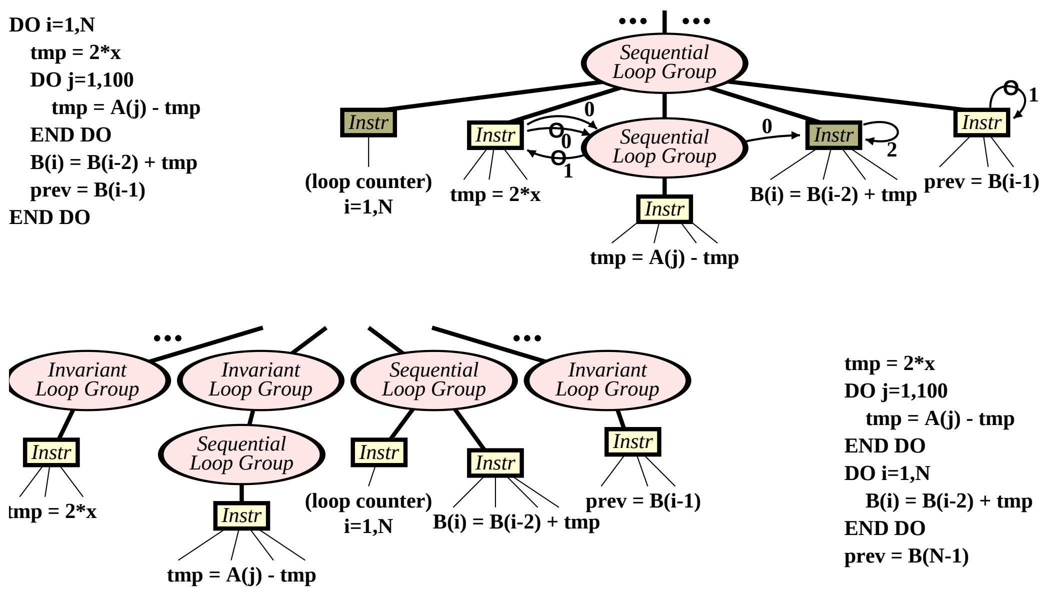


Figure 3.15: *Detection of a global sum Reduction*

Invariant Code Motion

In a loop, some operations may be invariant: instead of being repeated at each iteration, they may be moved out of The loop, without modifying the semantics of the program. On the initial program of figure 3.16, this happens for the first and last instructions, as well as for for the sub-loop. Note that *variable expansion* of scalar **tmp** is also possible, but *invariant code motion* is



```
DO i=1,N
   tmp = 2*x
   DO j=1,100
      tmp = A(j) - tmp
   END DO
   B(i) = B(i-2) + tmp
   prev = B(i-1)
END DO
```

```
tmp = 2*x
DO j=1,100
   tmp = A(j) - tmp
END DO
DO i=1,N
   B(i) = B(i-2) + tmp
END DO
prev = B(N-1)
```

Figure 3.16: *Invariant Code Motion on the Group Tree*

more profitable. Invariant code detection and motion is discussed in compiler litterature, for example in [2]. However, it focuses on instructions that can be moved *before* the loop. In PARTITA, we also detect the instructions that can be moved *after* the loop. This leads to the idea that there are some operations that must definitely remain in the loop. We shall say they are *anchored* in the loop. These are the loop's *loop counter*, and the *write* operations, except those who write in the same memory location at different iterations, and whose written value is only used during the same iteration and then overwritten at next iteration. Therefore a *write* operation is anchored except if we have at the same time:

- all outcoming **flow** dependences have a distance of exactly 0.
- there is at least one outcoming **output** dependence of distance 0 or 1, and none of distance greater than 1.

All operations that are caught in a chain of dependences between two anchored operations must remain in the loop too. The other operations may be moved out of the loop. This algorithm detects invariant code both before and after the loop, even in complex situations involving sub-loops. Figure 3.16

shows an example. We suppose we know `N` > 0, so that the loop body is executed at least once. For clarity, only dependences useful to the algorithm are shown. The groups that contain anchored operations are shown darkened.

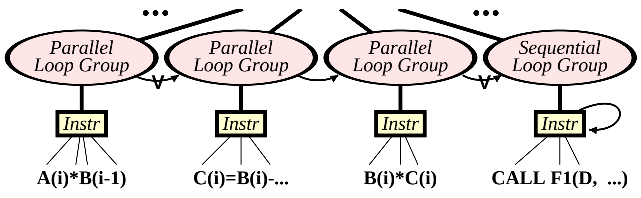
Loop Fusion

Loop Fusion is the reciprocal of Loop Distribution. It reduces the number of distinct loops, and therefore the loop overhead. Also, when fusing operations from the same original instruction, Loop Fusion reduces the number of temporary variables. The condition to apply Loop Fusion is that the fused groups must have the same parent group, and no other group is caught between them in the projected dependence graph.

The drawback of Loop Fusion is that the type of the fused group often turns out to be the worst of the original types. One subtlety comes when fusing two *parallel* groups. One must examine the projected dependences between the sub-groups of the fused group. If all dependences are of distance zero, then the new group may be parallel. Otherwise, fusion can be a bad idea, or there must be a synchronization between two iterations of the loop, or a `BARRIER` in the loop body. In the worst case, it may happen that fusion of two *parallel* groups returns a *sequential* group.

Figure 3.17 shows Loop Fusion applied on groups, top-down, first between *parallel* groups, then between *Instr* groups. We suppose that procedure `F1`

```
DO i=1,N
   C(i)=B(i)-A(i)*B(i-1)
   CALL F1(D,B(i)*C(i))
END DO
```



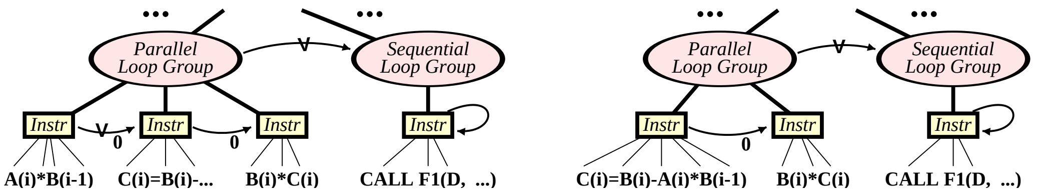


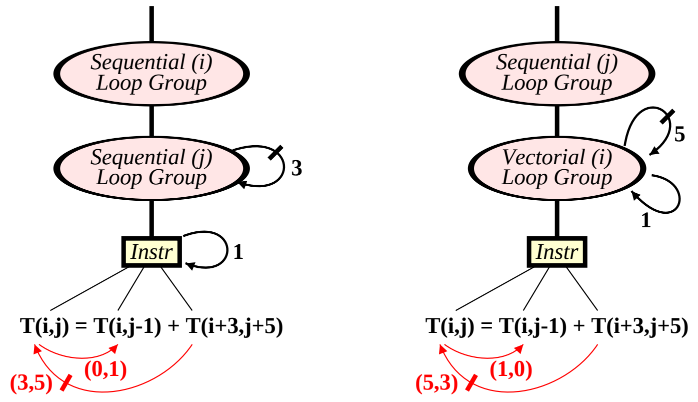
Figure 3.17: *Loop Fusion between Loop Groups and Instr Groups*

has no side-effect, but has some effect on its first argument that induces a cycling dependence. Notice that the dependance distances inside the resulting parallel groups are all zero. Therefore the fused group is *parallel*. Figures 3.11 and 3.12 are also examples of group fusion, when read in the reverse direction!

Loop Exchange

Consider the nested groups of figure 3.18, corresponding to two perfectly nested loops. Classically, these loops may be exchanged if some constraint

```
DO i = 1,N
   DO j = 1,M
      T(i,j) = T(i,j-1) + T(i+3,j+5)
   ENDDO
ENDDO
```



Sequential (i) Loop Group
Sequential (j) Loop Group
3
Instr
1
T(i,j) = T(i,j-1) + T(i+3,j+5)
(3,5)
(0,1)
Sequential (j) Loop Group
Vectorial (i) Loop Group
5
1
Instr
T(i,j) = T(i,j-1) + T(i+3,j+5)
(5,3)
(1,0)

Figure 3.18: *Loop Exchange on the Group Tree*

holds on the distance vector of all data dependences in the loop. Precisely, all dependance distances with respect to the inner loop must be greater or equal to 0. This must be checked on the data dependences *before* projection, because the projection operation has lost the distance at the inner loop level. On the example of figure 3.18, the constraint holds. The rightmost group tree is better for a vectorial architecture because the inner group is vectorial.

3.4.5 A tactic combining Elementary Transformations

When working on large applications, it is unrealistic to apply elementary loop transformations interactively. Some tactic (heuristic) must be provided that gives good results on most loops. The tactic in PARTITA has the following successive phases:

1. **initialization:** The initial group tree reflects the initial subroutine: one *sequential* loop group per loop, one *Instr* group per instruction.

2. **splitting:** Loop distribution is applied repeatedly, bottom-up on the group tree. The resulting groups are typed as *sequential* or *parallel* according to the dependencies pattern, and special algorithms refine some groups type as *invariant*, *reduction*, or *induction*. *sequential* groups that can benefit from *Variable Expansion* are transformed.

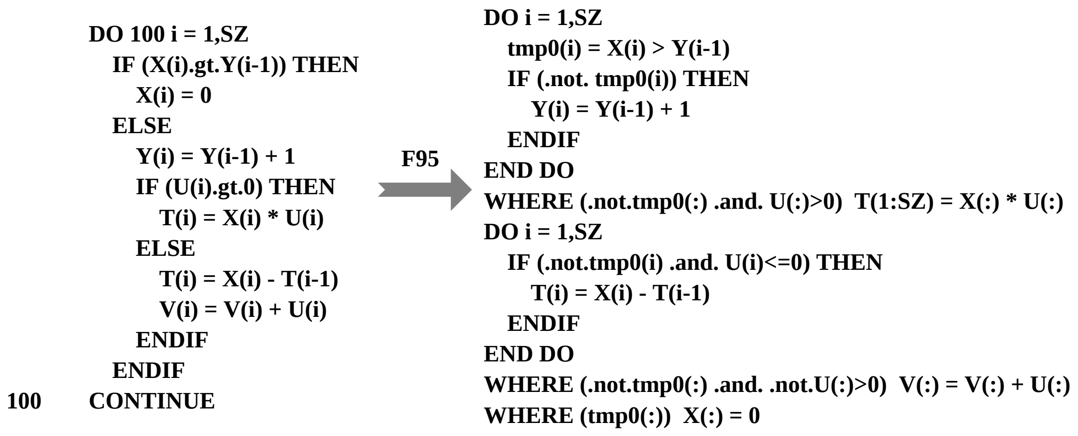
3. **merging:** Loop fusion is applied repeatedly, top-down on the group tree. A cost/benefit is run at each step, to evaluate whether fusion is profitable for the given target architecture. During this phase, *parallel* groups may be turned into *vectorial* or *forall* groups if the syntax permits this and if the target architecture can use this.

4. **cleaning:** Some transformations, such as *Variable Expansion* or detection of *induction variables* are useful only if they allow for extraction of parallel loops. If the cost/benefit analysis turned these loops back to sequential, then *Variable Expansion* or *induction variables expansion* become counterproductive, and must be undone. This is easy to do on the group tree.

5. **ordering:** Until then, the order of the sub-groups of a group was meaningless. Now we order these groups, actually choosing the order or the regenerated program. The ordering must respect the projected dependencies between the sub-groups. We look for the best possible order, which is presumably an NP-hard problem, so we implement a heuristic.

We show the result on small example programs. The parallelization tactic is strongly driven by a description of the target language and architecture, so for each program the result is most often different for different targets. Let us start with vectorization. Figure 3.19 shows vectorization towards target `Fortran95` of a single loop fragment. This required loop distribution, and a temporary array to remember a test. The test could not be duplicated because array `Y` is modified.

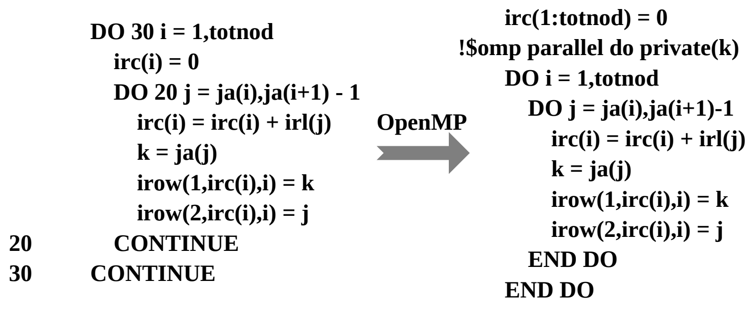
Consider now a target language, such as OpenMP, that in addition provides parallel loops. On figure 3.20 the outer loop is parallelized, while the inner loop must remain sequential. Also, variable `k` is localized with respect to loop `i`. On figure 3.21, PARTITA was obliged to insert a `BARRIER` synchronization, because there is a dependence of distance 3. Functions `Func1` and `Func2` are user-defined, and have no side-effect. Since target `OpenMP` also



```
       DO 100 i = 1,SZ
          IF (X(i).gt.Y(i-1)) THEN
             X(i) = 0
          ELSE
             Y(i) = Y(i-1) + 1
             IF (U(i).gt.0) THEN
                T(i) = X(i) * U(i)
             ELSE
                T(i) = X(i) - T(i-1)
                V(i) = V(i) + U(i)
             ENDIF
          ENDIF
100    CONTINUE
```

```
DO i = 1,SZ
   tmp0(i) = X(i) > Y(i-1)
   IF (.not. tmp0(i)) THEN
      Y(i) = Y(i-1) + 1
   ENDIF
END DO
WHERE (.not.tmp0(:) .and. U(:)>0)  T(1:SZ) = X(:) * U(:)
DO i = 1,SZ
   IF (.not.tmp0(i) .and. U(i)<=0) THEN
      T(i) = X(i) - T(i-1)
   ENDIF
END DO
WHERE (.not.tmp0(:) .and. .not.U(:)>0)  V(:) = V(:) + U(:)
WHERE (tmp0(:))  X(:) = 0
```

Figure 3.19: *Vectorization with Loop Distribution*



```
      DO 30 i = 1,totnod
         irc(i) = 0
         DO 20 j = ja(i),ja(i+1) - 1
            irc(i) = irc(i) + irl(j)
            k = ja(j)
            irow(1,irc(i),i) = k
            irow(2,irc(i),i) = j
20       CONTINUE
30    CONTINUE
```

```
   irc(1:totnod) = 0
!$omp parallel do private(k)
   DO i = 1,totnod
      DO j = ja(i),ja(i+1)-1
         irc(i) = irc(i) + irl(j)
         k = ja(j)
         irow(1,irc(i),i) = k
         irow(2,irc(i),i) = j
      END DO
   END DO
```

Figure 3.20: *Generation of a Parallel DO Loop*

provides vectorization, with a higher preference, the third instruction of the loop has been vectorized.

```
      DO 100 i=1,1000,2
         R(i) = Func1(S(i))
         T(i) = Func2(R(i+3))
         A(i) = 2*T(i) - A(i)
100   CONTINUE
```

OpenMP

```
!$omp parallel do
   PARALLEL DO i = 1,1000,2
      T(i) = Func2(R(i+3))
!$omp barrier
      R(i) = Func1(S(i))
   END DO
   A(1:1000:2) = 2*T(1:1000:2) - A(1:1000:2)
```

Figure 3.21: *Automatic Insertion of Necessary Synchronization*

Target language HPF provides the `forall` construct, which is an extension to vectorial notation, and the syntax of the parallel loops changes slightly. Figure 3.22 shows code generation for this target. Note that the dependence pattern of the loop forces PARTITA to split one instruction. Function `Func1` is supposed pure and `Proc2` has no side-effects. Note also that the test is invariant with respect to the inner loop.

```
      DO 40 i = 1,n
         DO 50 j = 1,m
            IF (test(i,i).ge.0) GOTO 50
            M1(i,j) = M2(i,j+1) + aa(j) + M1(i+1,j)
            M2(i,j) = Func1(3*M1(i+1,j-1) + bb(j,i))
50       CONTINUE
         CALL Proc2(test(i,i),i)
40    CONTINUE
```

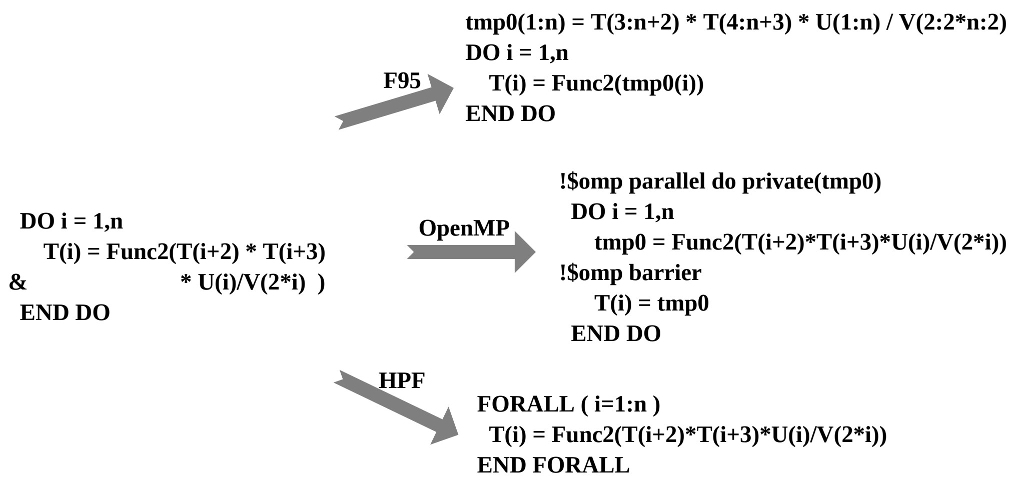
HPF

```
   FORALL ( i=1:n , test(i,i) < 0)
      tmp1(i,1:m) = 3*M1(i+1,0:m-1) + bb(1:m,i)
      M1(i,1:m) = M2(i,2:m+1) + aa(1:m) + M1(i+1,1:m)
   END FORALL
!HPF$ INDEPENDENT
   DO i = 1,n
      IF (test(i,i) < 0) THEN
!HPF$ INDEPENDENT
         DO j = 1,m
            M2(i,j) = Func1(tmp1(i,j))
         END DO
      ENDIF
      CALL Proc2(test(i,i),i)
   END DO
```

Figure 3.22: *Generation of* `FORALL` *and* `INDEPENDENT` *loops*

Changing the target changes dramatically the generated program: figure 3.23 shows the output of PARTITA for the three targets above. Function `Func2` is `PURE` in the HPF sense. This same example with target `f95light` would remain unmodified (sequential), because `f95light` is a less "agressive" parallelization which forbids instruction splitting.

Let us now illustrate other transformations automatically applied by PARTITA when they appear profitable. Figure 3.24 shows *invariant code motion*



```
 DO i = 1,n
    T(i) = Func2(T(i+2) * T(i+3)
&                     * U(i)/V(2*i)  )
 END DO
```

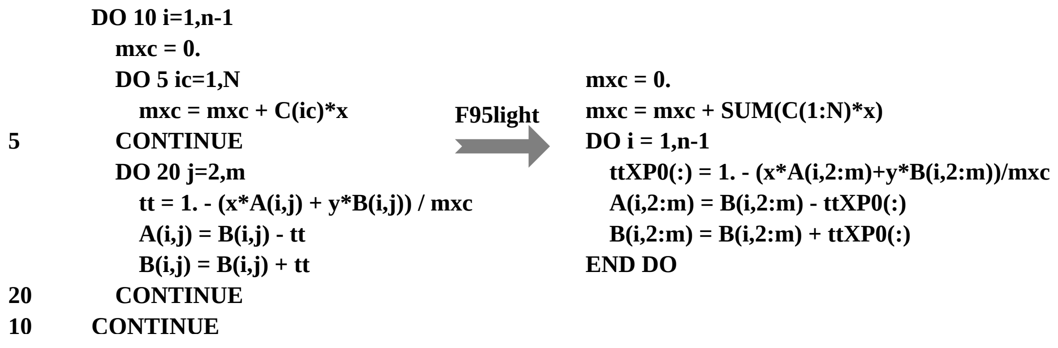
```
tmp0(1:n) = T(3:n+2) * T(4:n+3) * U(1:n) / V(2:2*n:2)
DO i = 1,n
   T(i) = Func2(tmp0(i))
END DO
```

```
!$omp parallel do private(tmp0)
  DO i = 1,n
     tmp0 = Func2(T(i+2)*T(i+3)*U(i)/V(2*i))
!$omp barrier
     T(i) = tmp0
  END DO
```

```
FORALL ( i=1:n )
 T(i) = Func2(T(i+2)*T(i+3)*U(i)/V(2*i))
END FORALL
```

Figure 3.23: *Results for different Target Descriptions*

of the first half of the loop, and *scalar expansion* to allow vectorization of the second half. The remaining sequential loop could be vectorized too, if expanding scalar `tt` along dimension `i`, but target `f95light` forbids two-dimensional expansions. On figure 3.25 another fragment takes profit of *in-*



```
      DO 10 i=1,n-1
         mxc = 0.
         DO 5 ic=1,N
            mxc = mxc + C(ic)*x
5        CONTINUE
         DO 20 j=2,m
            tt = 1. - (x*A(i,j) + y*B(i,j)) / mxc
            A(i,j) = B(i,j) - tt
            B(i,j) = B(i,j) + tt
20       CONTINUE
10    CONTINUE
```

```
mxc = 0.
mxc = mxc + SUM(C(1:N)*x)
DO i = 1,n-1
   ttXP0(:) = 1. - (x*A(i,2:m)+y*B(i,2:m))/mxc
   A(i,2:m) = B(i,2:m) - ttXP0(:)
   B(i,2:m) = B(i,2:m) + ttXP0(:)
END DO
```

Figure 3.24: *Invariant code motion, reduction, and scalar expansion*

variant code motion and detection of *induction variables*. Variables `l`, `m`, and `k`, have been found induction. Only the exit values of `l` and `k` are regenerated, because the other induction variables are not used after of the loop, nor do they appear in the formal arguments of the subroutine. When an induction variable is detected, its occurences in the program are not directly replaced by the induction value. Therefore if no parallelism is found, there is no need

```
      SUBROUTINE invar2(a,l,s,k,T)
      INTEGER a,l,m,s,i,j,k,T
      DIMENSION T(2000,2000)
C
      DO 720 i=2,1000
         s = 1
         DO 730 j=1,100
            l = a + i
            m = a + j
            T(l,m)=0
            s = s + j
            k = k + a
 730     CONTINUE
         k = k + 1
 720  CONTINUE
      END
```

$\xrightarrow{F95}$

```
SUBROUTINE invar2(a,l,s,k,T)
...
!
T(a+2:a+1000,a+1:a+100) = 0
l = a + 1000
s = 1
s = s + SUM((/ (j, j = 1,100) /))
k = k + 99900*a + 999
END SUBROUTINE invar2
```

Figure 3.25: *Induction variables, invariant code, and reduction*

to *undo* the expansion. Thus on figure 3.26 variable `k`, although inductive, is not expanded. Reductions are frquent and deserve some care. Figure 3.27

```
      DO 100 i=1,100
         T(k) = T(k-3) + 3
         k = k + 3
         C(i) = 0
 100  CONTINUE
```

$\xrightarrow{F95}$

```
DO i = 1,100
   T(k) = T(k-3) + 3
   k = k + 3
END DO
C(1:100) = 0
```

Figure 3.26: *Undoing unnecessary induction variable expansion*

shows a reduction that involves many instructions and tests. Directives are available to improve parallelization. On figure 3.28 parallelization is possible only because `offset` > 1 and array `IV` is injective.

3.4.6 A glimpse at the User Interface

The goal of the interface is to show which parallelization is chosen by PARTITA and why. The user can then modify the target or the program, e.g. with directives, to improve the result. There is a color code to display the parallelization status on the original code. Figure 3.29 shows this interface for a selected loop in a large application. As this report is printed in black and white, we have replaced the colors with patterns. Since we are not satisfied

```
DO i=1,N
   s1 = 3*C(i) + s1 - A(i)*B(i)
   IF (A(i).ge.10) THEN
      s1 = s1 - A(i)
   ENDIF
ENDDO
DO i=1,N
   s2 = s2 + B(i)
   s1 = s1 + B(i)*C(i)
   s2 = 2 - s2
ENDDO
DO i=20,80
   DO j=10,90
      p1 = p1 * D(i,j)
      p1 = p1 / E(j,i)
   ENDDO
ENDDO
```

$\xrightarrow{\text{F95}}$

```
s1 = s1 + SUM(3*C - A*B)
s1 = s1 - SUM(A, MASK = A>=10)
s1 = s1 + SUM(B*C)
DO i = 1,N
   s2 = s2 + B(i)
   s2 = s2 - 2
END DO
p1 = p1 * PRODUCT(D(20:80,10:90))
p1 = p1 / PRODUCT(E(10:90,20:80))
```

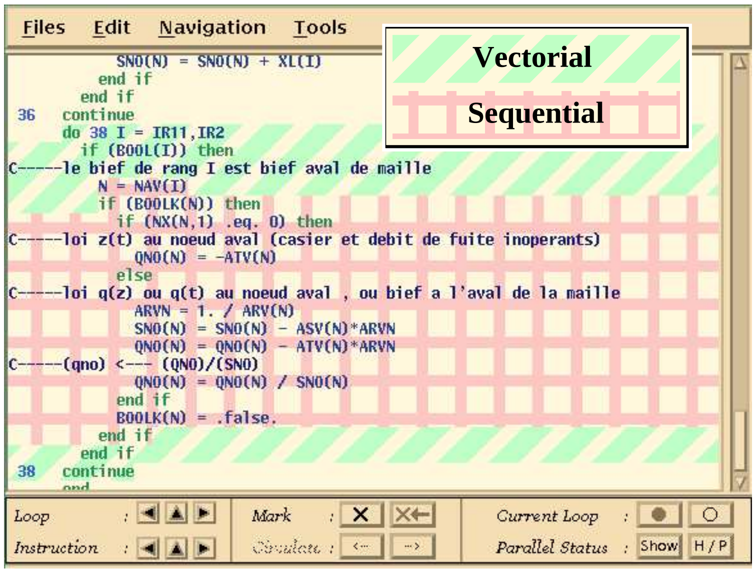
Figure 3.27: *Detection of Reduction Operations*

```
C$PRED (offset .gt. 1)
          DO 10 i=m1, m2
             A(i) = A(i+offset) + B(i)
             B(i-offset) = A(i) + B(i+offset)
 10       CONTINUE
C$INJECTIVE IV
          DO 20 i=m1,m2
             A(IV(i)) = B(IV(i))+ x*A(IV(i)+1)
             k=IV(i)
             B(k) = B(k)*y - A(k)
 20       CONTINUE
```

$\xrightarrow{\text{F95}}$

```
!$PRED (offset .gt. 1)
A(m1:m2) = A(m1+offset:m2+offset) + B(m1:m2)
B(m1-offset:m2-offset) =
                 A(m1:m2) + B(m1+offset:m2+offset)
!$INJECTIVE IV
DO i = m1,m2
   A(IV(i)) = B(IV(i)) + x*A(IV(i)+1)
END DO
B(IV) = B(IV)*y - A(IV)
```

Figure 3.28: *Use of directives by* PARTITA



```
           SNO(N) = SNO(N) + XL(I)
         end if
       end if
 36    continue
       do 38 I = IR11,IR2
         if (BOOL(I)) then
C-----le bief de rang I est bief aval de maille
           N = NAV(I)
           if (BOOLK(N)) then
             if (NX(N,1) .eq. 0) then
C-----loi z(t) au noeud aval (casier et debit de fuite inoperants)
               QNO(N) = -ATV(N)
             else
C-----loi q(z) ou q(t) au noeud aval , ou bief a l'aval de la maille
               ARVN = 1. / ARV(N)
               SNO(N) = SNO(N) - ASV(N)*ARVN
               QNO(N) = QNO(N) - ATV(N)*ARVN
C-----(qno) <--- (QNO)/(SNO)
               QNO(N) = QNO(N) / SNO(N)
             end if
             BOOLK(N) = .false.
           end if
         end if
 38    continue
       end
```

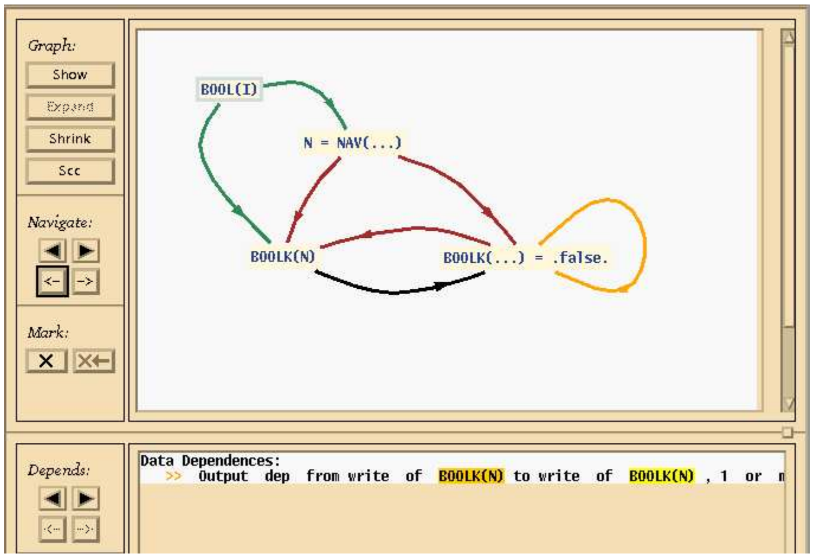
Figure 3.29: *Parallelization status display*

with the large part painted as "sequential", we look at the dependence graph for this loop. It comes in a separate window, as shown on figure 3.30, and exhibits the dependence cycle that causes non-parallelizability. The bottom of this window shows the precise data dependence information. From it we can find out that some data dependence are actually *false* dependences. Here we can remove them because we know that array `NAV` is `INJECTIVE`. Then the new parallelization status is much better, and we get the FORTRAN95 result in the window shown in figure 3.31.

3.5 Application 2: SPMD Parallelization

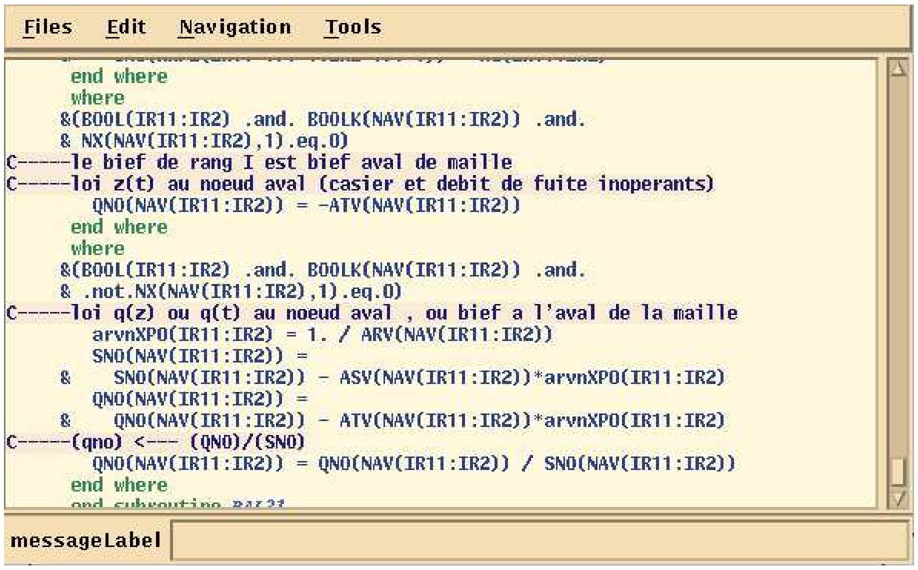
This section describes the use of the Dependence Graph in PARTITA, in order to provide a tool for Single Program Multiple Data (SPMD) parallelization of programs working on unstructured meshes. These programs are very frequent in numerical simulation. They are highly adapted to SPMD parallelization because they are very regular with respect to mesh elements (nodes, edges, triangles, tetrahedra, *etc*), and therefore a geometric partitioning of the mesh gives a very efficient parallelization.

We describe the SPMD parallelization approach and the goal of the tool



```
Data Dependences:
  >>  Output  dep  from write  of  BOOLK(N) to write  of  BOOLK(N) , 1  or
```

Figure 3.30: *Displayed dependence graph for selected loop*



```
      end where
      where
     &(BOOL(IR11:IR2) .and. BOOLK(NAV(IR11:IR2)) .and.
     & NX(NAV(IR11:IR2),1).eq.0)
C-----le bief de rang I est bief aval de maille
C-----loi z(t) au noeud aval (casier et debit de fuite inoperants)
        QNO(NAV(IR11:IR2)) = -ATV(NAV(IR11:IR2))
      end where
      where
     &(BOOL(IR11:IR2) .and. BOOLK(NAV(IR11:IR2)) .and.
     & .not.NX(NAV(IR11:IR2),1).eq.0)
C-----loi q(z) ou q(t) au noeud aval , ou bief a l'aval de la maille
        arvnXPO(IR11:IR2) = 1. / ARV(NAV(IR11:IR2))
        SNO(NAV(IR11:IR2)) =
     &    SNO(NAV(IR11:IR2)) - ASV(NAV(IR11:IR2))*arvnXPO(IR11:IR2)
        QNO(NAV(IR11:IR2)) =
     &    QNO(NAV(IR11:IR2)) - ATV(NAV(IR11:IR2))*arvnXPO(IR11:IR2)
C-----(qno) <--- (QNO)/(SNO)
        QNO(NAV(IR11:IR2)) = QNO(NAV(IR11:IR2)) / SNO(NAV(IR11:IR2))
      end where
```

Figure 3.31: *Generated* FORTRAN95 *program after directive insertion*

in section 3.5.1. Then in section 3.5.2 we formalize this problem in terms of a global analysis of the program's data dependence graph. We discuss implementation in section 3.5.3 and we terminate with a glimpse of the tool's user interface in section 3.5.4.

3.5.1 Objectives of the SPMD tool

As opposed to section 3.4, the SPMD parallelization approach only makes minimal changes to the source program. In some tools, it is necessary to introduce extra indirection arrays to reflect the partition of the mesh []. We believe this even is not necessary. In our approach, the only modification to the source program is the insertion of a handful of calls to communication routines. Each partitioned sub-mesh receives a new node numbering, which is standalone, and only communication routines need to know about the correspondence between sub-meshes boundaries. This knowledge is stored as "communication lists" that are built by the mesh partitioner. The remaining question is where to insert the communications between the sub-meshes into the original program. This decision is driven by the kind of *overlap* that was created on the boundary of the sub-meshes.

We must emphasize that this SPMD approach is restricted to one (large) class of procedures, that repeatedly perform *gather-scatter* loops and *global reductions* of arrays based on mesh elements:

- A gather-scatter loop is a loop on some sort of mesh entities (nodes, edges, triangles, *etc*), that access values that are stored on neighbor mesh entities, compute with these values, and store contributions back onto neighbor entities. These contributions are accumulated with a commutative-associative operation (generally +), so that the mesh elements may be swept in any order.

- A global reduction accumulates values that are computed on each mesh node, edge, triangle, *etc*, through a commutative-associative operation. This results in a scalar, which can be used for example to decide whether the computation has reached convergence or not.

Here is an example sketch of this kind of procedures:

```
New_Values = initialization
Repeat
    Old_Values = New_Values
    New_Values = 0
    Foreach Element ∈ Mesh
        gather the Old_Values from neighbors of Element
        compute the contribution of Element
        assemble into New_Values for neighbors of Element
    End Foreach
Until || New_Values − Old_Values || < $\varepsilon$
```

This SPMD approach relies on a geometrical partition of the mesh into sub-meshes, possibly with some amount of overlap on the boundaries. Partition itself is performed by specialized tools, which try to minimize the size of the interface between sub-meshes, called the boundaries. These tools use a variety of heuristics, such as the "recursive bissection" method. The amount of overlap itself (the *"overlapping pattern"*) is chosen by the end-user, and given as a constraint to the mesh partitioner. We show here two possible overlapping patterns, with one layer of overlapping **nodes** on figure 3.32, and with one layer of overlapping **triangles** on figure 3.33.

Let's take a closer look at communications between sub-meshes, for example on the partitioned mesh of figure 3.33. Node a, on the boundary of sub-mesh A, is duplicated as a' on sub-mesh B. We say that a is in the *kernel* of A, while a' is in the *overlap* of B. Suppose that just before execution of a gather-scatter loop, the `Old_Values` stored in a' are coherent with those in a. We can see on the figure that, during a gather-scatter loop on triangles, the values are correct for all triangles, even duplicated. Therefore the accumulated `New_Values` are correct for all kernel nodes such as a or b, but not for overlap nodes such as a' or b'. As a consequence, before the next gather-scatter loop, every overlap node must receive the correct value from its corresponding kernel node. This can be achieved by calling a single communication procedure from the original program. Figure 3.34 summarizes the complete parallelization process. Our tool focuses on the "Communications insertion" problem, to generates the SPMD source program.

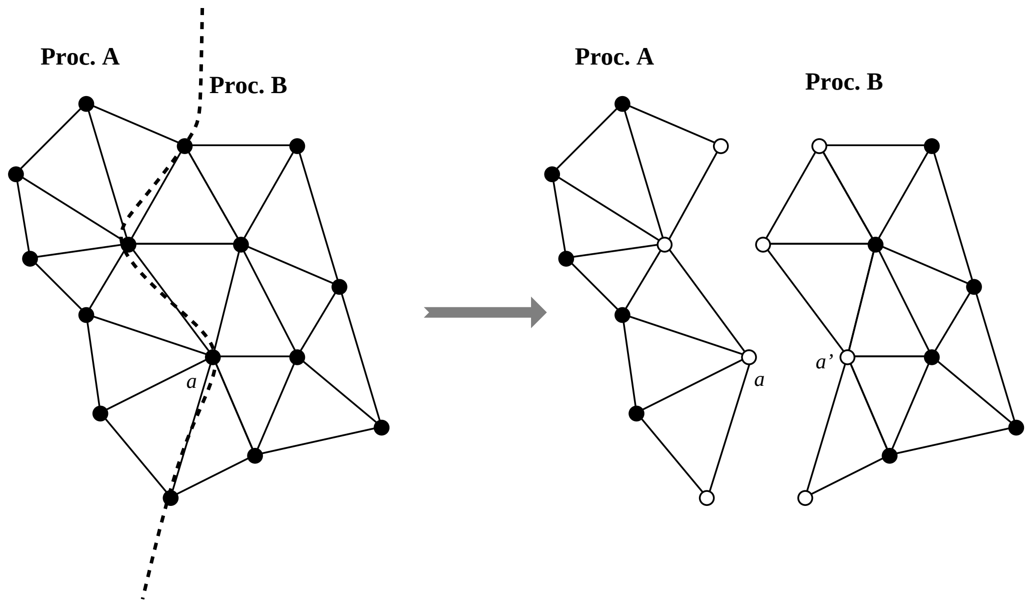


Figure 3.32: *Partition with one layer of overlapping nodes*

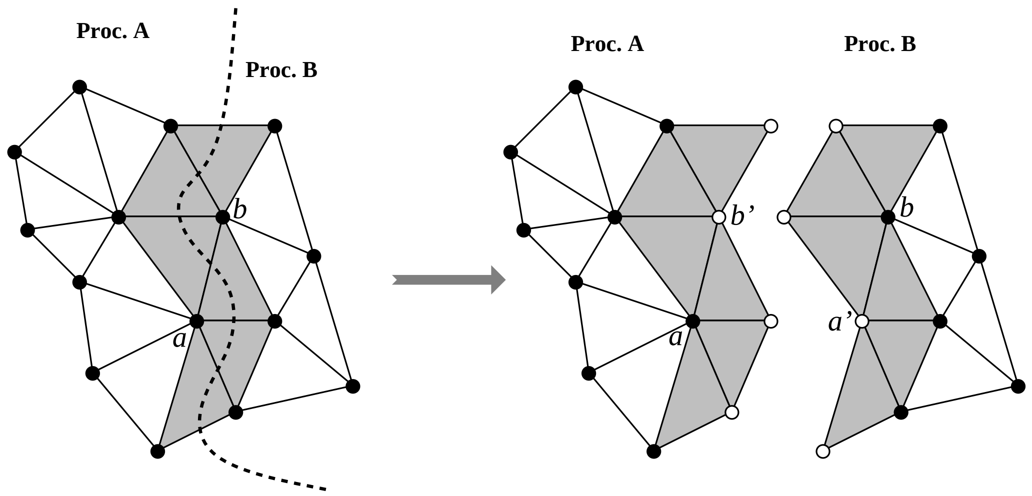


Figure 3.33: *Partition with one layer of overlapping triangles*

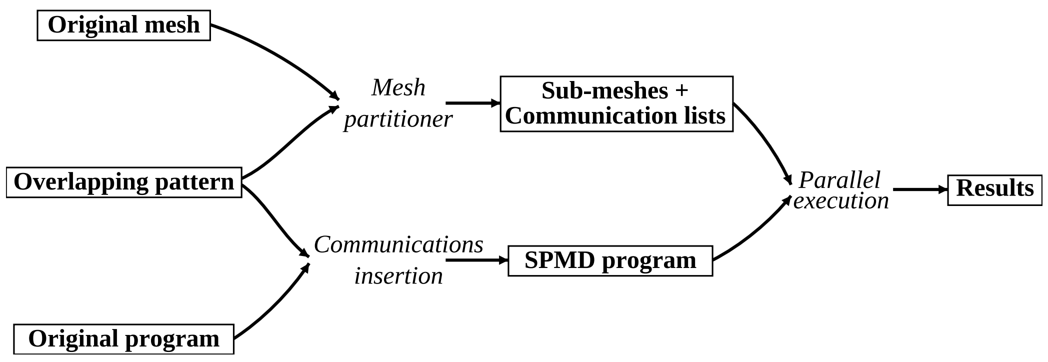


Figure 3.34: *General parallelization process*

3.5.2 Formalization of SPMD synchronization placement

To formalize the problem of insertion of synchronization calls, we introduce the notion of *state* of the values stored on mesh elements. For each array which is distributed on nodes, edges, triangles, *etc*, as well as for scalars resulting from **reduction** operations on these arrays, we define one *"good"* state, i.e. when values on the overlap nodes are coherent, or when the reduction is finally done on the complete mesh. We also define other *"worse"* states, when the values on the overlap nodes are out of date. When there are many layers of overlapping elements, there are many progressively worse and worse states, as the incoherence propagates from one overlap layer to another. We shall denote these states with the number of incoherent layers as an index. In our simple examples, this index can only be "1". Naturally, the coherent states are indexed with a "0". Therefore, for the overlapping pattern of figure 3.33, the possible states are the following:

$\mathbf{Tri}_0$, $\mathbf{Edg}_0$, $\mathbf{Edg}_1$, $\mathbf{Nod}_0$, $\mathbf{Nod}_1$, $\mathbf{Sca}_0$, and $\mathbf{Sca}_1$.

Notations **Tri**, **Edg**, and **Nod** refer to values distributed on triangles, edges, and nodes respectively, and **Sca** refers to a scalar resulting from a reduction. The index has the meaning stated above. We can see there is no such state as $\mathbf{Tri}_1$ for this overlapping pattern.

The idea is that the state of the values that flow through the program evolves. Across a gather-scatter loop, it evolves probably to a "worse" state, which is a consequence of the mesh topology. Conversely, it can evolve to a "better" state across a call to a communication routine that updates the overlap values. Figure 3.35 describes the evolution of the state across two sorts of gather-scatter loops: on the left a loop on mesh triangles, on the right

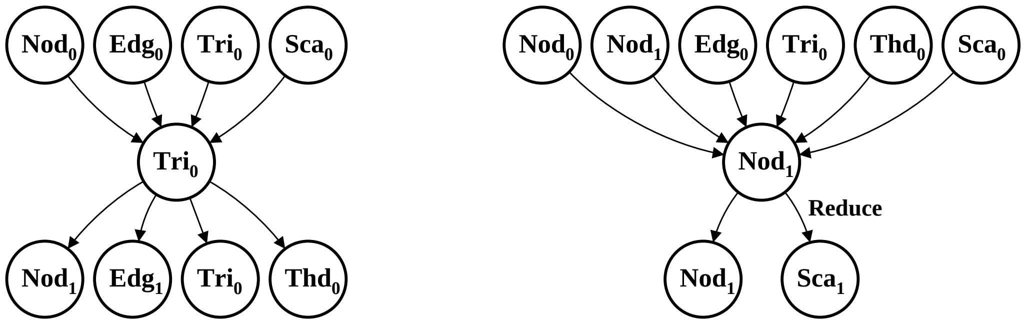


Figure 3.35: *State of the flowing data across loops on triangles and nodes*

a loop on mesh nodes, except the overlap nodes (a' or b'). These diagrams are built by hand, and follow from the mesh topology and the overlapping pattern. Conversely, there are additional "**Update**" transitions, that represent the updates of the overlap variables by appropriate communication calls. All these diagrams can be combined into a single *finite state automaton* that describes all the allowed evolutions of the state. Figure 3.36 shows this so-called *"Overlap Automaton"* for the overlapping pattern of figure 3.33. The Overlap Automaton is specific to each data organisation and its overlapping pattern. It is independent from the particular program, data, or partition. In principle, it must be provided by the user. However, in the very frequent case of 2D triangular meshes and 3D tetrahedron meshes, these automata are predefined for the most popular overlapping patterns. For example, figure 3.37 shows the Overlap Automata for, on the left a 2D mesh split like on figure 3.32, and on the right a 3D mesh split with one layer of overlapping tetrahedra.

A placement of the communications in the SPMD program is valid if the evolution of the state of data across all **flow** and **value** dependences of the program obeys the overlap automaton. More precisely, we must find a mapping M_n from the nodes of the Dependence Graph to the states of the Overlap Automaton, and M_a from the Data Dependences to the transitions of the Overlap Automaton, such that:

1. For every Dependence Graph node N of the program's input, $M_n(N)$ equals its given initial state.

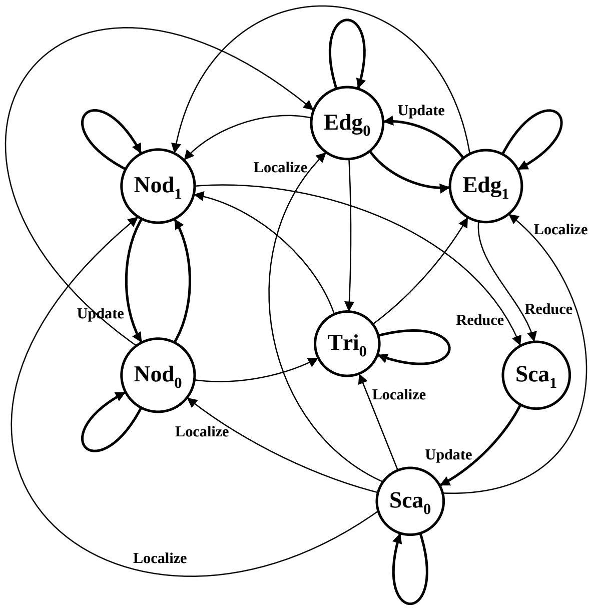
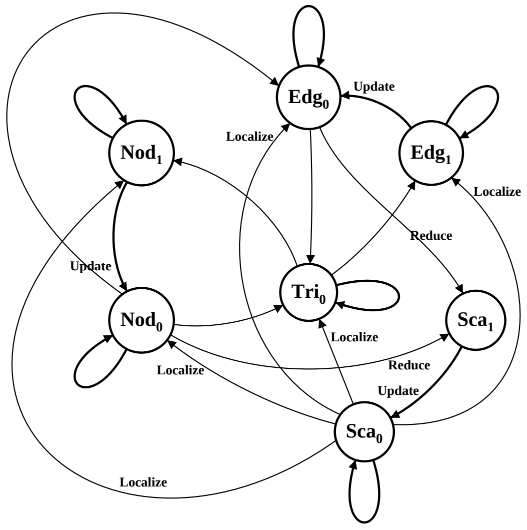


Figure 3.36: *Overlap Automaton for 2D mesh split with overlapping triangles*



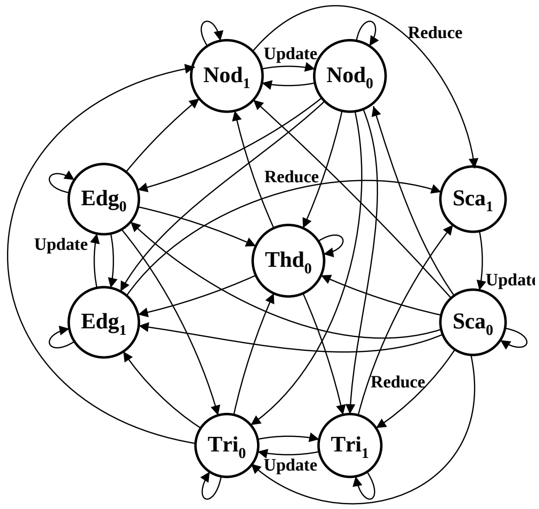


Figure 3.37: *Other Overlap Automata for 2D and 3D meshes*

2. For every Dependence Graph node N of the program's output, $M_n(N)$ equals its required final state.

3. For every **flow** and **value** Data Dependence A in the program,

$$origin(M_a(A)) = M_n(origin(A))$$

$$destination(M_a(A)) = M_n(destination(A))$$

From this mapping, we deduce how to build the SPMD parallelized program:

1. Each Data Dependence mapped to a transition labelled "**Update**", requires a communication between its origin and its destination.

2. Each loop mapped to a state with subscript 1, must not be run on duplicated mesh elements.

3.5.3 Implementation and Results

We implemented the algorithm above inside the FORESYS environment. We took profit of the Dependence Graph already available for the PARTITA parallelization tool. Finding a mapping of the Dependence Graph on the Overlap Automaton is a classical problem. However, the search for the mapping is exponential with respect to the size of the Dependence Graph, which is large. Moreover, there often exist many solution mappings, and the search algorithm must implement **backtracking** to find them all. Therefore, we implemented a heuristic that uses Dependence Graphs of recursively nested parts of the program. We run the mappings detection on each part, bottom-up. Each part is analyzed to find *all* possible mappings. Then only the "best" mappings are kept and used at the enclosing level. This might miss some solutions in very rare cases, but experience shows there is no problem in practice. Each solution is given an approxiamte cost, based on elementary costs for communications and also on the overhead for executing a gather-scatter loop on the overlap mesh elements. Finally, only a few "best" solutions are proposed to the end-user.

Nevertheless, this implemented algorithm is rather slow. On our largest application example (*NSC3DM*, 34 routines, 4800 code lines), it runs for approximately 1 hour. We now believe that this implementation based on the Dependence Graph is good for its close link with the formalization, but is inefficient because

- the Dependence Graph is very large,
- we don't need the fine grain rescheduling abilities of the Dependence Graph, since we don't change the Flow Graph of the program,
- going back from data dependences labelled "Update" to actual locations in the program is difficult.

Actually, all we need is a data flow analysis, that would propagate the states of all variables through the Flow Graph of the program. This analysis would be far cheaper than what we have implemented. Also, backtracking should be limited further.

Let's now look at the performances of the SPMD programs. Our application examples are two-dimensional and three-dimensional flow solvers, written in Fortran 77, named *Exp2D* (13 routines, 1500 code lines), *Imp2D* (18 routines, 1800 code lines), and *NSC3DM* (34 routines, 4800 code lines). They solve the compressible Navier-Stokes equations on unstructured triangular meshes. Time advancing is *explicit* for *Exp2D*, and *implicit* for *Imp2D* and *NSC3DM*, therefore requiring the resolution of a large sparse linear system at each time step. This is done with Jacobi relaxations. The partitioned meshes have one layer of duplicated triangles (*resp.* tetrahedra) in 2D (*resp.* 3D). This choice is justified by the comparison of various overlapping patterns given in [18]. Each application comes with its test case:

- *Imp2D* computes the external laminar viscous flow around a `NACA0012` airfoil (Mach number: 0.85, Reynolds number: 2000). The 2D mesh contains 48792 nodes, 96896 triangles and 145688 edges.
- *NSC3DM* computes the external flow around an `ONERA M6` wing (Mach number: 0.84, angle of incidence: 3.06°). The 3D mesh contains 15460 nodes, 80424 tetrahedra and 99891 edges.

Although this is more the result of the SMPD tactic itself, rather than a result of our method to insert communications, we show some performance results of the SPMD programs obtained. More can be found in [18]. In the following tables, N_p is the number of processes for the parallel execution; "Elapsed" is the average total elapsed execution time and "CPU" the total CPU time; the parallel speedup $S(N_p)$ is calculated on the elapsed times. Simulations are performed with 64 bit arithmetic computations, on the following computing platforms:

- *(cf table 3.1)* a SGI Origin 2000 (located at the *Centre Charles Hermite* in Nancy), equipped with Mips R10000/195 Mhz processors with 4 Mb of cache memory. The SGI implementation of MPI was used.

	Imp2D on NACA0012 airfoil			*NSC3DM* on ONERA M6 wing		
N_p	Elapsed	CPU	$S(N_p)$	Elapsed	CPU	$S(N_p)$
1	2793 s	2773 s	**1.0**	1318 s	1305 s	**1.0**
4	815 s	809 s	**3.5**	390 s	387 s	**3.4**
8	314 s	310 s	**8.9**	198 s	196 s	**6.7**
16	147 s	145 s	**19.0**			

Table 3.1: Performances on a SGI Origin 2000 system

- *(cf table 3.2)* a cluster of 12 Pentium Pro P6/200 Mhz, with a 100 Mbit/s FastEthernet switch. The code was compiled by the G77 GNU compiler with maximal optimization options. Communications used MPICH (version 1.1).

	Imp2D on NACA0012 airfoil			*NSC3DM* on ONERA M6 wing		
N_p	Elapsed	CPU	$S(N_p)$	Elapsed	CPU	$S(N_p)$
1	6883 s	6854 s	**1.0**	7488 s	7429 s	**1.0**
4	2289 s	2182 s	**3.0**	2583 s	2382 s	**2.9**
8	1440 s	1218 s	**4.8**	1409 s	1254 s	**5.3**

Table 3.2: Performances on a cluster of 12 Pentium Pro

The results show a good parallel speedup, especially on table 3.1. The decreasing sizes of the sub-meshes resulting from the 8 and 16 sub-meshes decomposition explain the super-linear speed-up observed on the SGI Origin 2000: on this platform the R10000 processor possesses a 4 Mb cache memory, a rather large value that contributes to high computational rates. On table 3.2 the speedups are still good, but show the impact of higher communication costs prevailing on the cluster architecture.

3.5.4 A glimpse at the User Interface

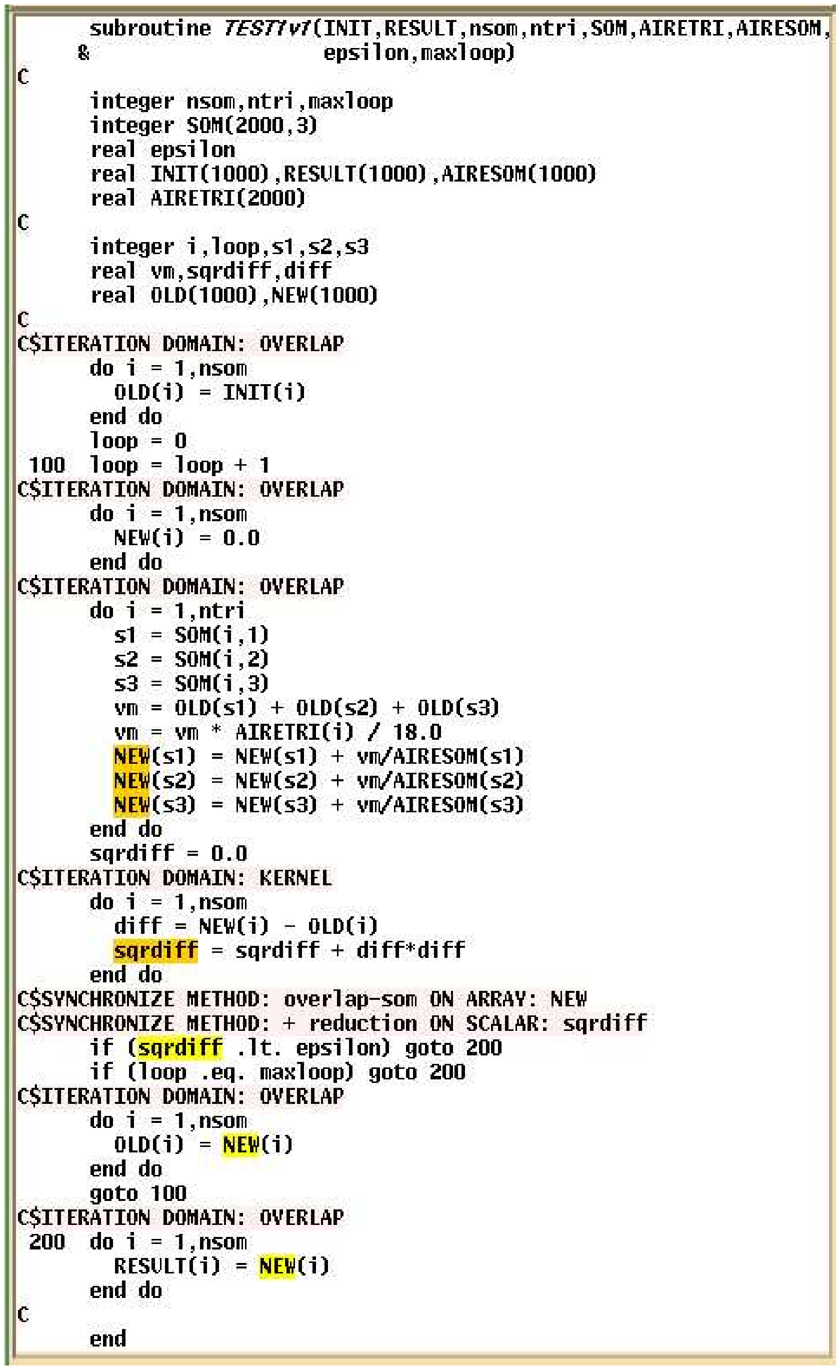
SPMD parallelization must be interactive. The user must tell which are the arrays that are distributed on the mesh entities, as well as the selected overlapping pattern. So we devised a specialized interface to tell:

- the program fragment to parallelise,
- the Overlap Automaton corresponding to the current geometry of the mesh and of the overlap,
- an assignment from each variable name and each loop, to the set of its possible states. This set specifies that each of these objects (variables and loops) is *aligned* once and for all on one category of mesh elements (nodes, edges, triangles, ...), and is partitioned accordingly.

Since we have the Dependence Graph already at hand at this stage, telling the alignment of each variable and loop is not too tedious. There exist rules to deduce these alignments from others already given. For example, given a loop aligned on, say, triangles, we know that all arrays accessed directly with the loop index are also aligned on triangles. On our largest application example (*NSC3DM*, 34 routines, 4800 code lines) only a dozen of clicks were required.

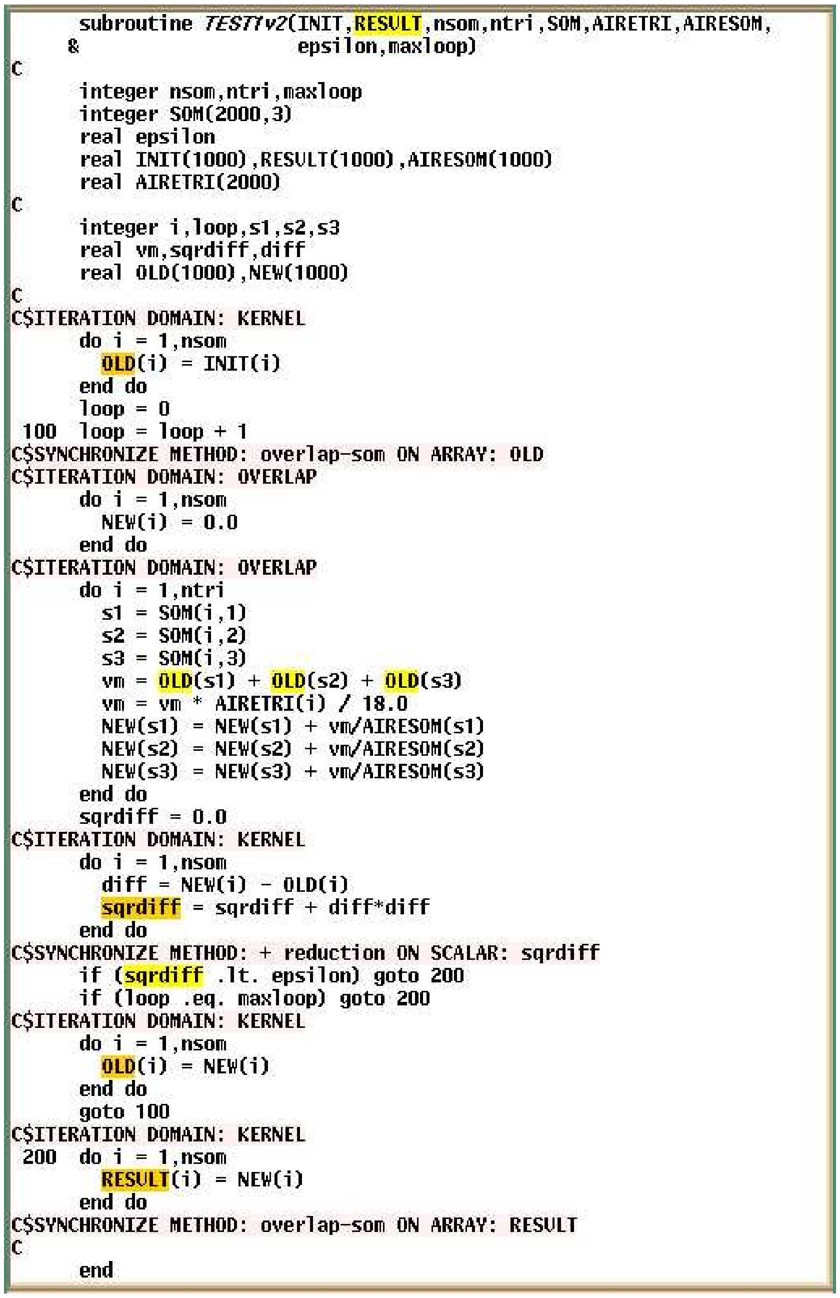
Then follows a verification stage, that checks that all loops in the fragment are really parallelizable. There must be no loop-carried dependences, except for the reductions corresponding to the *scatter* phase.

Then comes the search for a mapping, and the resulting SPMD programs are built. Figure 3.38 shows one solution for a small example program. Notice the comment lines inserted where communication is required. Also, both ends of the data dependences labelled "**Update**" are highlighted, to explain why this communication is required. Each loop on mesh elements is also annotated to show whether it must run on the sub-mesh kernel only, or also on the overlap. Figure 3.39 shows another solution, which is neither better nor worse than the first solution. The choice is up to the user. Both solutions set basically the same communications, but the solution on figure 3.39 has the advantage of grouping the two main communications, thereby saving an additional communication overhead. On the other hand, the solution on figure 3.38 delays one communication so that the iteration space of some loops may be restricted to the kernel nodes, saving some instructions on the overlap.



```
      subroutine TEST1v1(INIT,RESULT,nsom,ntri,SOM,AIRETRI,AIRESOM,
     &                    epsilon,maxloop)
C
      integer nsom,ntri,maxloop
      integer SOM(2000,3)
      real epsilon
      real INIT(1000),RESULT(1000),AIRESOM(1000)
      real AIRETRI(2000)
C
      integer i,loop,s1,s2,s3
      real vm,sqrdiff,diff
      real OLD(1000),NEW(1000)
C
C$ITERATION DOMAIN: OVERLAP
      do i = 1,nsom
        OLD(i) = INIT(i)
      end do
      loop = 0
 100  loop = loop + 1
C$ITERATION DOMAIN: OVERLAP
      do i = 1,nsom
        NEW(i) = 0.0
      end do
C$ITERATION DOMAIN: OVERLAP
      do i = 1,ntri
        s1 = SOM(i,1)
        s2 = SOM(i,2)
        s3 = SOM(i,3)
        vm = OLD(s1) + OLD(s2) + OLD(s3)
        vm = vm * AIRETRI(i) / 18.0
        NEW(s1) = NEW(s1) + vm/AIRESOM(s1)
        NEW(s2) = NEW(s2) + vm/AIRESOM(s2)
        NEW(s3) = NEW(s3) + vm/AIRESOM(s3)
      end do
      sqrdiff = 0.0
C$ITERATION DOMAIN: KERNEL
      do i = 1,nsom
        diff = NEW(i) - OLD(i)
        sqrdiff = sqrdiff + diff*diff
      end do
C$SYNCHRONIZE METHOD: overlap-som ON ARRAY: NEW
C$SYNCHRONIZE METHOD: + reduction ON SCALAR: sqrdiff
      if (sqrdiff .lt. epsilon) goto 200
      if (loop .eq. maxloop) goto 200
C$ITERATION DOMAIN: OVERLAP
      do i = 1,nsom
        OLD(i) = NEW(i)
      end do
      goto 100
C$ITERATION DOMAIN: OVERLAP
 200  do i = 1,nsom
        RESULT(i) = NEW(i)
      end do
C
      end
```

Figure 3.38: *One possible generated SPMD program*



```
      subroutine TEST1v2(INIT,RESULT,nsom,ntri,SOM,AIRETRI,AIRESOM,
     &                    epsilon,maxloop)
C
      integer nsom,ntri,maxloop
      integer SOM(2000,3)
      real epsilon
      real INIT(1000),RESULT(1000),AIRESOM(1000)
      real AIRETRI(2000)
C
      integer i,loop,s1,s2,s3
      real vm,sqrdiff,diff
      real OLD(1000),NEW(1000)
C
C$ITERATION DOMAIN: KERNEL
      do i = 1,nsom
        OLD(i) = INIT(i)
      end do
      loop = 0
 100  loop = loop + 1
C$SYNCHRONIZE METHOD: overlap-som ON ARRAY: OLD
C$ITERATION DOMAIN: OVERLAP
      do i = 1,nsom
        NEW(i) = 0.0
      end do
C$ITERATION DOMAIN: OVERLAP
      do i = 1,ntri
        s1 = SOM(i,1)
        s2 = SOM(i,2)
        s3 = SOM(i,3)
        vm = OLD(s1) + OLD(s2) + OLD(s3)
        vm = vm * AIRETRI(i) / 18.0
        NEW(s1) = NEW(s1) + vm/AIRESOM(s1)
        NEW(s2) = NEW(s2) + vm/AIRESOM(s2)
        NEW(s3) = NEW(s3) + vm/AIRESOM(s3)
      end do
      sqrdiff = 0.0
C$ITERATION DOMAIN: KERNEL
      do i = 1,nsom
        diff = NEW(i) - OLD(i)
        sqrdiff = sqrdiff + diff*diff
      end do
C$SYNCHRONIZE METHOD: + reduction ON SCALAR: sqrdiff
      if (sqrdiff .lt. epsilon) goto 200
      if (loop .eq. maxloop) goto 200
C$ITERATION DOMAIN: KERNEL
      do i = 1,nsom
        OLD(i) = NEW(i)
      end do
      goto 100
C$ITERATION DOMAIN: KERNEL
 200  do i = 1,nsom
        RESULT(i) = NEW(i)
      end do
C$SYNCHRONIZE METHOD: overlap-som ON ARRAY: RESULT
C
      end
```

Figure 3.39: *Another possible generated SPMD program*

Chapter 4

Automatic Differentiation: Presentation, Techniques, Tools and Applications

This chapter is devoted to Automatic Differentiation (AD), an innovative and very interesting static analysis and transformation of programs. Unlike parallelization, AD does not preserve the semantics of program, but rather adds new program outputs which are mathematical derivatives of the original results. We started the reseach presented here around 1998, at INRIA first in the SINUS project-team, then in the new TROPICS project-team. This chapter has two main goals.

- One goal is to emphasize how the classical techniques from compilation and parallelization transpose to AD, in order to get algorithms that are better understood and more efficient.
- The other goal is to present some research results specific to AD, about internal algorithms and also about application.

This distinction between the two goals above is mostly rhetoric, since the research results are often based on compilation theory, and thus the two goals are linked strongly.

Since AD is not as well known as parallellization, section 4.1 is necessary to present its principles. Also, some examples of application are necessary to justify this complex transformation. This is done in section 4.2, which presents two proeminent application fields: inverse problems and gradient-based optimization. Then, following the same plan as in the previous chapter, we discuss some advances in AD technology that we brought from compiler theory and methods. Thus section 4.3 shows that the Flow Graph internal representation brings a cleaner and more efficient inversion of the flow of control for the AD *reverse mode*. Next section 4.4 presents *activity analysis* and the new *TBR analysis* in terms of data-flow equations that help prove interesting properties. It also shows application of data-flow analyses to get optimal *snapshots* and to find AD-specific dead code. Next section 4.5 gathers results based on data-dependence analysis for rescheduling of differentiated instructions and for a study of the data dependence graph of *adjoint* programs. Last but not least, section 4.6 presents our TAPENADE AD tool, which incorporates and validates our AD research, and which is used on many real programs, including industrial applications.

4.1 Automatic Differentiation of Programs

Automatic Differentiation (AD) is a technique to evaluate derivatives of a function $f : X \in I\!R^m \mapsto Y \in I\!R^n$ defined by a computer program `P`. In AD, the original program is automatically transformed into a new program `P'` that computes the derivatives *analytically*. For reference, we recommend the monography [24], selected articles of a recent conference [11], or the AD community website at `www.autodiff.org`.

The goal of AD is to compute derivatives without going back to the underlying mathematical equations, considering only the source program `P`. This will spare a tedious extra discretization and implementation phase. Moreover, sometimes the mathematical equations are not available. How can we reach this goal? Naturally, one can do *divided differences*. For a given set of program's inputs X, program `P` computes a result Y. Given now some normalized direction dX in the space of the inputs, one can run program `P` again on the new set of inputs $X + \varepsilon.dX$, where ε is some very small positive number. Divided Differences return an *approximation* of the derivative by the formula:

$$\frac{\mathtt{P}(X + \varepsilon.dX) - \mathtt{P}(X)}{\varepsilon}.$$

The *centered* divided differences give a better approximation, at the cost of an extra run of `P` by the formula:

$$\frac{\mathtt{P}(X+\varepsilon.dX)-\mathtt{P}(X-\varepsilon.dX)}{2\varepsilon}.$$

In any case, these are just approximations of the derivatives. Ideally, the exact derivative is the limit of these formulas when ε tends to zero. But this makes no sense on a real computer, since very small values of ε lead to truncation errors, and therefore to erroneous derivatives. This is the main drawback of divided differences: some tradeoff must be found between truncation errors and approximation errors. Finding the best ε requires numerous executions of the program, and even then the computed derivatives are just approximations.

To get rid of approximation errors, AD computes derivatives **analytically**. Each time the original program holds some variable `v`, the differentiated program holds an additional variable with the same shape, that we call the *differential* of `v`. Moreover, for each operation in the original program, the differentiated program performs additional operations dealing with the differential variables. For example, suppose that the original program comes to executing the following instruction on variables `a`, `b`, `c`, and array `T`:

$$\mathtt{a = b*T(10) + c} \tag{4.1}$$

Suppose also that variables $\dot{\mathtt{b}}$, $\dot{\mathtt{c}}$, and array $\dot{\mathtt{T}}$ are available and contain one particular sort of differential: the tangent derivatives, i.e. the first-order variation of `b`, `c`, and `T` for a given variation of the input. Then the differentiated program must execute additional operations that compute $\dot{\mathtt{a}}$, using `b`, `c`, `T` and their differentials $\dot{\mathtt{b}}$, $\dot{\mathtt{c}}$, and $\dot{\mathtt{T}}$. These must somehow amount to:

$$\dot{\mathtt{a}}\ \mathtt{=}\ \dot{\mathtt{b}}\mathtt{*T(10) + b*}\dot{\mathtt{T}}\mathtt{(10) +}\ \dot{\mathtt{c}} \tag{4.2}$$

The derivatives are computed analytically, using the well known formulas on derivation of elementary operations. Approximation errors, which are a nuisance when using Divided Differences, have just vanished.

At this point, let us mention an elegant manner to implement AD: *overloading.* Overloading is a programming technique, available in several languages, where one can redefine the semantics of basic functions (such as arithmetic operations), according to the type of their arguments. For example, instruction (4.1) can easily subsume instruction (4.2), if only the type of variables is changed from `REAL` to pairs of `REAL`, and the semantics of `+`, `*`, etc, are augmented to compute the derivatives into, say, the second component of the above pairs of `REAL`. Even when the language does not support overloading, there is an elegant workaround [36]: just replace all `REAL`'s by `COMPLEX` numbers, and put their derivative into the imaginary part! It is easy to check that the arithmetic of `COMPLEX` numbers can be a good approximation of overloaded operations on derivatives. The advantage of overloading is that it requires very little code transformation. Drawbacks are that not all languages support overloading (or `COMPLEX` numbers), that overloaded programs are poorly optimized by the compiler, and more importantly that overloading is not suitable for the *reverse* mode of AD (*cf* section 4.1.3). Therfore, we rejected overloading for our purpose, and we preferred **source transformation** techniques, using technology from compilation and parallelization.

Automatic Differentiation can compute many different sorts of derivatives. At least in theory, it can yield a complete Jacobian matrix, i.e. the partial derivatives of each output with respect to each input. Higher order differentiation is possible too, e.g. computation of Hessian tensors. In practice, these mathematical objects are often too large or too expensive to compute. Therefore, AD can also return smaller objects at a cheaper cost, e.g directional derivatives, gradients, directional higher-order derivatives, or Taylor series expansions. Similar techniques apply to domains slightly outside of strict differentiation: AD techniques are used to build programs that compute on *intervals* or on *probabilistic* data. We believe the most promising of these "*modes*" is the computation of *gradients*, whereas computation of first-order directional derivatives (called *tangents*) is the most straightforward, and is therefore an interesting basic mode. In the remainder, we shall focus on these two modes of AD. Before that, the next section will introduce the general formal framework used to describe AD.

4.1.1 Computer Programs and Mathematical Functions

Remember that we want to compute exact derivatives analytically, based on a given computer program `P` which is seen as the principal *specification* of the mathematical function f to differentiate. Therefore, we need to introduce a general framework in which programs can be identified with functions. We first identify programs with sequences of instructions, identified in turn with composed functions.

Programs contain *control*, which is essentially discrete and therefore non-differentiable. Consider the set of all possible run-time sequences of instructions. Of course there are a lot (often an infinity!) of such sequences, and therefore we never build them explicitly! The control is just how one tells the running program to switch to one sequence or another. For example this small `C` program piece:

```
if (x <= 1.0)
  printf("x too small");
else {
  y = 1.0;
  while (y <= 10.0) {
     y = y*x;
     x = x+0.5;
  }
}
```

will execute according to the control as one of the following *sequences of instructions*:

```
printf("x too small");
y = 1.0;
y = 1.0; y = y*x; x = x+0.5;
y = 1.0; y = y*x; x = x+0.5; y = y*x; x = x+0.5;
y = 1.0; y = y*x; x = x+0.5; y = y*x; x = x+0.5; y = y*x; x = x+0.5;
```

and so on... Each of these sequences is differentiable. The new program generated by Automatic Differentiation uses the original program's control to guarantee it computes the differentials of the actual run-time *sequence of instructions*. This is only **piecewise** differentiation. Thus, this differentiated program will probably look like:

```
if (x <= 1.0)
  printf("x too small");
else {
  dy = 0.0;
  y = 1.0;
  while (y <= 10.0) {
     dy = dy*x + y*dx;
     y = y*x;
     x = x+0.5;
  }
}
```

However it sometimes happens, like in this example, that the control itself depends on differentiated variables. In that case, a small change of the initial values may result in a change of the control. Here, a small change of `x` may change the number of iterations of the while loop, and the derivative is not defined any more. Yet the new program generated by Automatic Differentiation will return a result, and using this derivative may lead to errors. In other words, the original program, with control, is only piecewise differentiable, and "state of the art" AD does not take this into account correctly. This is an open research problem. In the meantime, we simply assume that this problem happens rarely enough. Experience on real programs shows that this is a reasonable assumption. However, this problem is widely known and it limits the confidence end-users place into AD.

Now that programs are identified with sequences of instructions, these sequences are identified with composed functions. Precisely, the sequence of instructions :

$$\mathtt{I_1; I_2; ... I_{p-1}; I_p;}$$

is identified to the function :

$$f = f_p \circ f_{p-1} \circ \ldots \circ f_1$$

Each of these functions is naturally extended to operate on the domain of all the program variables: variables not overwritten by the instruction are just transmitted unchanged to the function's result. We can then use the chain rule to write formally the derivative of the program for a given input X :

$$\begin{array}{rcl} f'(X) & = & (f'_p \circ f_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(X)) \\ & & .\ (f'_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(X)) \\ & & .\ \ldots \\ & & .\ (f'_1(X)) \\ & = & f'_p(X_{p-1})\ .\ f'_{p-1}(X_{p-2})\ .\ \ldots\ .\ f'_1(X_0) \end{array} \tag{4.3}$$

Each f'_k, derivative of function f_k, is a Jacobian matrix. For short, we defined $X_0 = X$ and $X_k = f_k(X_{k-1})$ to be the values of the variables just after executing the first k instructions. Computing the derivatives is just computing and multiplying the elementary Jacobian matrices $f'_k(X_{k-1})$.

Let us conclude this section with some bad news: for average size applications, the Jacobian matrix $f'(X)$ is often too large! Therefore it is not reasonable to compute it explicitly: the matrix-matrix products would be too expensive, and the resulting matrices would be too large to be stored. Except in special cases, one must not compute $f'(X)$. Fortunately, many uses of derivatives do not need $f'(X)$, but only a "view" of it, such as $f'(X).\dot{X}$ for a given $\dot{X}$ or $f'^t(X).\overline{Y}$ for a given $\overline{Y}$. These two cases will be discussed in the next two sections.

4.1.2 The Tangent Mode of Automatic Differentiation

For some applications, what is needed is the so-called *sensitivity* of a program. For a given small variation $\dot{X}$ in the input space, we want the corresponding first-order variation of the output, or "sensitivity". By definition of the Jacobian matrix, this sensitivity is $\dot{Y} = f'(X).\dot{X}$. Historically, this is the first application of AD, probably because it is the easiest.

Recalling equation (4.3), we get:

$$\dot{Y} = f'_p(X_{p-1})\ .\ f'_{p-1}(X_{p-2})\ .\ \ldots\ .\ f'_1(X_0)\ .\ \dot{X} \tag{4.4}$$

To compute $\dot{Y}$ efficiently, one must of course do it from right to left, because Matrix$\times$Vector products are so much cheaper than Matrix$\times$Matrix products. This turns out to be easy, because this formula requires X_0 first, and then X_1, and so on until X_{p-1}. In other words, the intermediate values from the original program `P` are used as they are computed. Differentiated instructions, that compute the Jacobian matrices and multiply them, can be done along with the initial program. We only need to interleave the original instructions and the derivative instructions.

In the tangent mode, the differentiated program is just a copy of the given program, Additional derivative instructions are inserted just before each instruction. The control structures of the program are unchanged, i.e. the Call Graph and Flow Graph have the same shape in `P` and `P'`. Figure 4.1 illustrates AD in tangent mode. On the left columm is an example subroutine, that measures a sort of discrepancy between two given arrays `T` and `U`. The right column shows the tangent differentiated subroutine, which computes tangent derivatives, conventionally shown with a *dot* above. For example, the differentiated instruction that precedes instruction `e = SQRT(e2)`, implements the following vector assignment that multiplies the instruction's elementary Jacobian by the vector of tangent derivatives:

$$\begin{bmatrix}\dot{\mathtt{e2}}\\ \dot{\mathtt{e}}\end{bmatrix} = \begin{bmatrix}1 & 0\\ 0.5/\sqrt{\mathtt{e2}} & 0\end{bmatrix}\cdot\begin{bmatrix}\dot{\mathtt{e2}}\\ \dot{\mathtt{e}}\end{bmatrix}$$

original: $\mathtt{T,U} \mapsto \mathtt{e}$	**tangent mode:** $\mathtt{T},\dot{\mathtt{T}},\mathtt{U},\dot{\mathtt{U}} \mapsto \mathtt{e},\dot{\mathtt{e}}$
	$\dot{\mathtt{e2}}$ `= 0.0`
`e2 = 0.0`	`e2 = 0.0`
`do i=1,n`	`do i=1,n`
	$\dot{\mathtt{e1}}$ `=` $\dot{\mathtt{T}}$`(i)-`$\dot{\mathtt{U}}$`(i)`
`e1 = T(i)-U(i)`	`e1 = T(i)-U(i)`
	$\dot{\mathtt{e2}}$ `=` $\dot{\mathtt{e2}}$ `+ 2.0*e1*`$\dot{\mathtt{e1}}$
`e2 = e2 + e1*e1`	`e2 = e2 + e1*e1`
`end do`	`end do`
	$\dot{\mathtt{e}}$ `= 0.5*`$\dot{\mathtt{e2}}$`/SQRT(e2)`
`e = SQRT(e2)`	`e = SQRT(e2)`

Figure 4.1: *AD in tangent mode*

The tangent mode of AD is rather straightforward. However, it can benefit from some specific optimizations based on static analysis of the program. In particular *activity* analysis (*cf* section 4.4.1) can vastly simplify the tangent differentiated program, by statically detecting derivatives that are always zero.

4.1.3 The Reverse Mode of Automatic Differentiation

Maybe the most promising application of AD is the computation of *gradients* of a program. If there is only one scalar output $f(X) = y$ ($n = 1$), the Jacobian $f'(X)$ has only one row, and it is called the *gradient* of f at point X. If $n > 1$, one can go back to the first case by defining a *weighting* $\overline{Y}$, such that $< f(X).\overline{Y} >$ is now a scalar. If the components of $f(X)$ were some kind of optimization criteria, this amounts to defining a single optimization criterion $< f(X).\overline{Y} >$, whose gradient at point X can be computed. By definition of the Jacobian matrix, this gradient is $\overline{X} = f'^t(X).\overline{Y}$. This gradient is useful in optimization problems (*cf* section 4.2), because it gives a descent direction in the input space, used to find the optimum.

Recalling equation (4.3), we get:

$$\overline{X} = f_1'^t(X_0) \;.\; f_2'^t(X_1) \;.\; \ldots \;.\; f_{p-1}'^t(X_{p-2}) \;.\; f_p'^t(X_{p-1}) \;.\; \overline{Y} \tag{4.5}$$

Here also, efficient computation of equation (4.5) must be done from right to left. In this case, one can show that the computation cost of this gradient is only a small multiple of the computation cost of the original function f by `P`. In contrast, the program $\dot{\texttt{P}}$ resulting from the *tangent mode* of AD, which also costs a small multiple of `P`'s execution time, returns only a column of the Jacobian Matrix. To obtain the gradient, one must run $\dot{\texttt{P}}$ once for each element of the cartesian basis of the input space. Therefore the cost of computing the gradient using the tangent mode is proportional to the dimension of the input space. Similarly, to compute the gradient using Divided Differences also requires a number of evaluations of `P` proportional to the dimension of the input space. For each basis direction e_i in the input space, i.e. for each component of vector X, one must run `P` at least once for $X + \varepsilon.e_i$, and in fact more than once to find a good enough ε. Thus the reverse mode of AD provides the gradient at a much cheaper cost.

However, there is a difficulty: the intermediate values X_{p-1} are used first, and then X_{p-2}, and so on until X_0. This is the inverse of their computation order in program `P`. A complete execution of `P` is necessary to get X_{p-1},and only then can the Jacobian×vector products be evaluated. But then X_{p-2} is required, whereas instruction $\texttt{I}_{p-1}$ may have overwritten it! There are mainly two ways to achieve this, called the *Recompute-All* (RA) and the *Store-All* (SA) approaches.

The RA approach recomputes each needed X_k on demand, by restarting the program on input X_0 until instruction $\texttt{I}_k$. This is the fundamental tactic of the AD tool TAMC/TAF [20]. The cost is extra execution time, grossly proportional to the square of the number of run-time instructions p. Figure 4.2 summarizes this tactic graphically. Left-to-right arrows represent execution of original instructions I_k, right-to-left arrows represent the execution of the reverse instructions $\overline{I}_k$ which implement $\overline{X}_{k-1} = f_k'^t(X_{k-1}).\overline{X}_k$. The big black dot represents the storage of all variables needed to restart execution from a given point, which is called a *snapshot*, and the big white dots represent restoration of these variables from the snapshot.

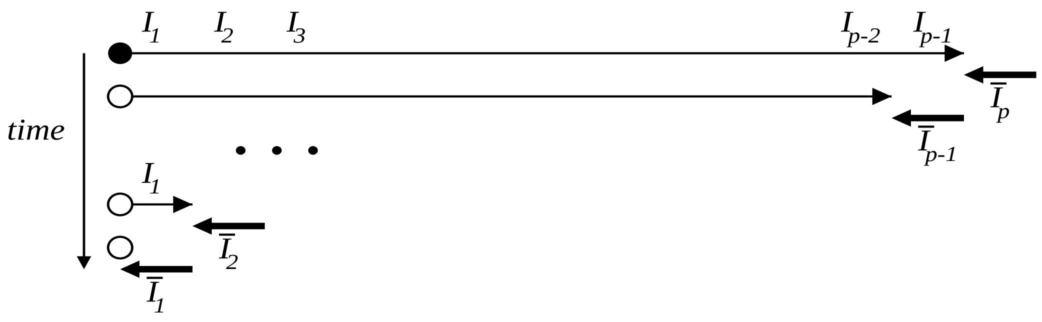


Figure 4.2: *The "Recompute-All" tactic*

The SA approach stores each X_k in memory, onto a stack, during a preliminary execution of program P, known as the *forward sweep*. Then follows the so-called *backward sweep*, which computes each $f_k'^t(X_{k-1})$ for $k = p$ down to 1, poping the X_{k-1} from this stack upon demand. This is the basic tactic in ADIFOR [6, 8] and TAPENADE. The cost is memory space, essentially proportional to the number of run-time instructions p. Figure 4.3 summarizes this tactic graphically. Small black dots represent storage of the X_k on the stack, before next instruction might overwrite them, and small white dots represent their popping from the stack when needed. We draw these dots smaller than on figure 4.2 because it turns out we don't need to store all X_k, but only the variables that will be overwritten by $\mathtt{I}_{k+1}$. Figure 4.4 illustrates AD in reverse mode, using the SA approach as done by TAPENADE. Actually, TAPENADE generates a simplified code, resulting from static analyses and improvements described later in this report. For clarity, figure 4.4 does not benefit from these improvements. The original program is the one of figure 4.1. Reverse derivatives are conventionally shown with a *bar* above. The *forward sweep* is on the left column, and the *backward sweep* on the right.

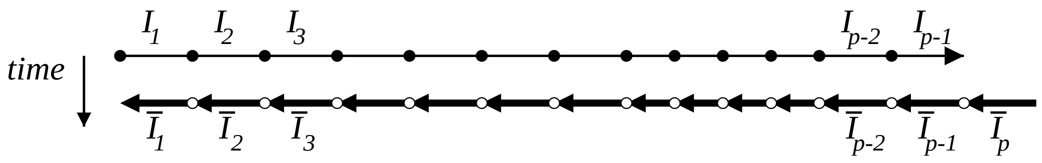


Figure 4.3: *The "Store-All" tactic*

Notice that in the backward sweep, the `do` loop now runs from `i=n` down to `1`. Considering again instruction `e = SQRT(e2)`, differentiation produces the following vector assignment that multiplies the instruction's *transposed* elementary Jacobian by the vector of reverse derivatives:

$$\begin{bmatrix} \overline{\texttt{e2}} \\ \overline{\texttt{e}} \end{bmatrix} = \begin{bmatrix} 1 & 0.5/\sqrt{\texttt{e2}} \\ 0 & 0 \end{bmatrix} . \begin{bmatrix} \overline{\texttt{e2}} \\ \overline{\texttt{e}} \end{bmatrix}$$

This takes two derivative instructions, because $\overline{\texttt{e}}$ must be reset to zero.

reverse mode: `T,U,`$\overline{\texttt{e}}$ $\mapsto$ $\overline{\texttt{T}}$,$\overline{\texttt{U}}$	
forward sweep:	*backward sweep:*
`e2 = 0.0`	$\overline{\texttt{e2}}$ `= 0.0`
`do i=1,n`	$\overline{\texttt{e1}}$ `= 0.0`
` PUSH(e1)`	$\overline{\texttt{e2}}$ `=` $\overline{\texttt{e2}}$ `+ 0.5*`$\overline{\texttt{e}}$`/SQRT(e2)`
` e1 = T(i)-U(i)`	$\overline{\texttt{e}}$ `= 0.0`
` PUSH(e2)`	`do i=n,1,-1`
` e2 = e2 + e1*e1`	` POP(e2)`
`end do`	$\overline{\texttt{e1}}$ `=` $\overline{\texttt{e1}}$ `+ 2*e1*`$\overline{\texttt{e2}}$
`e = SQRT(e2)`	` POP(e1)`
	$\overline{\texttt{T}}$`(i) =` $\overline{\texttt{T}}$`(i) +` $\overline{\texttt{e1}}$
	$\overline{\texttt{U}}$`(i) =` $\overline{\texttt{U}}$`(i) -` $\overline{\texttt{e1}}$
	$\overline{\texttt{e1}}$ `= 0.0`
	`end do`
	$\overline{\texttt{e2}}$ `= 0.0`

Figure 4.4: *AD in reverse mode, Store-All tactic*

Notice furthermore the calls to `PUSH` and `POP`, that store and retrieve the intermediate values of variables `e1` and `e2`: in practice, not *all* values need be stored before each instruction (*cf* section 4.4.2). Only the value(s) that are going to be overwritten need be stored.

The RA and SA approaches appear very different. However, on large programs `P`, neither the RA nor the SA approach can work. The SA approach uses too much memory (almost proportional to the **run-time** number of instructions). The RA approach consumes computation time (it will grossly square the run-time number of instructions). Both ways need to use a special trade-off technique, known as *checkpointing*. The idea is to select one or many pieces of the run-time sequence of instructions, possibly nested. For each piece `p`, one can spare some repeated recomputation in the RA case, some memory in the SA case, at the cost of remembering a *snapshot*, i.e. a part of the memory state at the beginning of `p`. On real programs, language constraints usually force the pieces to be subroutines, loops, loop bodies, or fragments of straight-line code.

There has been little work on the evaluation and comparison of these strategies. With the notable exception of Griewank's schedule of nested checkpoints (SA strategy) [23], which was proved optimal for `P` being a loop with a fixed number of iterations [25], there is no known optimal checkpointing strategy for arbitrary programs. Moreover, no theoretical comparison between the RA and SA approaches exist, nor a common framework in which these approaches could be combined. This is an open research problem, which could yield a huge benefit for AD tools, and help disseminate AD among possible users.

Let us now compare checkpointing on RA and SA in the ideal case of a pure straight-line program. We claim that checkpointing makes RA and SA come closer. Figure 4.5 shows how the RA approach can use checkpointing for one program piece `p` (the first part of the program), and then for two levels of nested pieces. On very large programs, 3 or more nested levels can be useful. At the second level, the memory space of the snapshot can be reused for different program pieces. The benefit comes from the checkpointed piece being executed fewer times. The cost is memory storage of the snapshot, needed to restart the program just after the checkpointed piece. The benefit is higher when `p` is at the beginning of the enclosing program piece. Similarly, figure 4.6 shows how the SA approach can use the same one-level and two-levels checkpointing schemes. Again, the snapshot space used for the second level of checkpointing is reused for two different program pieces. The benefit comes from the checkpointed piece being executed the first time without any storage of intermediate values. This divides the maximum size of the stack

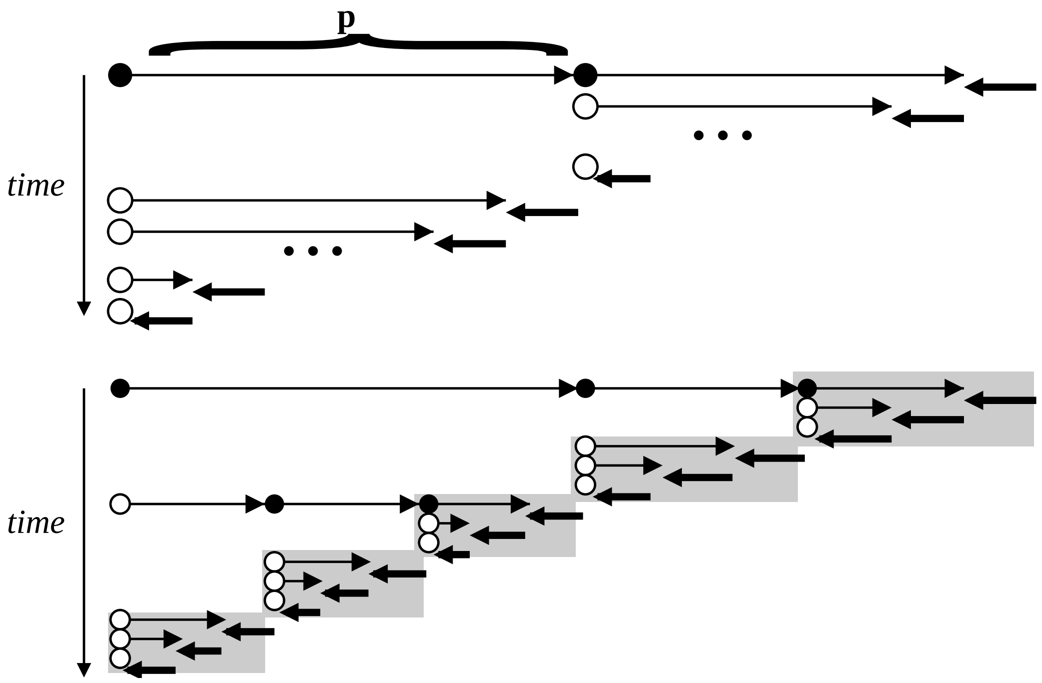


Figure 4.5: *Checkpointing on the "Recompute-All" tactic*

by 2. The cost is again the memory size of the snapshots, plus this time an extra execution of the program piece p. This makes the two schemes come closer as the number of nested ckeckpointing levels grow. On figure 4.5, the

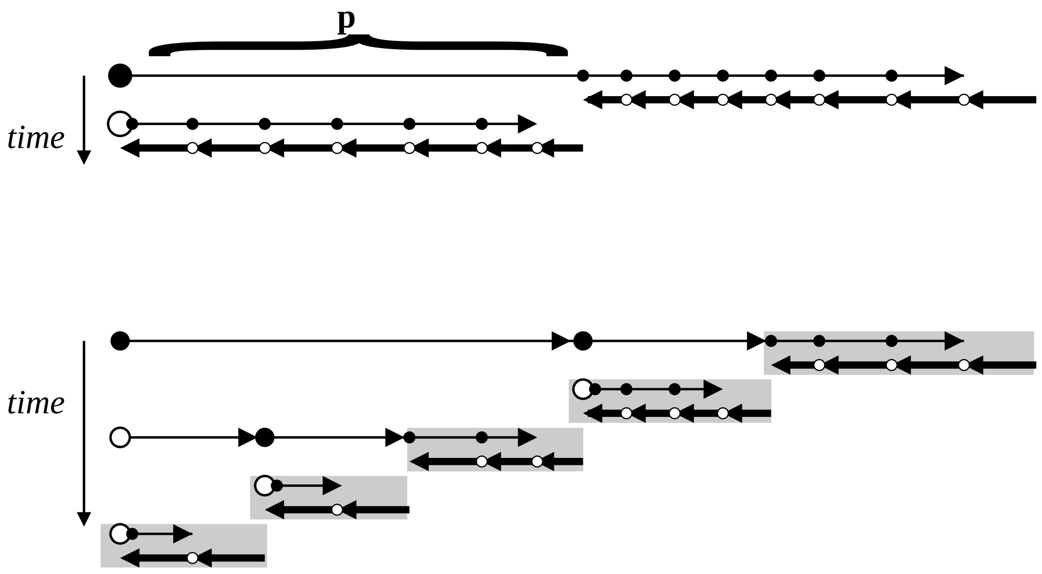


Figure 4.6: *Checkpointing on the "Store-All" tactic*

part on a grey background is a smaller scale reproduction of the basic RA scheme of figure 4.2. Similarly on figure 4.6, the grey box is a smaller scale reproduction of the basic SA scheme of figure 4.3. Apart from what happens at the "leaves" (the grey boxes), the figures 4.5 and 4.6 are identical. This shows that RA and SA with intense checkpointing can differ very little. The question remains to compare pure SA and pure RA, but it becomes less crucial as these are applied to smaller pieces of the program. However, we believe SA is more efficient, especially on small pieces of program, because the stack can stay in cache memory. We chose the SA approach for our tool TAPENADE. But SA could do better with some amount of recomputing!

4.2 Motivating Applications of Reverse AD

4.2.1 Inverse Problems and Data Assimilation

We call *inverse* all problems where unknown values cannot be measured directly, but instead we know some other measurable values which are *consequences of* – or *depend on* – the unknown. Given actual measures of the measurable values, the inverse problem is how to find the unknown values which are behind them.

To do this, we assume we have a physical model that we believe represents the way the unknown values determine the measurable values. When this model is complex, then the inverse problem is nontrivial. From this physical model we get a mathematical model, which is in general a set of partial differential equations. Let us also assume, but this is not absolutely necessary, that from the mathematical model, through discretization and resolution, we get a program that computes the measurable values from the unknown values.

Let us formalize the problem. This classical formalization comes from *optimal control theory.* We are studying the state W of a given system. In general, this state is defined for every point in space, and also if time is involved for every instant in an observation period $[0, T]$. Traditionally, the time coordinate is kept apart from the others (i.e. space). The mathematical model relates the state W to a number of external parameters, which are the collection of initial conditions, boundary conditions, model parameters, etc, i.e. all the values that determine the state. The unknown values γ are just some of these external parameters. In general this relation is implicit. It is a set of partial differential equations that we write:

$$\Psi(\gamma, W) = 0 \tag{4.6}$$

Equation (4.6) takes into account all external parameters, but we are only concerned here by the dependence on γ. In optimal control theory, we would call γ our control variable.

Any value of γ thus determines a state $W(\gamma)$. We can easily extract from this state the measurable values, and of course there is very little chance that these values exactly match the values actually measured W_{obs}. Therefore we start an optimization cycle to modify the unknown values γ, until the resulting measurable values match best. We thus define a *cost function* that measures the discrepancy on the measurable values in $W(\gamma)$. In practice, not all values in $W(\gamma)$ can be measured in W_{obs}, but nevertheless we can define this cost function J as the sum at each instant of some squared norm of the discrepancy of each measured value $\|W(\gamma) - W_{obs}\|^2$.

$$j(\gamma) = J(W(\gamma)) = \frac{1}{2}\int_{t=0}^{T} \|W(\gamma)(t) - W_{obs}(t)\|^2 dt \tag{4.7}$$

Therefore the inverse problem is to find the value of γ that minimizes $j(\gamma)$, which is such that $j'(\gamma) = 0$. If we use a gradient descent algorithm to find γ, then we need at least to find the value of $j'(\gamma)$ for each γ.

Here are two illustration examples:

- If the system we study is a piece of earth crust, it is difficult to measure directly the locations of the different layers of rock. However, these locations condition shock wave propagation, and eventually condition the measurable delay after which an initial shock is received at a distant place. So γ is the location of layers of rock, Ψ models wave propagation inside the rock, W is the position of the waves at each instant, from which we deduce the theoretical reception delays at points of measurement. J is the discrepancy we must minimize to find the best estimation of rock layers location. The same method applies to find an unknown drag coefficient of the bottom of a river, given the model Ψ that captures the shape of the river bed, and measured values of the river's surface shape and input flow.

- In meteorology, the system studied is the evolution of the atmosphere. The data assimilation problem looks for the best estimation of the initial state from which the simulation will start. This initial state W_0 at $t = 0$ is largely unknown. This is our γ. All we have is measurements at various places for various times in $[0, T]$. We also know that this initial state and all later states in $[0, T]$ must obey the atmospheric equations $\Psi(\gamma, W) = 0$. The inverse problem that looks for the initial state $\gamma = W_0$ that generates the sequence of states at each time in $[0, T]$ which is closest to the observed values, using a gradient descent, is called *variational data assimilation.*

To find $j'(\gamma)$, the mathematical approach first applies the chain rule to equation (4.7), yielding:

$$j'(\gamma) = \frac{\partial J(W(\gamma))}{\partial \gamma} = \frac{\partial J}{\partial W}\frac{\partial W}{\partial \gamma} \tag{4.8}$$

The derivative of W with respect to γ come from the state implicit equation (4.6), which we differentiate with respect to γ to get:

$$\frac{\partial \Psi}{\partial \gamma} + \frac{\partial \Psi}{\partial W}\frac{\partial W}{\partial \gamma} = 0 \tag{4.9}$$

Assuming this can be solved for $\frac{\partial W}{\partial \gamma}$, we can then replace it into equation (4.8) to get:

$$j'(\gamma) = -\frac{\partial J}{\partial W}\frac{\partial \Psi}{\partial W}^{-1}\frac{\partial \Psi}{\partial \gamma} \tag{4.10}$$

Now is the time to consider complexity of resolution. Equation (4.10) involves one system resolution and then one product. First notice that

$$\frac{\partial \Psi}{\partial W}$$

is definitely too large to be computed explicitly, and therefore its inverse cannot be computed and stored either. Nowadays both Ψ and W are discretized with millions of dimensions. So our choices are either to run an iterative resolution for

$$\frac{\partial \Psi}{\partial W}^{-1}\frac{\partial \Psi}{\partial \gamma} \tag{4.11}$$

and then multiply the result by $\frac{\partial J}{\partial W}$, or else to run an iterative resolution for

$$\frac{\partial J}{\partial W}\frac{\partial \Psi}{\partial W}^{-1} \tag{4.12}$$

and then multiply the result by $\frac{\partial \Psi}{\partial \gamma}$. We notice that $\frac{\partial \Psi}{\partial \gamma}$ has many columns, following the dimension of γ, which can be several thousands. Therefore computation of (4.11) requires as many resolutions of the linear system

$$\frac{\partial \Psi}{\partial W}x = \frac{\partial \Psi}{\partial \gamma}$$

Conversely, j is a scalar, $\frac{\partial J}{\partial W}$ is a row vector, and it takes only one resolution to compute (4.11), solving

$$\Pi^*\frac{\partial \Psi}{\partial W} = \frac{\partial J}{\partial W}$$

for Π, which is called the *adjoint state*. This second approach is more efficient. To summarize, this preferred *adjoint method* first solves

$$\frac{\partial \Psi}{\partial W}^* .\Pi = \frac{\partial J}{\partial W}^*$$

for the adjoint state Π, then just computes

$$j'(\gamma) = -\Pi^* \frac{\partial \Psi}{\partial \gamma}$$

Suppose now that we already have a resolution program, i.e. a procedure $\mathtt{P}_\Psi$ which, given γ, returns W_γ. We also have a procedure $\mathtt{P}_j$ which, given a W, evaluates the cost function, i.e. the discrepancy between W and the observed W_{obs}. Then we can avoid all the programming step involved by the above mathematical method, using AD. Automatic Differentiation in reverse mode of the program that computes

$$j = \mathtt{P}_j(\mathtt{P}_\Psi(\gamma))$$

directly gives the gradient of j, i.e. the desired $j'(\gamma)$. This is indeed very close to the mathematical resolution with the adjoint state: the reverse mode actually computes a discretized adjoint on the program. One difference is that the adjoint mechanism is applied to the whole program $\mathtt{P}_j \circ \mathtt{P}_\Psi$, including the resolution algorithm, whereas the resolution algorithm for Π above may be different from the resolution algorithm for W.

So to get $j'(\gamma)$, we can either write the adjoint equations, then discretize them and solve them, or else we can use the reverse mode of AD on the program that computes j from γ. The first method is more difficult, because it involves a new implementation. The second method ideally doesn't require additional programming.

There is a difficulty though. The resolution for $j'(\gamma)$ by AD uses reverse differentiation of $\mathtt{P}_\Psi$. $\mathtt{P}_\Psi$ takes γ and returns W, computed iteratively. $\overline{\mathtt{P}_\Psi}$ takes $\overline{W}$ and returns $\overline{\gamma}$, computed iteratively with the same number of iterations. The question is "will this second iteration converge?". Jean-Charles Gilbert has shown [21] that under some complex but widely satisfied constraints, if an iterative resolution converges to a result, then its AD derivative converges to the derivative of the result. So we are confident that $\overline{\mathtt{P}_\Psi}$ eventually converges to the desired $\overline{\gamma}$. But we are not sure it will converge at same speed! In other words, it is not sure that an efficient algorithm to compute $W(\gamma)$ and then $j(\gamma)$ yields an efficient resolution algorithm to compute $j'(\gamma)$.

The next section illustrates an hybrid approach that uses the AD reverse derivatives for the non-iterative part of the resolution, and then solves the adjoint system by hand with an ad-hoc resolution algorithm. This may be an answer to the difficulty above.

4.2.2 Optimization in Aerodynamics

This application uses AD to optimize the shape of a supersonic aircraft, in order to minimize the sonic boom felt on the ground. The problem, shown on figure 4.7, is modelized numerically as the following chain of operations,

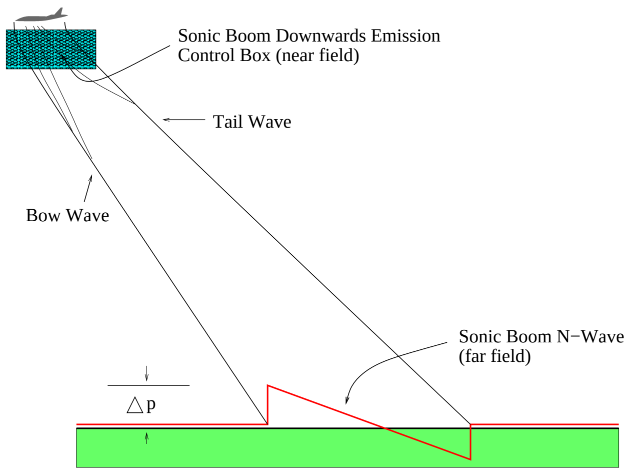


Figure 4.7: *The sonic boom: shock wave patterns on the near and far fields.*

that go from the parameters γ that define the geometry of the plane to a measure of the sonic boom on the ground, the cost function $j(\gamma)$.

$$\begin{array}{ccccccccc} \text{Control} & & & & \text{Euler} & & \text{Pressure} & & \text{Cost} \\ \text{points} & \Rightarrow & \text{Geometry} & \Rightarrow & \text{flow} & \Rightarrow & \text{shock} & \Rightarrow & \text{function} \\ \gamma & & \Omega(\gamma) & & W(\gamma) & & \nabla p(W) & & j(\gamma) \end{array}$$

The intermediate steps are: the complete geometry $\Omega(\gamma)$ of the plane, the Euler flow $W(\gamma)$ around this geometry, and the pressure gradients $\nabla p(W)$ under the plane. The cost function $j(\gamma)$ actually combines the integral of the squared pressure gradients with the discrepancy from prescribed lift and drag coefficients.

We want to minimize $j(\gamma)$, using a gradient method, by iterative modifications of the shape parameters γ. This gradient descent is driven by $j'(\gamma)$, which we must compute.

Since the chain from γ to $j(\gamma)$ is actually a program, we can apply AD in the reverse mode to the complete program to get $j'(\gamma)$. However this is impractical due to the size of the program. Remember that the reverse mode, with the Store-All strategy, has a memory cost proportional to the execution time of the program. There are also questions about convergence: the original program solves the flow $W(\gamma)$ iteratively. It is not sure whether it makes sense to differentiate this complete iterative process.

We proposed in [35] an hybrid approach, where we only apply AD to selected parts of the original program, then use the differentiated pieces in a new, hand-coded solver, yielding the adjoint of the discretized flow equations and finally the gradient. To this end, we go back to the mathematical equations. Basically, we consider a minimization problem under a particular additional constraint: the flow equation $\Psi(\gamma, W(\gamma)) = 0$, which expresses the dependence of the flow field $W(\gamma)$ on the shape γ. We thus want to find the γ_0 that minimizes the cost functional $j(\gamma) = J(\gamma, W(\gamma))$, under the constraint $\Psi(\gamma, W(\gamma)) = 0$. Here, this constraint is the compressible Euler equations, solved in a domain Ω_γ parametrized by γ. This minimization problem is solved using Lagrange multipliers. The problem's Lagrangian is

$$L(W, \gamma, \Pi) \; = \; J(\gamma, W) + \langle \Psi(\gamma, W), \Pi \rangle, \tag{4.13}$$

where Π is the *adjoint state*, which is a generalized Lagrange multiplier, and $\langle\ ,\ \rangle$ is a suitable scalar product. Then the gradient $j'(\gamma)$ is found by solving

$$\begin{aligned}
&\Psi(\gamma, W(\gamma)) \;=\; 0 \\
&\nabla_W J(\gamma, W(\gamma)) \;-\; (\nabla_W \Psi(\gamma, W(\gamma)))^* \, \Pi(\gamma) \;=\; 0 \\
&j'(\gamma) \;=\; \nabla_\gamma J(\gamma, W(\gamma)) \;-\; (\nabla_\gamma \Psi(\gamma, W(\gamma)))^* \, \Pi(\gamma).
\end{aligned} \tag{4.14}$$

The first line gives $W(\gamma)$. The second line is the so-called *adjoint flow equation*, and gives the adjoint state $\Pi(\gamma)$. The last line gives the *gradient* $j'(\gamma)$ using $\Pi(\gamma)$ and $W(\gamma)$. This gradient will be used to update iteratively the former γ.

We then remark that, if we isolate the subprogram `Psi` of the flow solver program `P` that computes $\Psi(\gamma, W(\gamma))$, the reverse mode of AD can build automatically a new subprogram $\overline{\texttt{Psi}}_W$. This new subprogram, given any vector Π, returns the product $(\nabla_W \Psi(\gamma, W(\gamma)))^* \, \Pi$. Differentiation with respect to γ instead of W gives another subprogram $\overline{\texttt{Psi}}_\gamma$ that for any Π returns $(\nabla_\gamma \Psi(\gamma, W(\gamma)))^* \ \Pi$. Similarly, if we isolate the subprogram `J` that computes the cost function $J(\gamma, W(\gamma))$, the reverse mode of AD automatically builds subprograms $\overline{\texttt{J}}_W$ and $\overline{\texttt{J}}_\gamma$ that respectively compute $\nabla_W J(\gamma, W(\gamma))$ and $\nabla_\gamma J(\gamma, W(\gamma))$.

With these subroutines generated, what remains to be done by hand to get $j'(\gamma)$ is the solver that solves the *adjoint flow equation* for Π. Notice that AD does not give us the matrix $(\nabla_W \Psi(\gamma, W(\gamma)))^*$ explicitly. Anyway this (2^{nd}-order) Jacobian matrix, although sparse, is too large for efficient storage and use. Therefore, we build a *matrix-free* linear solver. Fortunately, we just need to modify the algorithm developed for the flow solver `P`. `P` uses a simplified (1^{st}-order) Jacobian for preconditioning the pseudo-Newton time advancing. This matrix is stored. We just transpose this simplified Jacobian and reuse its Gauss-Seidel solver to build a preconditioned fixed-point iteration, that solves the adjoint flow equation. This method is discussed in detail for a 2D application in [12]. We validated the resulting gradient by direct comparison with divided differences of the cost function. The relative error is about 10^{-6}.

The overall optimization process is made of two nested loops. The outer loop evaluates the gradient, using an adjoint state as described above, then calls the inner loop which does a 1D search to get the steepest descent parameter, and finally updates the control parameters

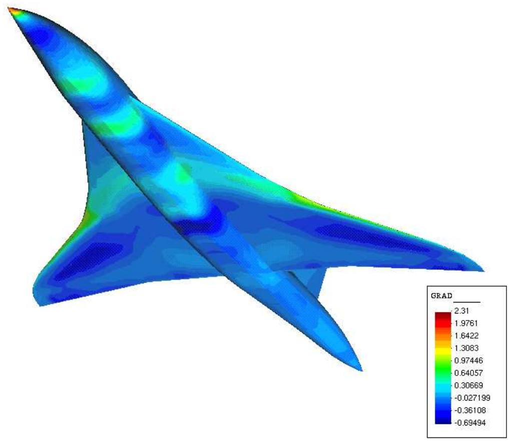
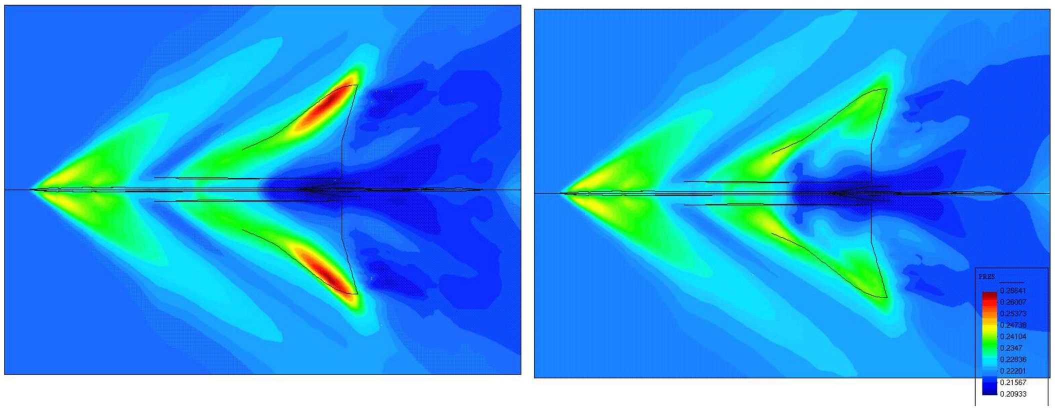


Figure 4.8: *Supersonic Business Jet: Gradient of the Cost Functional on the skin*

We applied this optimization program on the shape of a Supersonic Business Jet, currently under development at Dassault Aviation. The mesh consists of 173526 nodes and 981822 tetrahedra (for half of the aircraft). The inflow Mach number is 1.8 and the angle of attack is 3°. We target optimization of the wings of the aircraft only. Even then, the flow, the adjoint state, and the gradient $j'(\gamma)$ are computed taking into account the complete aircraft geometry. Figure 4.8 shows the gradient of our "sonic boom" cost functional j on the skin on the complete aircraft. Darker colors indicate places where modifying the shape strongly improves the sonic boom.



original geometry *optimized geometry*

Figure 4.9: *Supersonic Business Jet: Pressure distribution in a plane below the aircraft*

Figure 4.9 shows the evolution of the pressure on the near field, after 8 optimization cycles. We observe that the shock produced by the outboard part of the wings is dampened. However, within the Mach cone, close to the fuselage, the pressure peak has slightly increased after the optimization process. This increase is tolerable, compared to the reduction obtained on the end of the wings. Paper [49] describes further how this shape optimization improves the actual sonic boom on the ground.

4.3 Using the Flow Graph for flow inversion in reverse AD

In section 4.1.3 we saw that the backward sweep of the differentiated program must sweep through instructions in the reverse order. For a real program, with control, the control flow of the backward sweep must be the inverse of the control flow of the original program, or equivalently of the forward sweep. In this section, we advocate using the Flow Graph to achieve this.

When program `P` is well structured, as recommended by good programming practice, the Flow Graph is embedded in the syntax tree, and flow inversion can be done on the syntax tree. But things are not so simple on real programs. Instructions such as `cycle`, `exit`, `exception` raising, let alone the dreaded `goto`, introduce control which is not embedded in the syntax tree. Only the Flow Graph can represent explicitly this sort of control flow.

Figure 4.10 shows on top the Flow Graph of a piece of the forward sweep, and below the Flow Graph of the corresponding piece of the backward sweep. This shows the simple flow reversal strategy of ODYSSÉE 1.6, a predecessor of TAPENADE. Conditionals were reversed by duplicating the test from the forward sweep, and only `DO` loops could be reversed, with a loop index going in the reverse order. As one can see, ODYSSÉE could only handle well structured programs. On figure 4.10, the second arrow that reaches block `B5` comes from a `goto`, and it is impossible to generate the expected behavior at the end of block $\overline{\texttt{B5}}$ on the backward sweep. Even worse, test `t1` is now done between block $\overline{\texttt{B4}}$ and blocks $\overline{\texttt{B2}}$ and $\overline{\texttt{B3}}$. This is not compatible with the order of the forward sweep, where `t1` is executed between block `B1` and blocks `B2` and `B3`. If `t1` uses a variable `v` which is overwritten and then used in `B2`, then in the backward sweep, `t1` needs the former value `v` and $\overline{\texttt{B2}}$ needs the latter value, which is not compatible with the stack first-in/first-out mechanism.

Figure 4.11 shows an attempt at reversing arbitrary Flow Graphs. This is the strategy of ODYSSÉE 1.7. Basically, the forward sweep remembers the list of all Basic Blocks executed, and then the backward sweep executes the corresponding backward sweeps of each Basic Block. This is correct but extremely awkward, because all the control structure is flattened, and the compiler gives a poorly optimized code. This was a first attempt at using the Flow Graph.

We think that Flow Graph reversal must return a structured Flow Graph,

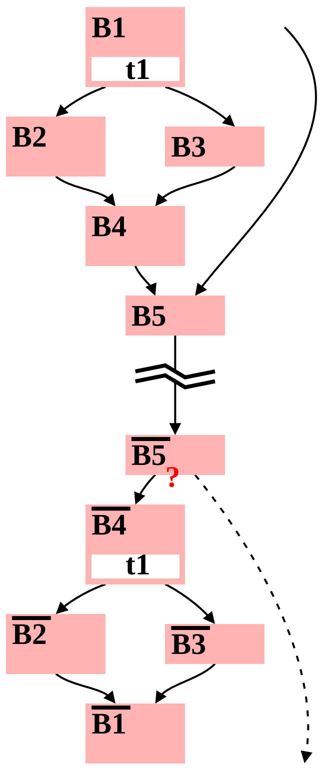


Figure 4.10: *Naive control reversal on the syntax tree (*ODYSSÉE 1.6*)*

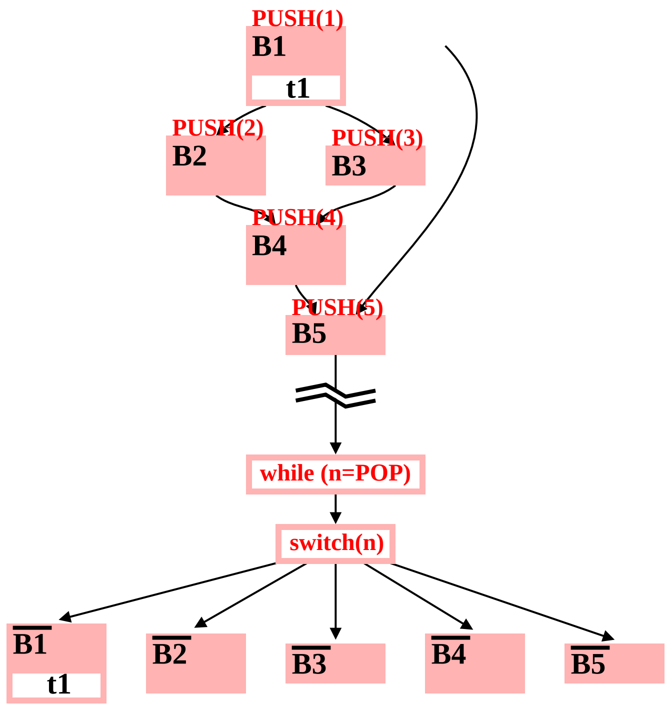


Figure 4.11: *General Flow Graph flattening and inversion (*ODYSSÉE 1.7*)*

with as many structured instructions as the forward sweep. The places where the flow of control splits in the backward sweep are the places where the flow of control merges in the forward sweep. This tells us where to add the instructions that store the flow decisions in the forward sweep: just before flow merges. This preserves the stack order of values stored for reverse AD. This is shown on figure 4.12. The structured test remains a structured test,

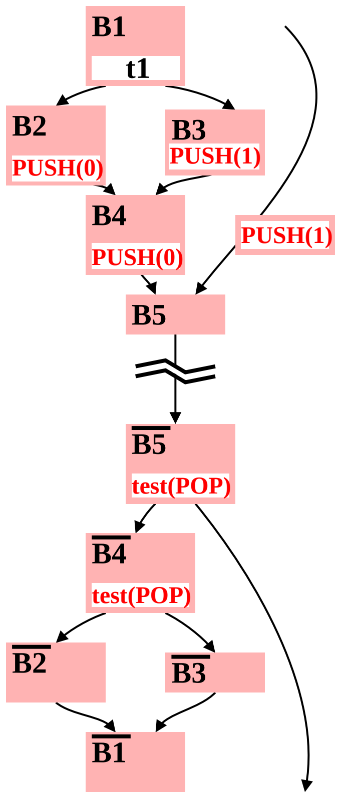


Figure 4.12: *Control reversal on the Flow Graph (*TAPENADE*)*

whereas the test at the end of $\overline{\texttt{B5}}$ may very well result in a `goto`. Figure 4.13 shows the strategy to reverse a loop: in the backward sweep is also a loop, and only the necessary control values are pushed on the stack.

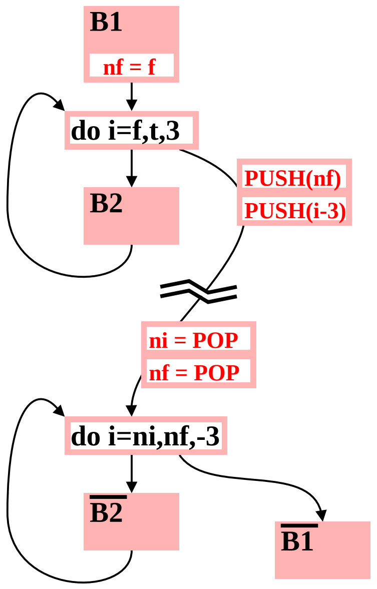


Figure 4.13: *control reversal of a* `DO` *loop (*TAPENADE*)*

4.4 Static Data Flow Analyses for a better AD

There is an already very large collection of Data Flow analyses developped for compilers (standard, optimizing, or parallelizing). However, AD exhibits specific behaviors that require new data flow analyses. In this section, we describe these new analyses and show their interest. The emphasis will be put on how these analyses can be formalized cleanly and implemented efficiently on programs internally kept as Flow Graphs, using so-called "*data flow equations*". In section 4.4.1, we discuss "*activity*" analysis, which is central to all modes of AD. In section 4.4.2, we give special attention to an analysis for the reverse mode, called "*TBR*". In section 4.4.3, we define an extension of the classical Read-Written analysis to optimize the size of "snapshots". In section 4.4.4, we present extensions to dead code detection, that can also improve AD-generated code.

4.4.1 Activity Analysis

In theory (*cf* section 4.1.1), AD must compute for each instruction the partial derivative of the instruction's output with respect to its inputs. For example in tangent mode, for any instruction such as `x(i)=a*b(i+j)`, the derivative instruction will a priori involve the derivatives $\dot{\texttt{a}}$, $\dot{\texttt{b}}$, $\dot{\texttt{x}}$ of `a`, `b`, `x`.

In practice, not all derivatives need be computed, because some of them can be statically proven always null, or never used. In AD tools, the end-user who requests differentiation of program `P` can specify which of the results of `P` must be differentiated, and with respect to which of the inputs of `P`. These variables are called respectively the *dependent* outputs and the *independent* inputs. Informally, an occurrence inside `P` of a variable that does not "depend" on any *independent* has certainly a zero derivative. Conversely, an occurrence of a variable on which no *dependent* "depends" has certainly a derivative which is useless to get the desired results. More formally, we shall say that the variable v_i assigned by an instruction "depends in a differentiable way" on the variable v_j used in this instruction, and we write $v_j \prec v_i$, iff the partial derivative of v_i with respect to v_j is defined. In other words, this partial derivative cannot be statically proven always null. Considering for example instruction `x(i)=a*b(i+j)`, the left-hand side `x(i)` depends on `a` and `b(i+j)`, but not on `i` nor `j`. We define $\prec^*$ as the closure of relation $\prec$:

$$\prec^* = \cup_{k=1}^{\infty} \prec^k$$

where $\prec^k$ is the composition of k times $\prec$. Given the set $\mathbf{x_I}$ of independent input variables, and the set $\mathbf{y_D}$ of dependent output variables, both sets provided by the end-user, the variables for which the derivative must be computed (*"active"* variables) are the v that are both *"varied"*, i.e.

$$\exists x \in \mathbf{x_I} : x \prec^* v$$

and *"useful"*, i.e.

$$\exists y \in \mathbf{y_D} : v \prec^* y$$

All other variables are called *"passive"*. For them, there is no derivative variable: no value is computed for this derivative variable, and if its value is used somewhere, it is replaced by zero and simplified. Therefore activity analysis does two things:

- Forwards from the beginning of the program, it must propagate the set of all variables that possibly depend on some independent input.

- Backwards from the end of the program, it must propagate the set of all variables on which some dependent output possibly depends.

Those are two static interprocedural data flow analyses. To master combinatorial explosion, we synthesize (i.e., *bottom-up*) the most expensive part of the analysis, which is the relation $\prec^*$ for each subroutine, during a preliminary phase called the *Differentiable Dependency Analysis*.

Differentiable Dependency Analysis

For each subroutine bottom-up on the Call Graph, we compute all pairs of variables $(v_b.v_a)$, v_b just *before* the subroutine and v_a just *after* the subroutine, such that $v_b \prec^* v_a$.. We call this set *Dep*. It can be implemented very efficiently as a matrix of Booleans, but we shall stick to sets for this formal description. We are going to give the data flow equations that compute *Dep* at each level of the program representation.

This dependency information for subroutines requires that we compute similar information at lower levels in the program representation, i.e. for each control structure, Basic Block, and for each instruction. If the program is recursive, there are cycles in the Call Graph, and therefore this computation needs a fix point at the Call Graph level.

For an individual instruction, there are two main cases: assignments and subroutine calls. We will examine subroutine calls when we deal with the interprocedural aspect. Let us see assignments first. After an assignment, the assigned variable depends on all variables that occur in differentiable position in the right-hand side. This set (call it *DP*) is given by the following constructive definition:

exp:	$e_1\ op\ e_2$	$\varphi(e_1)$	$e_1[i]$	v	c
$DP(exp)$:	$DP(e_1) \cup DP(e_2)$	$DP(e_1)$	$DP(e_1)$	$\{v\}$	$\emptyset$

Above, $op \in \{+, -, *, \ldots\}$, $\varphi \in \{\sin, \exp, \tan, \ldots\}$, $e_1[i]$ is an array reference, v is a single variable, and c a constant. All variables other than the assigned variable remain unchanged. They depend just on themselves. Thus for an assignment I : "v=*exp*"

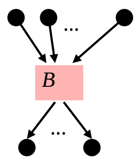
$$Dep(I) = \{(d.\mathtt{v}), \forall d \in DP(exp)\} \cup \{(x.x), \forall x \neq \mathtt{v}\}.$$

When dealing with arrays, we must overestimate *Dep* as follows: if the left-hand side `v` is an array reference, then some parts of the array may retain their old values, and therefore we must add dependence (`v.v`) to $Dep(I)$.

For a basic block, and more generally for any sequence of structured program pieces $\mathtt{p}_i$, we define the sequential composition $\otimes$ of the *Dep* sets as follows:

$$\begin{aligned} Dep(\mathtt{p}_1;\mathtt{p}_2) &= Dep(\mathtt{p}_1) \otimes Dep(\mathtt{p}_2) \\ &= \{(v_b.v_a) : \exists v_x : (v_b.v_x) \in Dep(\mathtt{p}_1) \wedge (v_x.v_a) \in Dep(\mathtt{p}_2)\} \end{aligned}$$

For a subroutine S, $Dep(S)$ is built from the *Dep* of each basic block in its flow graph, using data flow equations. For each basic block B, we introduce $InDep(B)$ (resp. $OutDep(B)$), the dependencies from the entry block of S to the *beginning* (resp. *end*) of B. The data flow equations given in figure 4.14 relate the *InDep* and the *OutDep* of adjacent basic blocks. These equations form a system that can be solved iteratively. We can prove



$$InDep(B) = \bigcup_{P\ predecessor\ of\ B} OutDep(P)$$

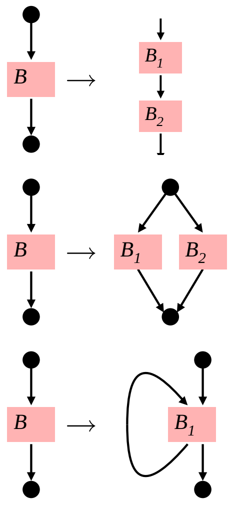
$$OutDep(B) = InDep(B) \otimes Dep(B)$$

Figure 4.14: *General data flow equations for differentiable dependency analysis*

that these iterations eventually terminate: the *InDep* and *OutDep* of each basic block are initialized to $\emptyset$, except for the *InDep* of the entry block, which is initialized to the identity "*Id*" (every variable depends on itself only). During iteration, the successive values of each basic block's *InDep* and *OutDep* are growing, inside a finite domain with a maximum element (every variable depends on every variable). Therefore iteration terminates, and after resolution the *OutDep* of the exit block of S is exactly the desired $Dep(S)$.

In the case of *structured* flow graphs, the data flow equations can be specialized into structured data flow equations, shown on figure 4.15. They compute the *Dep* sets explicitly, recursively bottom up on the structured flow graph, rather than iteratively on the unstructured flow graph. Structured data flow equations are less general, but more efficient. In particular, there is no more iterative solving, except inside the equation for loops, to compute the closure of $Dep(B_1)$, seen as a relation.

Subroutine calls are handled at the call graph level. For an individual instruction I which is "`call` S`(...)`", $Dep(I)$ is basically $Dep(S)$ (with a



$$Dep(B) = Dep(B_1) \otimes Dep(B_2)$$

$$Dep(B) = Dep(B_1) \cup Dep(B_2)$$

$$Dep(B) = (Dep(B_1))^*$$

Figure 4.15: *Structured data flow equations for the differentiable dependency analysis*

technical step of translating the variable names, as the name space of S differs from that of the calling subroutine). Therefore, the *Dep* set of each subroutine can be computed as soon as the *Dep* sets of all subroutines possibly called inside it have been computed. Consequently, when the call graph is acyclic, the *Dep* sets of each subroutine are computed by a bottom-up sweep. Otherwise they must be computed iteratively. We shall not describe here this iterative computation, which poses no fundamental problem anyway.

Varied and Useful variables

After the *Dep* sets are synthesized, activity analysis goes on propagating two data flow sets through the program:

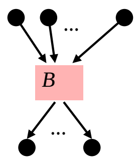
- The *varied* variables are variables v such that $\exists x \in \mathbf{x_I}, x \prec^* v$. $\mathit{InVary}(\mathtt{p})$ (resp. $\mathit{OutVary}(\mathtt{p})$) denotes the set of varied variables just *before* (resp. *after*) a given program piece `p`. For the whole program `P`, by definition, $\mathit{InVary}(\mathtt{P}) = \mathbf{x_I}$, which is then propagated forwards on the program flow.

- The *useful* variables are variables v such that $\exists y \in \mathbf{y_D}, v \prec^* y$. *InUseful*(`p`) (resp. *OutUseful*(`p`)) denotes the set of useful variables just *before* (resp. *after*) a given program piece `p`. For the whole program `P`, by definition, *OutUseful*(`P`) $= \mathbf{y_D}$, which is then propagated backwards on the program flow.

A variable is *active* when it is *varied* and *useful*. Both analyses run top down on the call graph. Solutions must be obtained iteratively if the call graph contains cycles. For an acyclic call graph, subroutines are analyzed in an order obtained by topological sorting. This approach ensures that all calls to subroutine S are analyzed before looking at S itself.

For a subroutine S, both analyses run similarly on the flow graph. The data flow equations are shown on figure 4.16 for unstructured flow graphs and on figure 4.17 for some sample structured flow graphs. The *InVary* set of the entry block of S is initialized to $InVary(S)$, which is the union of the varied variables before S, on all calling contexts. Similarly, the *OutUseful* set of the exit block of S is initialized to $OutUseful(S)$, which is the union of the useful variables after S, on all calling contexts. For these equations, we extended the operator $\otimes$ on sets V of variables as follows:

$$V \otimes Dep(B) = \{v_a : \exists v_b \in V : (v_b.v_a) \in Dep(B)\}$$
$$Dep(B) \otimes V = \{v_b : \exists v_a \in V : (v_b.v_a) \in Dep(B)\}$$



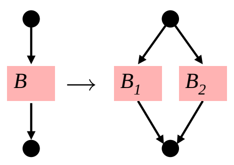
$$InVary(B) = \bigcup_{P \text{ predecessor of } B} OutVary(P)$$
$$OutVary(B) = InVary(B) \otimes Dep(B)$$
$$OutUseful(B) = \bigcup_{S \text{ successor of } B} InUseful(S)$$
$$InUseful(B) = Dep(B) \otimes OutUseful(B)$$

Figure 4.16: *General data flow equations for activity analysis*

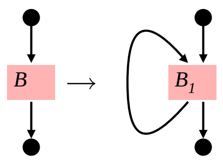
Finally, at the instruction level, the following rules propagate the varied variables forwards and the useful variables backwards, across any instruction I:

$$OutVary(I) = InVary(I) \otimes Dep(I)$$
$$InUseful(I) = Dep(I) \otimes OutUseful(I)$$

In addition to the above rules, for a subroutine call I : "`call` S`(...)`", the following rules accumulate the activity information for the present calling context into



$$\begin{aligned}
InVary(B_1) &= InVary(B_2) = InVary(B)\\
OutVary(B) &= OutVary(B_1) \cup OutVary(B_2)\\
OutUseful(B_1) &= OutUseful(B_2) = OutUseful(B)\\
InUseful(B) &= InUseful(B_1) \cup InUseful(B_2)
\end{aligned}$$



$$\begin{aligned}
InVary(B_1) &= InVary(B) \otimes (Id \cup Dep(B))\\
OutVary(B) &= OutVary(B_1)\\
OutUseful(B_1) &= (Id \cup Dep(B)) \otimes OutUseful(B)\\
InUseful(B) &= InUseful(B_1)
\end{aligned}$$

Figure 4.17: *Examples of structured data flow equations for activity analysis*

$InVary(S)$ and $OutUseful(S)$, their respective unions for all calling contexts. These sets will be used upon initialization when analyzing S itself.

$$\begin{aligned}
InVary(S) &= InVary(S) \cup InVary(I)\\
OutUseful(S) &= OutUseful(S) \cup OutUseful(I)
\end{aligned}$$

4.4.2 TBR Analysis

In section 4.1.3, we saw that the main drawback of the reverse mode of AD is the memory consumption to store intermediate values before they are overwritten during the forward sweep. Although checkpointing is a general answer to this problem, it is advisable to restrict this storage to intermediate values that are *really* needed by the backward sweep. Consider for example an assignment `x = a+2*b`. The partial derivatives of the sum do not use the values of `a` nor `b`. Therefore, as far as this instruction is concerned, there is no need to store the values of `a` nor `b` in case they get overwritten in the forward sweep. This is the purpose of the TBR analysis [19, 39, 33], which analyses the program to find which variables are *To Be Recorded*, and which are *not*.

The left column of figure 4.18 shows an example original code, which is the body of a subroutine that uses values from an array `x` to compute new values of `x`. The right column of figure 4.18 shows the corresponding backward sweep, which implements equation (4.5). The middle column displays the forward sweep. Now consider variable a. Each instruction that uses a has a

Original Code	**Forward Sweep**	**Backward Sweep**
i=0;j=10;a=3.14159	i=0;j=10;a=3.14159	
	COUNT=0	POP(COUNT)
while (check(j)) {	while (check(j)) {	while (COUNT>0) {
if (max(i,j)>7) {	if (max(i,j)>7) {	COUNT=COUNT-1
	PUSH(x(i))	POP(j)
x(i)=j+sin(x(i))	x(i)=j+sin(x(i))	POP(i)
	PUSH(true)	POP(test)
} else {	} else {	if (test) {
	PUSH(x(j))	POP(x(i))
x(j)=j*cos(x(j))+a	x(j)=j*cos(x(j))+a	$\overline{\mathtt{x}}$(i)=
	PUSH(false)	cos(x(i))*$\overline{\mathtt{x}}$(i)
}	}	} else {
	PUSH(i)	POP(x(j))
i=i+1	i=i+1	$\overline{\mathtt{x}}$(j)=
	PUSH(j)	-j*sin(x(j))*$\overline{\mathtt{x}}$(j)
j=j-1; a=a/2	j=j-1; a=a/2	}
	COUNT=COUNT+1	
}	}	}
	PUSH(COUNT)	

Figure 4.18: *Original code, forward sweep, backward sweep*

local Jacobian that does not require a. Therefore, it is not necessary to store a before it is overwritten by instruction a = a/2, and TBR analysis decided not to insert the corresponding PUSH/POP instructions. On the other hand, x, i, and j are recorded beacuse they are used in the backward sweep.

We are going to formalize TBR analysis on the Flow Graph, like we did before for activity analysis. Thus, TBR analysis follows the flow of the original code, propagating the set of variables whose current value is required in the backward sweep, and flags assignments that overwrite such a variable, so that its value will be recorded. As before, we identify a bottom-up analysis in order to control combinatorial explosion.

Bottom-Up TBR Analysis

For each structured piece `p` of the program, we synthesize a summary of the effect of `p` on TBR propagation. Concretely, this effect is composed of two parts:

- The *killed* variables, those variables present at the beginning of `p` that are certainly completely overwritten inside `p`. We denote this set *Kill*(`p`).
- The *adjoint-used* variables, those variables present at the end of `p` which will be used in the adjoint of `p`. We denote this set *AdjU*(`p`).

We are going to give the data flow equations that define these two sets at each level of the program representation.

For an individual instruction I, let us again focus on assignments. Subroutine calls will be treated later, when we look at interprocedural TBR analysis. As defined in section 4.4.1, the *AdjU* set of an assignment is empty if the assigned variable is not active after I. Otherwise, a variable is used in the adjoint of an assignment $a = \phi(B)$ if it appears in $\frac{\partial\phi}{\partial b_i}(B)$ or, in other words, if it appears in an expression e of $\phi(B)$ that is an argument of a non-linear operation such as `sin(`e`)` or e`*x` whose result is active. Furthermore, variables in the indices of the b_i and a are also used whenever the latter are array references. This leads us to the operational rules below, expressed recursively on the structure of the syntax tree:

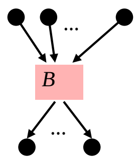
Kill(`T[exp]=b`) := $\emptyset$
Kill(`x=b`) := {`x`}

AdjU(`a=exp`) := **if** `exp` active **then** (*AdjU*(`a`)$\cup$*AdjU*(`exp`))$\setminus$*Kill*(`a=exp`) **else** $\emptyset$
(*and flag* `a` *to be recorded if it belongs to AdjU*(`exp`))
AdjU(`sin(a)`) := *VARS*(`a`)
AdjU(`a*b`) := (**if** `a` active **then** *VARS*(`b`)$\cup$ *AdjU*(`a`) **else** $\emptyset$) $\cup$
(**if** `b` active **then** *VARS*(`a`)$\cup$ *AdjU*(`b`) **else** $\emptyset$)
AdjU(`a+b`) := *AdjU*(`a`) $\cup$ *AdjU*(`b`)
AdjU(`T[exp]`) := *AdjU*(`T`) $\cup$ *VARS*(`exp`)
AdjU(*variable*) := $\emptyset$
AdjU(*constant*) := $\emptyset$

In the above, an expression is *active* when its value, considered as a temporary variable, is active. *VARS*(x) is the set of all variables occurring in expression x. The rules for the killed variables perform overestimation for arrays: if array region analysis is not performed, an assignment to one array cell does not kill the array.

Consider now the *AdjU* rule for an assignment, remember that the variable killed by the left-hand side represent different *mathematical* variables before and after this left-hand side. Therefore the variable after the assignment is brand new, not yet used by any adjoint instruction, and therefore actually erased from the *AdjU* set. The same reasoning applies inside a basic block, or for any sequence of structured program pieces (*cf* top of figure 4.20): *Kill* and *AdjU* sets are jointly defined by composition of the *Kill* and *AdjU* of these pieces. Variables in the *Kill* set are removed from the *AdjU* after the operation that actually overwrites them.

For a subroutine S, $AdjU(S)$ and $Kill(S)$ are built jointly and iteratively on the flow graph. For each basic block B, we introduce $InAdjU(B)$ (resp. $OutAdjU(B)$) and $InKill(B)$ (resp. $OutKill(B)$), the required and killed variables from the entry block of S to the *beginning* (resp. *end*) of B. The data flow equations are given in figure 4.19. They essentially state that a variable is required after basic block B if it is required on at least one path leading to B and is not killed in B, or else if it is required inside B. For every basic

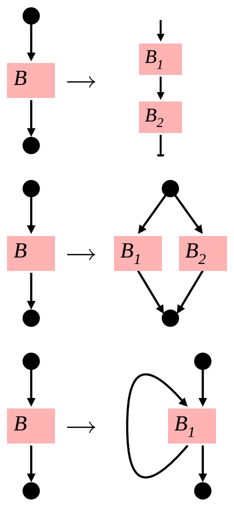


$$InAdjU(B) = \bigcup_{P \text{ predecessor of } B} OutAdjU(P)$$
$$InKill(B) = \bigcap_{P \text{ predecessor of } B} OutKill(P)$$
$$OutAdjU(B) = (InAdjU(B) \setminus Kill(B)) \cup AdjU(B)$$
$$OutKill(B) = InKill(B) \cup Kill(B)$$

Figure 4.19: *General data flow equations for bottom-up TBR analysis*

block B, $InAdjU(B)$, $OutAdjU(B)$, $InKill(B)$, and $OutKill(B)$ are initialized to $\emptyset$. Termination of the iterative resolution is granted by the fact that the computed sets are growing with respect to set inclusion and that they are bounded by the finite set of all variables present at this place in the program. After resolution, *OutAdjU* and *OutKill* of the exit block of S are exactly the desired $AdjU(S)$ and $Kill(S)$. In the case of structured flow graphs, the data flow equations can be specialized as shown in figure 4.20. No iteration is required.

At the call graph level, the *AdjU* and *Kill* sets of a call to a subroutine S are equal to $AdjU(S)$ and $Kill(S)$. This implies one bottom-up sweep for



$$AdjU(B) = (AdjU(B_1) \setminus Kill(B_2)) \cup AdjU(B_2)$$
$$Kill(B) = Kill(B_2) \cup Kill(B_1)$$

$$AdjU(B) = AdjU(B_1) \cup AdjU(B_2)$$
$$Kill(B) = Kill(B_1) \cap Kill(B_2)$$

$$AdjU(B) = AdjU(B_1)$$
$$Kill(B) = Kill(B_1)$$

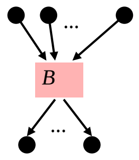
Figure 4.20: *Structured data flow equations for bottom-up TBR analysis*

acyclic call graphs.

Top-Down TBR Analysis

After the *AdjU* and *Kill* sets are synthesized, the second step of TBR analysis computes and propagates the *required* variables, i.e. those whose present value is possibly used by the adjoint of some previous instruction. *InReq*(`p`) (resp. *OutReq*(`p`)) denotes the set of the required variables just *before* (resp. *after*) a given program piece `p`. Each time an individual instruction *overwrites* a required variable (i.e. a variable present in the *InReq* set), we flag the overwritten variable as "to be recorded", and a `PUSH`/`POP` pair will be inserted. For the whole program `P`, *InReq*(`P`) is initialized to $\emptyset$, and is then propagated forwards on the program flow. Subroutines are swept top down on the call graph, in an order obtained by topological sorting. This ensures that a called subroutine is analyzed after *all* of its calling sites have been analyzed.

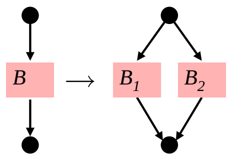
For a subroutine S, the data flow equations are shown in figure 4.21. Figure 4.22 shows specialized data flow equations for some sample structured flow graphs. The *InReq* set of the entry block of S is initialized to *InReq*(S), which is the union of the required variables before the call to S, on all calling contexts.



$$InReq(B) = \bigcup_{P \text{ predecessor of } B} OutReq(P)$$

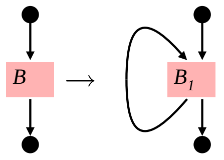
$$OutReq(B) = (InReq(B) \setminus Kill(B)) \cup AdjU(B)$$

Figure 4.21: *General data flow equations for top-down TBR analysis*



$$InReq(B_1) = InReq(B_2) = InReq(B)$$

$$OutReq(B) = OutReq(B_1) \cup OutReq(B_2)$$



$$InReq(B_1) = InReq(B) \cup AdjU(B_1)$$

$$OutReq(B) = (InReq(B) \setminus Kill(B_1)) \cup AdjU(B_1)$$

Figure 4.22: *Examples of structured data flow equations for top-down TBR analysis*

Observe that, unlike for dependencies (figure 4.15), the structured data flow equations for the loop are not iterative and therefore can be solved at low cost with no fixpoint. This result was demonstrated in [19, 39]. Here we reformulate the proof in the present formalism. The general data flow equations from figure 4.21, specialized to the structured loop of figure 4.22, are

$$\begin{aligned}&\mathit{InReq}(B_1) = \mathit{InReq}(B) \cup \mathit{OutReq}(B_1);\\ &\mathit{OutReq}(B_1) = (\mathit{InReq}(B_1) \setminus \mathit{Kill}(B_1)) \cup \mathit{AdjU}(B_1).\end{aligned}$$

Substituting $\mathit{OutReq}(B_1)$ into the first equation gives the fixpoint definition

$$\mathit{InReq}(B_1) = \mathit{InReq}(B) \cup (\mathit{InReq}(B_1) \setminus \mathit{Kill}(B_1)) \cup \mathit{AdjU}(B_1),$$

whose solution is the first data flow equation for structured loops:

$$\mathit{InReq}(B_1) = \mathit{InReq}(B) \cup \mathit{AdjU}(B_1).\ \square$$

Similarly, $\mathit{OutReq}(B)$ is equal to $\mathit{OutReq}(B_1)$ and

$$\mathit{OutReq}(B) = ((\mathit{InReq}(B) \cup \mathit{AdjU}(B_1)) \setminus \mathit{Kill}(B_1)) \cup \mathit{AdjU}(B_1),$$

which can be simplified to get the second data flow equation for structured loops:

$$\mathit{OutReq}(B) = (\mathit{InReq}(B) \setminus \mathit{Kill}(B_1)) \cup \mathit{AdjU}(B_1).\ \square$$

Finally, at the instruction level, the following rule propagates the "required" information forwards across any instruction I:

$$\mathit{OutReq}(I) = (\mathit{InReq}(I) \setminus \mathit{Kill}(I)) \cup \mathit{AdjU}(I)$$

and variables overwritten by I must be flagged as "to be recorded" if they belong to $\mathit{InReq}(I)$. In addition, for a subroutine call I : "`call` S`(...)`", the following rule accumulates the required information for the present calling context into $\mathit{InReq}(S)$. Top-down sweep on the call graph ensures that $\mathit{InReq}(S)$ will eventually contain the union for all calling contexts when S itself is analyzed.

$$\mathit{InReq}(S) = \mathit{InReq}(S) \cup \mathit{InReq}(I)$$

4.4.3 Optimal Snapshots

The above TBR analysis allows a reverse AD tool to reduce the size of the stack grossly by a constant factor: the proportion of active variables that are used only "linearly" in the program. This can be a gain of 10 to 20 percent.

But for large applications, we need more than just a constant factor. We definitely need nested checkpointing, which in theory can reduce the stack size logarithmically. For example in [23], Griewank shows that an optimal checkpointing for a loop of fixed length n, has a cost of only $O(log(n))$ in maximal stack size, and only $O(log(n))$ in maximal extra execution time.

We have similar complexity bounds in the more general case of a program made of a Call Tree with n subroutines, as shown on the left of figure 4.23. We choose to turn every subroutine or function call into a checkpointing piece "`p`", as defined in section 4.1.3 figure 4.6. This checkpointing scheme, known as the "*split mode*", is very easy to implement. The right part of figure 4.23 shows the Call Tree of the reverse differentiated program. When a forward sweep calls a subroutine `X`, it takes a snapshot to be able to run `X` again later, then calls the non-differentiated `X`. Later, during the backward sweep, it calls `X` again, but this time its forward sweep, immediately followed by `X`'s backward sweep. One can check easily that the maximum stack size,

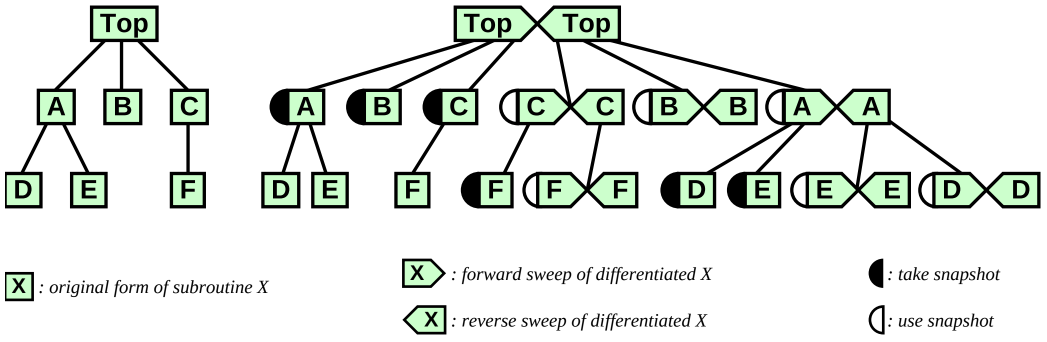

Top
A
B
C
D
E
F
X : original form of subroutine X
X : forward sweep of differentiated X
X : reverse sweep of differentiated X
: take snapshot
: use snapshot

Figure 4.23: *Recursive checkpointing scheme on a Call Tree*

due to recording of intermediate values *and* to snapshots, is proportional to the depth of the Call Tree. Similarly, the maximal number of duplicate executions of a subroutine is equal to its depth in the Call Tree. Finally, for well enough balanced Call Trees, this depth is proportional to the logarithm of the number n of subroutines, i.e. of the size of the program.

Our concern here is to keep the size of the snapshots low. A naive implementation would store the values of each and every variable around. This is definitely unacceptable for real applications. What is the smallest size of the snapshot? Strictly speaking, for a given checkpointing piece `p`, the goal is to run $\overline{\texttt{p}}$ later. With the notations of the classical Read-Written analysis, this requires that all variables in the "*Read*" set of $\overline{\mathtt{p}}$, $In(\overline{\mathtt{p}})$, be preserved. In these *Read* and *Written* sets, we consider only the original variables, and not their differentiated counterparts. Now how could one of these variables be altered? This can only happen if it is overwritten between the beginning of p and the beginning of $\overline{\mathtt{p}}$. Observing that p itself is included in the above checkpointing level *Above*(p) (it can be the Top level of the program), we call *Aft*(p) the piece of the original program from the exit of p to the end of *Above*(p). The piece of the reverse differentiated program which is executed between the beginning of p and the beginning of $\overline{\mathtt{p}}$ is therefore:

$$\mathtt{p};\overline{Aft(\mathtt{p})}.$$

Therefore, the variables that must be stored in the snapshot for p are:

$$Snapshot(\mathtt{p}) = Read(\overline{\mathtt{p}}) \cap Written(\mathtt{p};\overline{Aft(\mathtt{p})})$$

An easy-to-implement aproximation of *Snapshot*(p) is:

$$Snapshot(\mathtt{p}) \subset Read(\mathtt{p}) \cap Written(\text{Above}(\mathtt{p}))$$

This approximation is generally implemented in AD tools. Notice however that this is only an approximation. The standard Read-Written analysis often returns the *Read* and *Written* sets only from the beginning of a subroutine to its exit. However, it can be extended to compute from any point in a subroutine to the subroutine's exit. For example inside TAPENADE, this was done recently by Mauricio Araya. This allows us to use the better approximation:

$$Snapshot(\mathtt{p}) \subset Read(\mathtt{p}) \cap Written(\mathtt{p};Aft(\mathtt{p}))$$

Still this is not an exact approximation because, for all p:

$$Read(\overline{\mathtt{p}}) \subset Read(\mathtt{p}) \quad \text{and} \quad Written(\overline{\mathtt{p}}) \subset Written(\mathtt{p})$$

The proof for these two relations is straightforward. It comes from the fact that, even when p has results that are used after it, these results are not used after $\overline{\mathtt{p}}$, because we are in the backward sweep. On figure 4.23 for example, even if B has outputs that are later useful in C, these outputs are useless after $\overline{\mathtt{B}}$ because we will never reach C any more. Therefore some instructions of $\overline{\mathtt{p}}$ are in fact dead code, and should be removed. This reduces both the *Read* and *Written* sets. This is the topic of the next section 4.4.4. Moreover, $\overline{\mathtt{p}}$ has its own internal `PUSH/POP` mechanism to restore values. This reduces the *Written* set because it restores variables to their state at the entry of `p`. For example:

$$\mathit{Written}[\mathtt{PUSH(v);\ v=1/v;\ POP(v);\ \overline{v}=-\overline{v}/(v*v);}] = \emptyset \subset \mathit{Written}[\mathtt{v = 1/v;}]$$

A final remark is that checkpoints are usually many, nested or in sequence, and they can take profit of one another. Consider a program made of three successive phases $\mathtt{p}_1$; $\mathtt{p}_2$; $\mathtt{p}_3$. Suppose $\mathtt{p}_1$ and $\mathtt{p}_2$ are checkpointed. Then

$$\begin{array}{rcl} \mathit{Snapshot}(\mathtt{p}_1) & = & \mathit{Read}(\overline{\mathtt{p}_1}) \cap \mathit{Written}(\mathtt{p}_1; \overline{\mathtt{p}_2; \mathtt{p}_3}) \\ \mathit{Snapshot}(\mathtt{p}_2) & = & \mathit{Read}(\overline{\mathtt{p}_2}) \cap \mathit{Written}(\mathtt{p}_2; \overline{\mathtt{p}_3}) \end{array}$$

Suppose that a variable v is such that

$$v \in \mathit{Read}(\overline{\mathtt{p}_1}) \quad v \notin \mathit{Read}(\mathtt{p}_2) \quad v \notin \mathit{Written}(\mathtt{p}_2) \quad v \in \mathit{Written}(\overline{\mathtt{p}_3})$$

Then v is in the two snapshots, which is not necessary because it is the same value which is stored twice. v should be stored only in *Snapshot*($\mathtt{p}_2$), and retrieved twice. This becomes a difficult combinatorial problem between all snapshots present in a given differentiated program. This is still an open problem. Since large programs are bound to use a lot of snapshots, this question is gaining importance.

4.4.4 Dead Adjoint Code

Dead code elimination is a classical static transformation of programs, present in most compilers. This is a standard analysis applicable to all programs. However, programs differentiated in the reverse mode have specific properties that could be taken into account to obtain a better dead code elimination. In this section, we propose a specialized dead code analysis for reverse differentiated programs.

- The original outputs the main subroutine or function `P` are *not* outputs of the reverse differentiated $\overline{\mathtt{P}}$. The only outputs of $\overline{\mathtt{P}}$ we care about are the differentiated variables. This is also true for each checkpointing piece `p`, as can be checked on figure 4.23, beacuse the outputs of `p` are never used after $\overline{\mathtt{p}}$. Notice furthermore that the mere mechanism of reverse differentiation implies restoring previous values, therefore probably destroying the original outputs. The consequence for dead code is that an instruction that only contributes to an output, and does not contribute to the computation of a derivative, is dead code.

- When simplifying the reverse differentiated programs, one often encounters a `PUSH` followed by the corresponding `POP`. Similar situations arise with the "recompute-all" approach. This `PUSH/POP` pair is clearly dead code.

Let us illustrate the power of this specialized dead code elimination on an example taken from a real code. Figure 4.24 shows a small real routine (slightly shortened) taken from a Navier-Stokes flow solver. It contains a typical gather-scatter loop, which can account for many computations at run-time, and therefore many derivatives, because the number of mesh segments `NSG2-NSG1` may be large. We apply Reverse AD on a part of the program that includes subroutine `FLW2D1COL` above. We apply the "Store-All" approach, with the classical checkpointing scheme: one checkpoint for each subroutine call, as illustrated on figure 4.23. In this case, `FLW2D1COL` will eventually be differentiated, and figure 4.25 shows the resulting subroutine $\overline{\texttt{FLW2D1COL}}$. We assume that the call to subroutine `CHECK` does not create any loop-carried dependence, so that each iteration is actually independent. Anticipating on section 4.5.2, subroutine $\overline{\texttt{FLW2D1COL}}$ uses the special reverse mode for loops with Independent Iterations: Each iteration in the forward sweep is immediately followed by its corresponding backward sweep. Thus the forward sweep loop is merged with the backward sweep loop, resulting in a single loop. In subroutine $\overline{\texttt{FLW2D1COL}}$, Dead Adjoint Code detection would eliminate the lines on grey background. For example, the call to subroutine `CHECK` is dead code, because nothing depends on ots outputs. Therefore, there are two pairs of consecutive PUSH/POP that can be erased. Similarly, assignments to `rh3`, `dplim`, `rh2` are dead code because they are not used by live code, neither in the forward nor in the backward sweep. Notice however that assignment to `pm` is not dead, because `pm` is used in the backward sweep. Last but not least, the resulting simplified $\overline{\texttt{FLW2D1COL}}$ does not overwrite `rh3` nor `rh2` any more. Let us compare the *Read* and *Written* sets of `FLW2D1COL` and $\overline{\texttt{FLW2D1COL}}$. On the original subroutine, we have:

$$\mathit{Read}(\texttt{FLW2D1COL}) = \{\texttt{nsg1}, \texttt{nsg2}, \texttt{nubo}, \texttt{t3}, \texttt{pres}, \texttt{vnocl}, \texttt{g3}, \texttt{g4}, \texttt{rh3}, \texttt{rh4}, \texttt{sq}\}$$
$$\mathit{Written}(\texttt{FLW2D1COL}) = \{\texttt{rh3}, \texttt{rh4}, \texttt{sq}\}$$

However, on the simplified $\overline{\texttt{FLW2D1COL}}$, we have:

```
      subroutine FLW2D1COL(nsg1,nsg2,nubo,t3,pres,vnocl,
     +   g3,g4,rh3,rh4,ns,nseg,sq)
      integer nsg1,nsg2,ns,nseg,nubo(2,nseg),is1,is2
      real*8 t3(ns),g3(ns),g4(ns),rh3(ns),rh4(ns),pres(ns)
      real*8 vnocl(2,nseg),qsor,qsex,pm,dplim,sq

      do 30 iseg=nsg1,nsg2
         is1 = nubo(1,iseg)
         is2 = nubo(2,iseg)
         qsor = t3(is1)*vnocl(2,iseg)
         qsex = t3(is2)*vnocl(2,iseg)
         dplim = qsor*g4(is1)+qsex*g4(is2)
         rh4(is1) = rh4(is1) + dplim
         rh4(is2) = rh4(is2) - dplim
         pm = pres(is1)+pres(is2)
         dplim = qsor*g3(is1)+qsex*g3(is2)+pm*vnocl(2,iseg)
         rh3(is1) = rh3(is1) + dplim
         rh3(is2) = rh3(is2) - dplim
         call CHECK(pm,sq)
30    continue
      end
```

Figure 4.24: *An example gather-scatter loop from a real code*

```
  subroutine F̅L̅W̅2̅D̅1̅C̅O̅L̅(nsg1,nsg2 nubo,t3,t̅3̅,pres,p̅r̅e̅s̅,vnocl,
+   v̅n̅o̅c̅l̅,g3,g̅3̅,g4,g̅4̅,rh3,r̅h̅3̅,rh4,r̅h̅4̅,ns,nseg,sq,s̅q̅)
  < omitted declarations >
  do iseg=nsg1,nsg2
    is1 = nubo(1,iseg)
    is2 = nubo(2,iseg)
    qsor = t3(is1)*vnocl(2,iseg)
    qsex = t3(is2)*vnocl(2,iseg)
    dplim = qsor*g4(is1) + qsex*g4(is2)
    rh4(is1) = rh4(is1) + dplim
    rh4(is2) = rh4(is2) - dplim
    pm = pres(is1) + pres(is2)
    dplim = qsor*g3(is1)+qsex*g3(is2)+pm*vnocl(2,iseg)
    rh3(is1) = rh3(is1) + dplim
    rh3(is2) = rh3(is2) - dplim
    call PUSH(sq)
    call PUSH(pm)
    call CHECK(pm, sq)
  < forward sweep ends, backward sweep begins >
    call POP(pm)
    call POP(sq)
    call C̅H̅E̅C̅K̅(pm, p̅m̅, sq, s̅q̅)
    d̅p̅l̅i̅m̅ = r̅h̅3̅(is1) - r̅h̅3̅(is2)
    q̅s̅o̅r̅ = g3(is1)*d̅p̅l̅i̅m̅
    g̅3̅(is1) = g̅3̅(is1) + qsor*d̅p̅l̅i̅m̅
    q̅s̅e̅x̅ = g3(is2)*d̅p̅l̅i̅m̅
    g̅3̅(is2) = g̅3̅(is2) + qsex*d̅p̅l̅i̅m̅
    p̅m̅ = p̅m̅ + vnocl(2,iseg)*d̅p̅l̅i̅m̅
    v̅n̅o̅c̅l̅(2,iseg) = v̅n̅o̅c̅l̅(2,iseg) + pm*d̅p̅l̅i̅m̅
    p̅r̅e̅s̅(is1) = p̅r̅e̅s̅(is1) + p̅m̅
    p̅r̅e̅s̅(is2) = p̅r̅e̅s̅(is2) + p̅m̅
    d̅p̅l̅i̅m̅ = r̅h̅4̅(is1) - r̅h̅4̅(is2)
    q̅s̅o̅r̅ = q̅s̅o̅r̅ + g4(is1)*d̅p̅l̅i̅m̅
    g̅4̅(is1) = g̅4̅(is1) + qsor*d̅p̅l̅i̅m̅
    q̅s̅e̅x̅ = q̅s̅e̅x̅ + g4(is2)*d̅p̅l̅i̅m̅
    g̅4̅(is2) = g̅4̅(is2) + qsex*d̅p̅l̅i̅m̅
    t̅3̅(is2) = t̅3̅(is2) + vnocl(2,iseg)*q̅s̅e̅x̅
    v̅n̅o̅c̅l̅(2,iseg) = v̅n̅o̅c̅l̅(2,iseg)+t3(is2)*q̅s̅e̅x̅+t3(is1)*q̅s̅o̅r̅
    t̅3̅(is1) = t̅3̅(is1) + vnocl(2,iseg)*q̅s̅o̅r̅
  enddo
  end
```

Figure 4.25: *Reverse differentiation of subroutine* FLW2D1COL *from figure 4.24*

$$\begin{aligned}
\textit{Read}(\overline{\texttt{FLW2D1COL}}) &= \{\texttt{nsg1}, \texttt{nsg2}, \texttt{nubo}, \texttt{t3}, \texttt{pres}, \texttt{vnocl}, \texttt{g3}, \texttt{g4}\} \cup \\
&\qquad (\{\texttt{sq}\} \cap \textit{Read}(\overline{\texttt{CHECK}})) \\
\textit{Written}(\overline{\texttt{FLW2D1COL}}) &= \{\texttt{sq}\} \cap \textit{Written}(\overline{\texttt{CHECK}})
\end{aligned}$$

We saw in section 4.4.3 that these smaller *Read* and *Written* sets allow for smaller snapshots before the calls to `FLW2D1COL` and $\overline{\texttt{FLW2D1COL}}$.

4.5 Using the Dependence Graph in AD

In this section, we focus on the use of the Dependence Graph, a tool from the domain of parallelization, applied to Automatic Differentiation. On one hand, the Dependence Graph is a conceptual tool, that describes exactly the possible rescheduling of instructions. On the other hand, it is also a practical tool actually built during parallelization or AD, that allows for very precise transformations of programs. However a Dependence Graph is expensive, is computation time as well as memory space, and few applications actually build it completely on large programs. For example in section 4.5.1, the Dependence Graph is used to move differentiated instructions around, inside a given Basic Block. It is thus built only for this Basic Block. In section 4.5.2, we use the Dependence Graph more as a conceptual tool, studying the relationship between the Dependence graph of a program one one hand, of the adjoint program on the other hand. Computation and memory cost of the Dependence Graph are not important here. In section 4.5.3, we highlight the relationship, and more importantly the differences, between the Dependence Graph and the Computational Graph, which is used frequently by AD researchers. Confusion between these graphs lead to misunderstanding, and should be avoided.

4.5.1 Merging Differentiated Instructions

We present two instruction-level improvements of differentiated programs, which are more clearly implemented on the basis of the local Dependence Graph of each Basic Block. The first improvement concerns the reverse mode, the second improvement concerns the so-called “vector” or multi-directional mode.

Non-incremental adjoint instructions

Equation (4.5) of section 4.1.3 defines what must be computed by the adjoint instructions (backward sweep) of a reverse differentiated program. Straightforward evaluation of equation (4.5) is built with successive steps (multi-instructions $\overline{I}_k$) that compute:

$$\overline{X}_{k-1} = J(I_k)^t(X_{k-1}).\overline{X}_k \quad \textit{for } k = p \textit{ down to 1}$$

where $J(I_k)^t$ is the transposed of the Jacobian of instruction I_k. It is easy to see that these $\overline{I}_k$ contain a large number of *incrementation* instructions. Consider an instruction I which, like an assignment, uses several variables but overwrites only one. If the assigned variable is put first, the local Jacobian of I has the following shape:

$$J(I) \;=\; \begin{pmatrix} \bullet & \bullet & \bullet & \cdots & \bullet \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{pmatrix} \tag{4.15}$$

where dots "•" represent entries that may be non null, "1" represent entries that are exactly one, and the other entries are always null. Calling v_n the variables involved in I (v_1 assigned), and $\overline{v_n}$ their adjoint variables, the multi-instruction $\overline{I}$ must compute:

$$\begin{pmatrix} \overline{v_1} \\ \overline{v_2} \\ \overline{v_3} \\ \vdots \\ \overline{v_q} \end{pmatrix} := J(I)^t \;\times\; \begin{pmatrix} \overline{v_1} \\ \overline{v_2} \\ \overline{v_3} \\ \vdots \\ \overline{v_q} \end{pmatrix} = \begin{pmatrix} \bullet & & & & \\ \bullet & 1 & & & \\ \bullet & & 1 & & \\ \vdots & & & \ddots & \\ \bullet & & & & 1 \end{pmatrix} \times \begin{pmatrix} \overline{v_1} \\ \overline{v_2} \\ \overline{v_3} \\ \vdots \\ \overline{v_q} \end{pmatrix} \tag{4.16}$$

In a computer program, this Matrix×Vector product must be evaluated from the last row up, because $\overline{v_1}$ must not be overwritten before it is used by the other Row×Vector products. Each row, except for the topmost row, generates an instruction of the shape:

$$\overline{v_n} := \overline{v_n} + \textit{exp}{*}\overline{v_1}$$

which is an incrementation (*exp* does not contain $\overline{v_n}$).

Thus the following pattern ofter occurs in the backwards sweep: an adjoint variable $\overline{v_n}$ is set to some value (often 0.0), and then repeatedly incremented without being used, then it is used and possibly reset, etc... If the original "set" instruction and the following "increment" instructions can be moved around to form a sequence, then they can be merged into one single instruction. This results in the so-called *non-incremental* style (*cf* [24, Section 3.3]), by opposition with the original program's *incremental* style. The resulting program is more efficient, avoiding many storages and retrievals of $\overline{v_n}$. Of course an optimizing compiler can partly do this, but we can do it better using knowledge on the structure of the backward sweep.

The right tool to decide how instructions can be moved around is the Dependence Graph. Differentiation on a subroutine's Flow Graph actually produces a new Flow Graph. Each of the Basic Blocks of the backward sweep can be easily analyzed to get its local Dependence Graph, which is acyclic since it is local to the Block. In these blocks only appear overwrites of adjoint variables $\overline{v_n}$, and retrievals (`POP`'s) of original variables. The Dependence Graph is thus easily built. The granularity we choose for the Dependence Graph is of course one node per instruction: we don't aim at reordering expressions in instructions. On the dependence Graph, we look for an execution order (i.e. a topological sorting) which maximizes the number of "set;increment*" sequences. Finding the optimal order is probably NP-hard, but a simple greedy heuristic gives excellent results.

Illustration of this can be found on the example of section 4.4.4. Many adjoint instruction in the differentiated program $\overline{\texttt{FLW2D1COL}}$ have been merged. For example, instruction

$\overline{\texttt{dplim}}\texttt{ = }\overline{\texttt{rh3}}\texttt{(is1) - }\overline{\texttt{rh3}}\texttt{(is2)}$

results from the fusion of the three adjoint instructions

$\overline{\texttt{dplim}}\texttt{ = 0.0}$
$\overline{\texttt{dplim}}\texttt{ = }\overline{\texttt{dplim}}\texttt{ - }\overline{\texttt{rh3}}\texttt{(is2)}$
$\overline{\texttt{dplim}}\texttt{ = }\overline{\texttt{dplim}}\texttt{ + }\overline{\texttt{rh3}}\texttt{(is1)}$

Backwards sweeps transformed in this manner often look more like hand-written adjoints.

Loop fusion in "vector" Automatic Differentiation

In standard AD, we compute in tangent mode $\dot{Y} = f'(X).\dot{X}$ for a given $\dot{X}$ and in reverse mode $\overline{X} = f'(X)^t.\overline{Y}$ for a given $\overline{Y}$. Many applications require this for several vectors $\dot{X}$ or $\overline{Y}$, at the same time, for the same "point" X in the input space. One example is to get the whole Jacobian $f'(X)$, column by column: for the same X, one runs the differentiated tangent program n times, with $\dot{X}$ spanning all the canonical basis of $I\!R^n$. One can use fewer runs when the sparsity pattern of $f'(X)$ is known.

In any case, this amounts to several runs of the tangent (resp. adjoint) program, for the same X. If X is the same, all instructions from the original program run identically, on the same inputs, yielding the same results. This is a waste of time. The so-called "vector" or "multi-directional" mode of AD builds a program which embeds each derivative instruction into a small loop, that repeats it for each of the original $\dot{X}$ (resp. $\overline{Y}$). Figure 4.26 illustrates this "vector" mode on the same program as figure 4.1. Notice

original: `T,U` $\mapsto$ `e`	**vector tangent mode:** `T,`$\dot{\texttt{T}}^{\texttt{nd}}$`,U,`$\dot{\texttt{U}}^{\texttt{nd}}$ $\mapsto$ `e,`$\dot{\texttt{e}}^{\texttt{nd}}$
	`do j=1,nd`
	`   ė2(j) = 0.0`
	`end do`
`e2 = 0.0`	`e2 = 0.0`
`do i=1,n`	`do i=1,n`
	`   do j=1,nd`
	`      ė1(j) = Ṫ(j,i)-U̇(j,i)`
	`   end do`
` e1 = T(i)-U(i)`	`   e1 = T(i)-U(i)`
	`   do j=1,nd`
	`      ė2(j) = ė2(j) + 2.0*e1*ė1(j)`
	`   end do`
` e2 = e2 + e1*e1`	`   e2 = e2 + e1*e1`
`end do`	`end do`
	`do j=1,nd`
	`   ė(j) = 0.5*ė2(j)/SQRT(e2)`
	`end do`
`e = SQRT(e2)`	`e = SQRT(e2)`

Figure 4.26: *AD in vector tangent mode*

the loops appearing around the derivative instructions. All differentiated variables become arrays, since they are now dimensioned after the number `nd` of directions of simultaneous differentiation.

It is very tempting to try instructions reordering here: all these loops around a single derivative instruction incur a large loop overhead. Here too, the tool is the Dependence Graph, build locally on each differentiated Basic Block. The goal is to find the topological sorting of the Block's Dependence Graph that maximizes sequences of derivative loops, so that these loops can be fused. Loop fusion is allowed here, because all these loops have data-independent iterations, and they share the same iteration space `j=1,nd`. For example on figure 4.26, the first two instructions of the central loop can be switched, because it violates no data dependence (no use of `ė1` in the second instruction, and no use of `e1` in the first instruction. After loop fusion, we get this more efficient main loop body:

```
e1 = T(i)-U(i)
do j=1,nd
   ė1(j) = Ṫ(j,i)-U̇(j,i)
   ė2(j) = ė2(j) + 2.0*e1*ė1(j)
end do
e2 = e2 + e1*e1
```

4.5.2 The Adjoint Dependence Graph

In this section, we use the Dependence Graph more like a conceptual tool to demonstrate properties of adjoint differentiated programs. The fundamental property that we demonstrate is an isomorphism between the Dependence Graph of a (piece of a) program and the Dependence Graph of the backward sweep of its reverse differentiation. This was presented in report [30]. We then examine some consequences of this property for actual AD tools. In particular, we show a specific optimized reverse differentiation for parallel loops, published in [32]. We also discuss some issues related to preserving the parallel properties of the original code for its reverse differentiated version.

Adjoint Dependence Graph Isomorphism

We consider a program piece `p`. We call $\mathcal{G}$ the dependence graph of `p`, built with one node per atomic instruction I_k (e.g. assignment) of `p`. The backward sweep of the reverse differentiation on `p`, that we shall write $\overleftarrow{\mathtt{p}}$, also has a dependence graph $\overleftarrow{\mathcal{G}}$. We define the nodes of $\overleftarrow{\mathcal{G}}$ to be the adjoint $\overline{I_k}$ of the nodes I_k of $\mathcal{G}$. Notice that these nodes $\overline{I_k}$ are no longer single assignments like the I_k: as we saw in section 4.5.1, it may take several assignments to implement the vector assignment of equation (4.16). However, both I_k and $\overline{I_k}$ are *valid* is the following sense: they don't overwrite a variable and then use it. Only values that exist *before* this node are used by this node. In this respect, they can be considered as a simple assignment rather than a complex program. We will show that if I_k is *valid*, then $\overline{I_k}$ is lid too. The I_k and $\overline{I_k}$ are the nodes of dependence graph $\mathcal{G}$ and $\overleftarrow{\mathcal{G}}$. This means we will never consider splitting or rearranging their contents.

Let us formalize further data dependences, which are the arrows of the dependence graphs. In the sequel, we consider that the nodes of the dependence graphs are *valid* original or adjoint instructions I. Like we saw in section 3.3.1, data dependences come from accesses to variables. Each data dependences is *caused* by one or many variables. A variable v causes a data dependence between two nodes I_1 and I_2 if I_1 can be executed before I_2 (notation: $I_1 \prec I_2$), and if I_1 and I_2 both perform a *read* or a *write* of v, at least one of them performing a *write.* If I_1 and I_2 both only *read* v, v causes no dependence because, as far as v is concerned, I_1 and I_2 can be executed in any order without changing the result.

Similarly, we introduce the following refinement to data dependences: If I_1 and I_2 both only *increment* v, v causes no dependence between I_1 and I_2. In other words, two increments of v, with no access v between them, can be done in any order without changing the result. This deserves some discussion: this is true only if increments are *atomic.* If this is not the case, two increment operations done in parallel may create a race condition. In the following, we assume increments are atomic. This can be achieved at low level, using semaphores, or at high level, using *reduction* declarations. Also, atomicity is granted when the program is run sequentially.

To give a formal definition of data dependences, that takes profit of *increment* operations, we define the *effect* of any valid instruction I on any variable v, $E(I, v)$, by the following exclusive four cases:

- $E(I, v) = \textcircled{\textbf{n}}$ (*no access*) when variable v does not occur at all inside I.
- $E(I, v) = \textcircled{\textbf{r}}$ (*only read*) when v is only read and not overwritten in I.
- $E(I, v) = \textcircled{\textbf{i}}$ (*pure increment*) when v is read only once. Its value is incremented, and is reassigned to v. There are no other occurrences of v in I.
- $E(I, v) = \textcircled{\textbf{w}}$ (*general write*) otherwise.

Given two valid instructions I_1 and I_2, with $I_1 \prec I_2$, there is a dependence from I_1 to I_2 caused by variable v if and only if $\mathcal{D}(E(I_1, v), E(I_1, v))$ is true, where relation $\mathcal{D}$ is defined by the table of figure 4.27. One can easily check

	ⓦ	ⓡ	ⓘ	ⓝ
ⓦ	*true*	*true*	*true*	
ⓡ	*true*		*true*	
ⓘ	*true*	*true*		
ⓝ				

Figure 4.27: *Definition of data dependence, based on the effect of instructions on a given variable*

that two successive increments, or two successive reads, separated only by instructions that don't access the variable, cause no dependence. However, as one can expect, if there an intermediate instruction that accesses the variable, then there are dependences going from the first to the intermediate instruction, and then from the intermediate instruction to the second instruction, and no reordering is allowed. In this context, let us demonstrate our first lemma:

Lemma 1 ***(validity of adjoint instructions)*** *If the set of successive instructions I is valid, then the its adjoint set of instructions $\overline{I}$ is valid too.*

Proof:
We must show that $\overline{I}$ contains no variable which is overwritten and then used, inside the same execution of $\overline{I}$. Consider the list $v_k, k \in [1..q]$ of all variables occurring in p. We can suppose with no loss of generality that the v_k are reordered so that the first variable written during an execution of I is v_1, the next one v_2, and so on, and the variables that are only read or not used come at the end. Since I is valid, no variable is written and then used, inside the same execution of I. Therefore the value assigned to some variable v_k (if any) does not depend on the $v_i, i < k$. The Jacobian matrix of I is

thus upper triangular:

$$J(I) \;=\; \begin{pmatrix} \bullet & \bullet & \bullet & \cdots & \bullet \\ & \bullet & \bullet & \cdots & \bullet \\ & & \bullet & \cdots & \bullet \\ & & & \ddots & \vdots \\ & & & & \bullet \end{pmatrix} \tag{4.17}$$

Dots "•" represent non-zero entries in the matrix. By definition of the reverse mode of AD (*cf* section 4.1.3), the adjoint instruction $\overline{I}$ implements the following vector assignment (cf equation (4.5)):

$$\begin{pmatrix} \overline{v_1} \\ \overline{v_2} \\ \overline{v_3} \\ \vdots \\ \overline{v_q} \end{pmatrix} := J(I)^t \;\times\; \begin{pmatrix} \overline{v_1} \\ \overline{v_2} \\ \overline{v_3} \\ \vdots \\ \overline{v_q} \end{pmatrix} = \begin{pmatrix} \bullet & & & & \\ \bullet & \bullet & & & \\ \bullet & \bullet & \bullet & & \\ \vdots & \vdots & \vdots & \ddots & \\ \bullet & \bullet & \bullet & \cdots & \bullet \end{pmatrix} \times \begin{pmatrix} \overline{v_1} \\ \overline{v_2} \\ \overline{v_3} \\ \vdots \\ \overline{v_q} \end{pmatrix} \tag{4.18}$$

By a simple change of coordinates, reversing the order of the adjoint variables $\overline{v_k}$, equation (4.18) uses an upper triangular matrix again.

$$\begin{pmatrix} \overline{v_q} \\ \vdots \\ \overline{v_3} \\ \overline{v_2} \\ \overline{v_1} \end{pmatrix} := \begin{pmatrix} \bullet & \cdots & \bullet & \bullet & \bullet \\ & \ddots & \vdots & \vdots & \vdots \\ & & \bullet & \bullet & \bullet \\ & & & \bullet & \bullet \\ & & & & \bullet \end{pmatrix} \times \begin{pmatrix} \overline{v_q} \\ \vdots \\ \overline{v_3} \\ \overline{v_2} \\ \overline{v_1} \end{pmatrix} \tag{4.19}$$

Therefore if the Row×Vector products are done from the top row down, no variable is used after it is overwritten, and $\overline{I}$ is valid. □

Now we relate the effect $E(I, v)$ of an instruction I on a variable v, with the effect $E(\overline{I}, \overline{v})$ of the adjoint instruction $\overline{I}$ on the adjoint variable $\overline{v}$.

Lemma 2 ***(effect of adjoint instructions)*** *the effect on $\overline{v}$ of the adjoint $\overline{I}$ of a given data dependence node I is related to the effect of I on v in the following manner:*

- $E(I, v) = \textcircled{\textbf{n}} \implies E(\overline{I}, \overline{v}) = \textcircled{\textbf{n}}$
- $E(I, v) = \textcircled{\textbf{r}} \implies E(\overline{I}, \overline{v}) \in \{\textcircled{\textbf{i}}, \textcircled{\textbf{n}}\}$
- $E(I, v) = \textcircled{\textbf{i}} \implies E(\overline{I}, \overline{v}) \in \{\textcircled{\textbf{r}}, \textcircled{\textbf{n}}\}$

- $E(I,v) =$ ⓦ $\implies E(\overline{I},\overline{v}) \in \{$ⓦ,ⓡ,ⓘ,ⓝ$\}$

Proof:
Consider any variable v_k. Since I is valid, the effect of I on v_k is defined, and belongs to $\{$ⓦ,ⓡ,ⓘ,ⓝ$\}$. Let us focus on the k-th row and column of the Jacobian matrix $J(I)$ of equation (4.17). If $E(I,v_k) =$ ⓝ, then these row and column are all 0, except the diagonal element, which is 1. If $E(I,v_k) =$ ⓡ, then the k-th row is all 0, with 1 on the diagonal. If $E(I,v_k) =$ ⓘ, then the k-th column is all 0, with 1 on the diagonal. And if $E(I,v_k) =$ ⓦ, the k-th row and column can be anything. We summarize this as:

$E(I,v_k)$:	ⓝ	ⓘ	ⓡ	ⓦ
$J(I)$:	$\begin{pmatrix}\ddots & 0 & \\ & 1 & 0 \\ & & \ddots\end{pmatrix}$	$\begin{pmatrix}\ddots & 0 & \\ & 1 & \bullet \\ & & \ddots\end{pmatrix}$	$\begin{pmatrix}\ddots & \bullet & \\ & 1 & 0 \\ & & \ddots\end{pmatrix}$	$\begin{pmatrix}\ddots & \bullet & \\ & \bullet & \bullet \\ & & \ddots\end{pmatrix}$

where the $\bullet$ elements can very well be 0 or 1, as will be shown below on some degenerate cases. By definition, $\overline{I}$ implements equation (4.19). We observe that the matrix in equation (4.19) (call it $\overline{J(I)}$), is the symmetric of $J(I)$ with respect to the second diagonal. We have thus:

$E(I,v_k)$:	ⓝ	ⓘ	ⓡ	ⓦ
$\overline{J(I)}$:	$\begin{pmatrix}\ddots & 0 & \\ & 1 & 0 \\ & & \ddots\end{pmatrix}$	$\begin{pmatrix}\ddots & \bullet & \\ & 1 & 0 \\ & & \ddots\end{pmatrix}$	$\begin{pmatrix}\ddots & 0 & \\ & 1 & \bullet \\ & & \ddots\end{pmatrix}$	$\begin{pmatrix}\ddots & \bullet & \\ & \bullet & \bullet \\ & & \ddots\end{pmatrix}$

which implies the following:

- when $E(I,v_k) =$ ⓝ, $\overline{I}$ simply does not use nor modify $\overline{v_k}$. Therefore $\overline{v_k}$ does not occur in $\overline{I}$ and $E(\overline{I},\overline{v_k}) =$ ⓝ.

- when $E(I,v_k) =$ ⓡ, $\overline{I}$ does not read $\overline{v_k}$, except in one instruction which adds some value into $\overline{v_k}$. In the special case where the $\bullet$ values are all 0, $\overline{v_k}$ is just unmodified. Therefore $E(\overline{I},\overline{v_k}) \in \{$ⓘ,ⓝ$\}$.

- when $E(I,v_k) =$ ⓘ, $\overline{I}$ may read $\overline{v_k}$ several times, to compute values assigned to other adjoint variables, and then $\overline{v_k}$ itself is just unmodified. In the special case where the $\bullet$ values are all 0, $\overline{v_k}$ is not used at all. Therefore $E(\overline{I},\overline{v_k}) \in \{$ⓡ,ⓝ$\}$.

- when $E(I, v_k) = \textcircled{w}$, $\overline{I}$ may read $\overline{v_k}$ several times, and then overwrite it with some value, and then not use it any more. Therefore $E(\overline{I}, \overline{v_k})$ may be $\textcircled{w}$. However, since any of the $\bullet$ elements may be 0, and the diagonal $\bullet$ may be 1, $E(\overline{I}, \overline{v_k})$ may degenerate to $\textcircled{r}$, $\textcircled{i}$, or even $\textcircled{n}$. Therefore $E(\overline{I}, \overline{v_k}) \in \{\textcircled{w}, \textcircled{r}, \textcircled{i}, \textcircled{n}\}$. $\square$

Here are two examples of degenerate cases. Suppose I is instruction `y:=floor(x)`, where `floor` returns the integer part of a real number. The derivative of `floor`, when defined, is

$$\frac{\partial \texttt{floor(x)}}{\partial \texttt{x}} = 0$$

Instruction I is valid, and $E(I, \texttt{x}) = \textcircled{r}$. With respect to vector $(\texttt{y}, \texttt{x})$, $J(I)$ is equal to $\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$. Therefore $\overline{I}$ implements $\begin{pmatrix} \overline{\texttt{x}} \\ \overline{\texttt{y}} \end{pmatrix} := \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \times \begin{pmatrix} \overline{\texttt{x}} \\ \overline{\texttt{y}} \end{pmatrix}$, thus $\overline{I}$ contains only instruction $\overline{\texttt{y}}$`:=0`, and $E(\overline{I}, \overline{\texttt{x}}) = \textcircled{n}$.

As a second example, suppose I contains the two successive instructions `y:=2*x; x:=x+floor(x)`. Node I is valid, and $E(I, \texttt{x}) = \textcircled{w}$. On vector $(\texttt{y}, \texttt{x})$, $J(I)$ is $\begin{pmatrix} 0 & 2 \\ 0 & 1 \end{pmatrix}$, and $\overline{I}$ implements $\begin{pmatrix} \overline{\texttt{x}} \\ \overline{\texttt{y}} \end{pmatrix} := \begin{pmatrix} 1 & 2 \\ 0 & 0 \end{pmatrix} \times \begin{pmatrix} \overline{\texttt{x}} \\ \overline{\texttt{y}} \end{pmatrix}$, thus $\overline{I}$ contains instruction $\overline{\texttt{x}}$`:=`$\overline{\texttt{x}}$ `+2*`$\overline{\texttt{y}}$ followed by $\overline{\texttt{y}}$`:=0`, and $E(\overline{I}, \overline{\texttt{x}}) = \textcircled{i}$.

From lemma (2), we are now able to go from the effect of adjoint instructions back to the effect of the original instructions.

Lemma 3 ***(effect of original instructions)*** *The effect of an instruction I on a variable v can be deduced from the effect of its adjoint $\overline{I}$ on $\overline{v}$ in the following manner:*

- $E(\overline{I}, \overline{v}) = \textcircled{n} \implies E(I, v) \in \{\textcircled{w}, \textcircled{r}, \textcircled{i}, \textcircled{n}\}$
- $E(\overline{I}, \overline{v}) = \textcircled{r} \implies E(I, v) \in \{\textcircled{w}, \textcircled{i}\}$
- $E(\overline{I}, \overline{v}) = \textcircled{i} \implies E(I, v) \in \{\textcircled{w}, \textcircled{r}\}$
- $E(\overline{I}, \overline{v}) = \textcircled{w} \implies E(I, v) = \textcircled{w}$

Proof:
Since I and $\overline{I}$ are valid, their effect on variables v and $\overline{v}$ is defined, and must be one of $\textcircled{w}$, $\textcircled{r}$, $\textcircled{i}$, or $\textcircled{n}$. Therefore, it suffices to explore all possible cases in lemma (2). $\square$

We now use our refined notion of data dependence. We insist this supposes that incrementation operations ($\textcircled{i}$) are *atomic*. With this hypothesis, which can be enforced in various ways, we prove that the dependence graph of the adjoint program is isomorphic to a subgraph of the original dependence graph. This will allow us to transpose many properties of the original program to its adjoint.

Proposition 1 ***(adjoint data dependences)*** *If $\overleftarrow{\mathcal{G}}$ has an arrow from node $\overline{I_a}$ to node $\overline{I_b}$, then $\mathcal{G}$ has an arrow from node I_b to node I_a. Moreover, if the arrow from $\overline{I_a}$ to $\overline{I_b}$ is caused by a variable w, then the arrow from I_b to I_a is caused by a variable v such that $w = \overline{v}$.*

Proof:
Consider an arrow in $\overleftarrow{\mathcal{G}}$, going from node $\overline{I_a}$ to node $\overline{I_b}$. By definition of data dependence, this implies that $\overline{I_a} \prec \overline{I_b}$, which in turn implies by construction of the adjoint program that $I_b \prec I_a$. By definition, the data dependence is motivated by (at least) one variable w and means that for any such variable w, the effects of $\overline{I_a}$ and $\overline{I_b}$ on w are such that $\mathcal{D}(E(\overline{I_a}, w), E(\overline{I_b}, w))$ is true, where $\mathcal{D}$ is defined by the table of figure 4.27. This implies that variable w is assigned, somewhere in $\overline{I_a}$, $\overline{I_b}$, or both. Since the adjoint nodes $\overline{I_j}$ only assign adjoint variables, w is necessarily the adjoint $\overline{v}$ of some variable v in P. Considering each case where $\mathcal{D}(E(\overline{I_a}, \overline{v}), E(\overline{I_b}, \overline{v}))$, we check that $\mathcal{D}(E(I_b, v), E(I_a, v))$. For example, suppose $E(\overline{I_a}, \overline{v}) = \textcircled{w}$ and $E(\overline{I_b}, \overline{v}) = \textcircled{i}$. By lemma (2), we get $E(I_a, v) \in \{\textcircled{i}, \textcircled{w}\}$ and $E(I_b, v) \in \{\textcircled{r}, \textcircled{w}\}$, and in the four resulting cases, we can check that $\mathcal{D}(E(I_b, v), E(I_a, v))$. Together with $I_b \prec I_a$, this shows that there is an arrow in $\mathcal{G}$, motivated by v, that goes from I_b to I_a. $\square$

To illustrate this proposition, consider subroutine `FLW2D1COL` from figure 4.24 in section 4.4.4. Figure 4.28 show its dependence graph $\mathcal{G}$. Figure 4.29 shows the dependence graph $\overleftarrow{\mathcal{G}}$ of the backward sweep of its reverse differentiated version. To make comparison easier on the figures, differentiated nodes occupy the same location as their original nodes. We observe that every data dependence in $\overleftarrow{\mathcal{G}}$ is caused by an adjoint variable $\overline{v}$, such that v causes a data dependence between the corresponding nodes in $\mathcal{G}$ in the reverse direction. The distances of the data dependences are preserved. However, the “kind” of dependence (**flow**, **anti**, **output**) is not preserved.

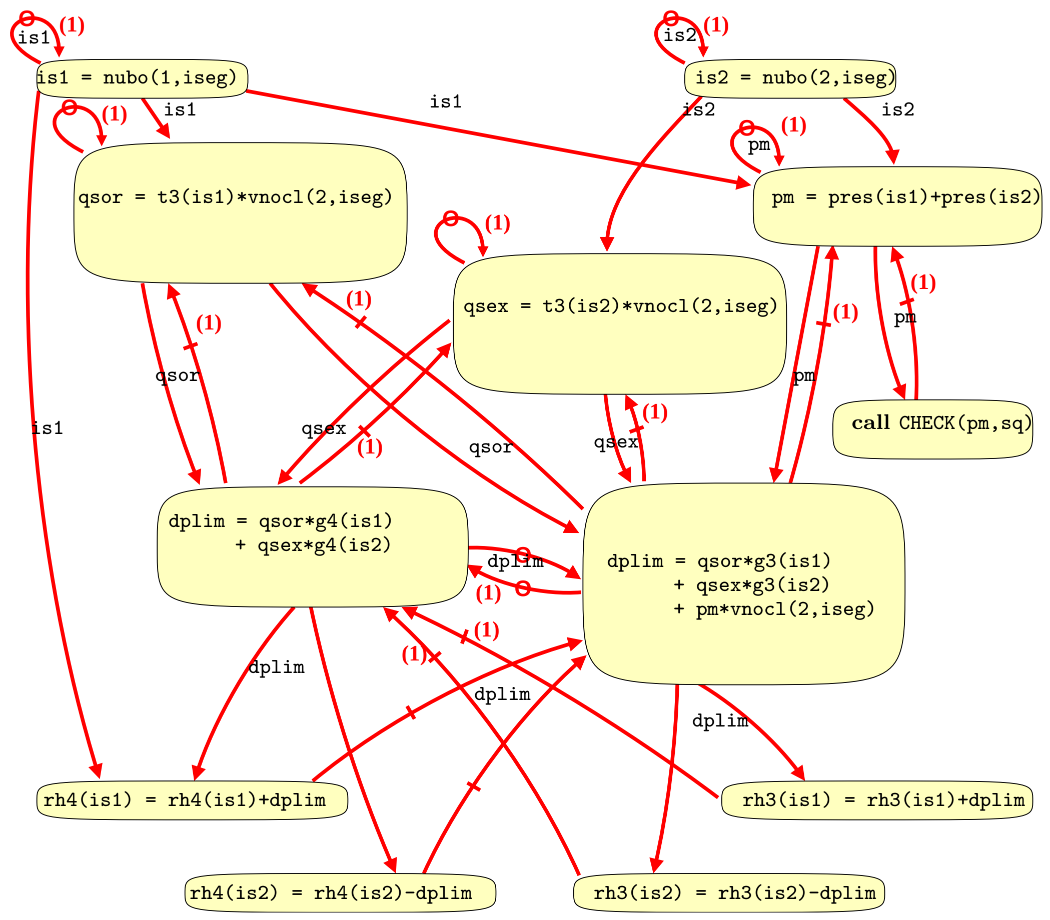


Figure 4.28: *Dependence Graph $\mathcal{G}$ of subroutine* FLW2D1COL

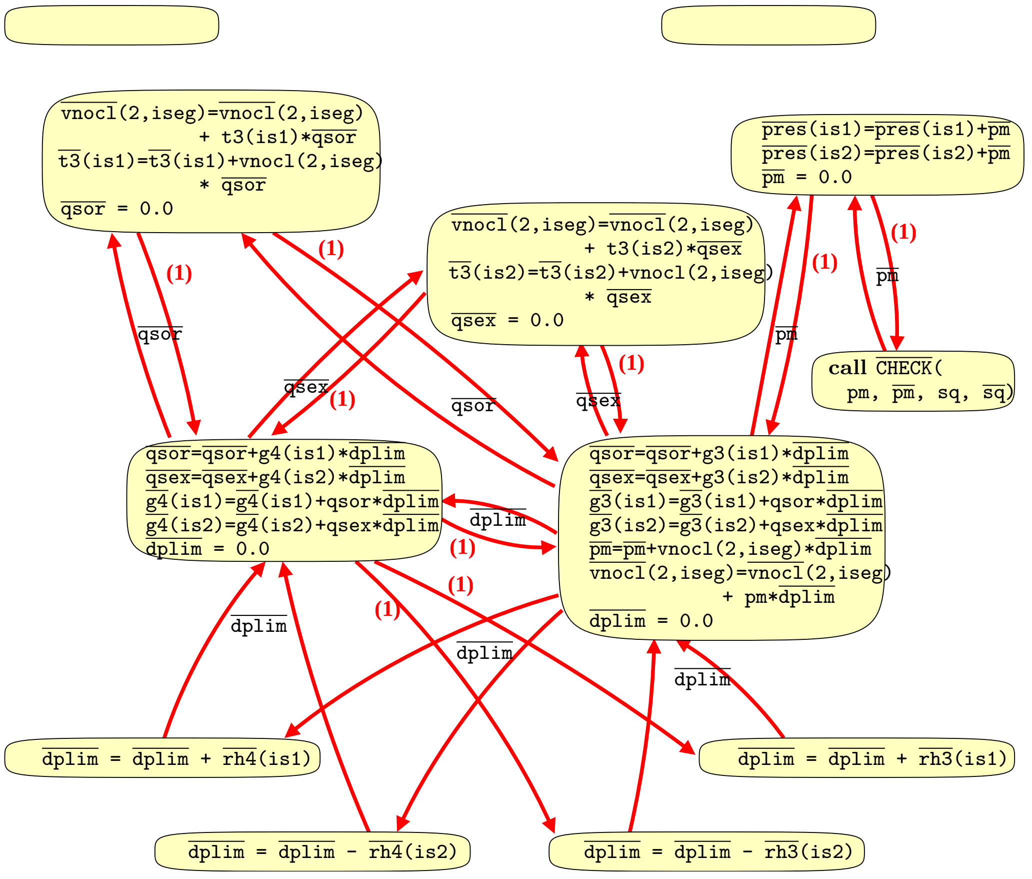


Figure 4.29: *Dependence Graph* $\overleftarrow{\mathcal{G}}$ *of the backward sweep of* $\overline{\texttt{FLW2D1COL}}$

Consequences for Parallel and Vectorial programs

A consequence of proposition (1) is that parallel properties of a loop can be transposed to its adjoint loop. If we take the strict definition that parallel loops are loops with no loop-carried data dependence, i.e. no data dependence inside the loop with a non-zero distance, then proposition (1) ensures that the adjoint loop has no loop-carried dependence either.

If we look at subroutine `FLW2D1COL` again, it contains a loop which is not strictly speaking parallel, because there are loop-carried dependences on figure 4.28 (dependances with distance vector **(1)**). However, the loop-carried dependences are *anti* and *output* dependences, known as "artificial", that can be removed by expansion or localization of the variables that cause them (*cf* section 3.4.2). If we localize `is1`, `is2`, `qsor`, `qsex`, `pm`, and `dplim`, then artifical data dependences disappear, and the loop is parallel. The adjoint loop is therefore parallel. Notice that this implies $\overline{\texttt{qsor}}$, $\overline{\texttt{qsex}}$, $\overline{\texttt{pm}}$, and $\overline{\texttt{dplim}}$ are localized too. Remember that the incrementation operations must be atomic. With these hypotheses, one can see that the adjoint subroutine $\overline{\texttt{FLW2D1COL}}$ is now composed of two successive parallel loops, on the same iteration domain. We can apply loop fusion (*cf* section 3.4.4), and we end up with the $\overline{\texttt{FLW2D1COL}}$ subroutine shown on figure 4.24. In other words, when a parallel loop is immediately followed by its adjoint loop, and this can be arranged for systematically, then the parallel loops and the adjoint differentiation operators commute, as summarized on figure 4.30. This transformation brings

Standard:

```
do // i= 1,N
         body(i)
end
do i= N,1
      $\overline{body(i)}$
end
```

$\Longleftrightarrow$

Improved:

```
do i= 1,N
      body(i)
      $\overline{body(i)}$
end
```

Figure 4.30: *Equivalent transformation of a reverse parallel Loop*

a huge benefit, even on sequential programs. The reason is that, for each iteration of the loop, the `PUSH` are immediately followed by their `POP` inside the same loop iteration. Therefore the stack size does not grow as much as before loop fusion.

Notice furthermore that there are no PUSH/POP calls left in subroutine $\overline{\texttt{FLW2D1COL}}$ after ajoint dead code elimination. Therefore the loop is definitely parallel. Things are not so simple if some PUSH/POP calls remain, because these share a stack. Parallelization requires localizing the stack too. Equivalently, one could get rid of PUSH/POP calls in the loop using a bunch of local temporary variables.

Proposition (1) also helps preserve *vectorial* properties. Consider a vectorial instruction, such as

```
A(1:n) := B(0:n-1)*B(2:n+1) + c
```

There are no loop-carried dependences in the loop implicitly represented here, and therefore no loop-carried dependences in the adjoint loop. As far as dependences are concerned, the adjoint is itself vectorial. Notice that the read of c is spread among the "iterations", and this results in a SUM reduction in the adjoint program:

```
B̄(0:n-1) := B̄(0:n-1) + B(2:n+1)*Ā(1:n)
B̄(2:n+1) := B̄(2:n+1) + B(0:n-1)*Ā(1:n)
c̄ := c̄ + SUM(Ā(1:n))
Ā(1:n) := 0
```

On the other hand, the adjoint of instruction:

```
A(1:n) := A(0:n-1)*A(2:n+1)
```

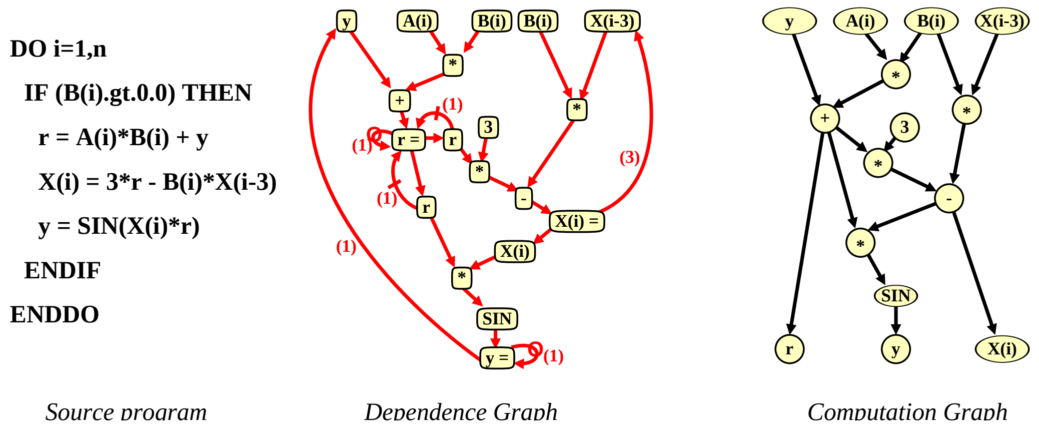
would not be immediately vectorial, because of the loop-carried data-dependence in the implicit loop, from the reads to the writes of A.

Application of proposition (1) to parallel programs that use message-passing appears promising too. Experiments are reported in [17]. One unsolved question is how to inverse the control flow and the data flow when the communications are not synchronous. A corollary of proposition (1) shows that the adjoint of a SEND instruction is a RECEIVE, and vice-versa. If the receiver of a value has no means to say where this value comes from, and at which instant, it might be difficult to send the adjoint value backwards in the adjoint program. This is still an open problem, and this proposition is a first step at formalizing it.

4.5.3 Computation Graph and Dependence Graph

We hope we have established the fact that the Dependence Graph is essential for AD, both for theoretical studies and for AD tools algorithms. There is another similar graph, though, which also plays an important role in AD: the *computation graph* [24, Chapter 8]. It may be useful to compare these two graphs and discuss their differences.

A Computation Graph represents the chain of elementary computations that lead from the inputs to the results. See figure 4.31 for an illustration example. For a simple expression, the computation graph is roughly the syntax



```
DO i=1,n
  IF (B(i).gt.0.0) THEN
    r = A(i)*B(i) + y
    X(i) = 3*r - B(i)*X(i-3)
    y = SIN(X(i)*r)
  ENDIF
ENDDO
```

Figure 4.31: *Comparison of Computation and Dependence Graphs*

tree of the expression. For a sequence of instructions, the computation graph captures the fact that an intermediate result, e.g. r, may be reused in several following instructions, by setting arrows from the expression whose result is r to each read of the variable that holds r. Therefore the computation graph is no longer a tree but a Directed Acyclic Graph, but in a sense it still views the sequence of instructions as one expression that allows intermediate results to be used more than once. The principal characteristic is that variables are abstracted away: except for top and bottom rows, nodes represent values and not variables. The top row shows variables that hold the input values, the bottom row shows variables that receive the result values. To go back from the computation graph to a sequence of executable instructions requires to re-introduce variables, very much like the *register allocation* problem in compiler theory. Computation graphs are not defined for programs with control, only for sequences of instructions. For example on figure 4.31, the computation graph only represents the "then" part of the loop body, whereas the dependence graph has cycling arrows that capture loop-carried dependences. There are some attempts to extend computation graphs in this direction, but the lack of explicit representation of variables makes it difficult to represent data dependences. Now we can stress the two main differences that we feel between the Computation Graph and the Dependence Graph:

Computation Graph is more mathematical, Dependence Graph is more operational: For a basic block of instructions, the Computation Graph looks like the Dependence Graph, except that variables are not represented any more. Consequently the Computation Graph does not represent the execution order, which is strongly related to the true, anti, and output dependences in the Dependence Graph. There is no anti nor output dependences in a Computation Graph, and each node is in fact a single-assignment variable, i.e. a mathematical value.

Computation Graph is more static, Dependence Graph is more dynamic: The Computation Graph is not well defined for loops and arbitrary control, whereas dealing with loops is essential for Dependence Graphs in order to parallelize them. To our knowledge, the notion of *distance* in the iteration space is not defined between the two ends of a Computation Graph arrow.

When AD is concerned, another sort of computation graph is often used. Unfortunately, it is often also called "Computation Graph", which is a pity. Let us call it *Jacobian Computation Graph.* It represents the chain of elementary operations that compute the partial derivatives of all results with respect to all inputs. It shares similarity with ordinary computation graphs, but it is specialized for Jacobian coefficients: each edge is labelled with the partial derivative of the downwards node with respect to the upwards node, and ordinary nodes have no label any more. The Jacobian Computation Graph of the inside instructions of figure 4.31 is shown on the left of figure 4.32. The Jacobian Computation Graph is a framework to compute the Jacobian of the results with respect to the inputs. The chain rule tells us that the partial derivative of one result with respect to one input is the sum, for all paths linking input and result, of the product of all the partial derivatives which label the edges along the path. Computing the Jacobian amounts to building an equivalent bipartite graph between inputs and results, by progressively eliminating the intermediate nodes and recomputing the labels on the new edges by multiplication of the labels on the previous edges. The right of figure 4.32 shows the equivalent final bipartite graph. The rule of the game is to find the best elimination order of intermediate nodes, which reaches the bipartite graph with a minimal number of multiplications. Actually, node elimination is just a special case of the so-called *edge elimination* and *face elimination* [40], and only the latter is able to represent the optimal

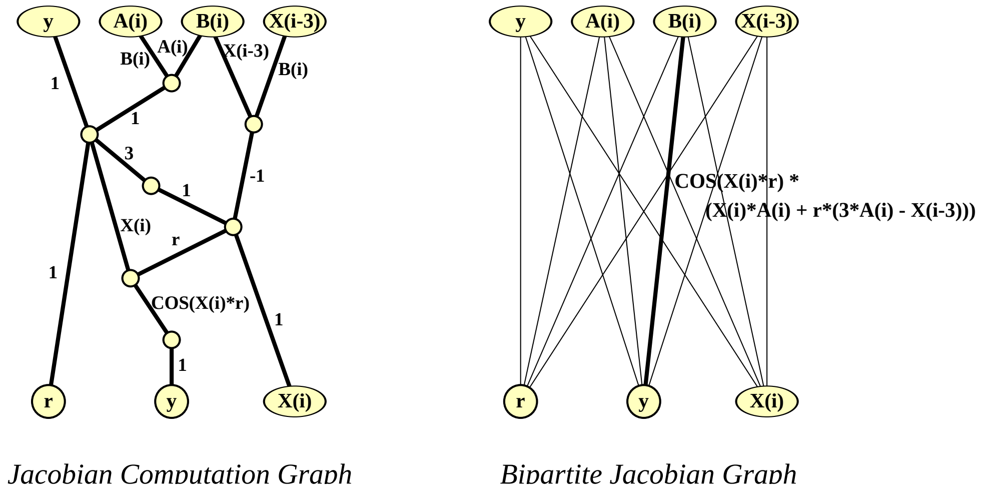


Figure 4.32: *Jacobian Computation Graph of the program from figure 4.31*

elimination sequence. But finding this optimal is conjectured to be NP-hard, and researchers explore a number of heuristics to get as close as possible to this optimum.

The Jacobian Computation Graph is even more different from the Dependence Graph. It is not directed any more. It represents arithmetic operations that are only sums and products. It is used only to find the best elimination order, i.e. the sequence of operations that compute the Jacobian with a minimal cost. It is not used to change the order between the original instructions. To our knowledge, there is still no Jacobian Computation Graph defined for entiere programs with control.

4.6 Application: The TAPENADE AD Tool

This section describes the Automatic Differentiation tool TAPENADE [34], which is developped by our research team. To this date, the people who contributed most to this very large implementation task are Valérie Pascual, Rose-Marie Greborio, Frédéric Olier, Jean-Philippe Sautarel, Marc-Aurèle Ngoxuan, Jean-Charles Rihaoui, Benjamin Dauvergne, Christophe Massol, and myself. TAPENADE progressively implements the results of our research about models and static analyses for AD. Development of TAPENADE started

Automatic Differentiation Tool

Name: TAPENADE version 2.0

Date of birth: January 2002

Ancestors: Odyssée 1.7

Address: `www.inria.fr/tropics/tapenade.html`

Specialties: AD Reverse, Tangent, Vector Tangent, Restructuration
Reverse mode Strategy: Store-All, Checkpointing on calls
Applicable on: FORTRAN95, FORTRAN77, and older
Implementation Languages: 90% JAVA, 10% C
Availablility: Java classes for Linux and Solaris, or Web server

Internal features: Type Checking, Read-Written Analysis, Forward and Backward Activity, Dead Adjoint Code, TBR

Figure 4.33: TAPENADE*'s ID card*

in 1999. It is the successor of ODYSSÉE, a former AD tool developped by INRIA and université de Nice from 1992 to 1998. TAPENADE and ODYSSÉE share some fundamental concepts. However, the whole internal program representation was redesigned, as well as static analyses to improve differentiation time and result. In particular, TAPENADE and ODYSSÉE share no part of their source code, and even the implementation language changed from CAML to JAVA. TAPENADE is distributed by INRIA. At present, we are aware of regular industrial use of TAPENADE at Dassault Aviation, CEA Cadarache, Rolls-Royce (UK), BAe (UK), Alenia (Italy). Academic colleagues use it on a regular basis at INRA (French agronomy research center), Oxford (UK), Queen's University Belfast (UK), Argonne National Lab (USA). TAPENADE is still under permanent development, and figure 4.33 gives a synthetic view of its present state.

Section 4.6.1 summarizes the objectives of TAPENADE, and presents the architecture that we devised to meet these objectives. Section 4.6.2 presents the underlying Automatic Differentiation models, that are helpful to really understand why TAPENADE created the program it created. The performances of the tool are shown in section 4.6.3, for differentiation time as well as for the quality of the differentiated programs. Section 4.6.4 gives a feeling of the user interface, and section 4.6.5 conclude with developments to come.

4.6.1 Objectives and Architecture

TAPENADE has the following two principal objectives:

- To serve as an implementation platform to experiment with new modes of AD or to validate new AD algorithms and analyses. Ideally, this experimantation could also take place outside of our research team.

- To provide external end-users, academic or industrial, with state-of-the-art AD of real programs, with no restriction on the style or on the size of the application to differentiate.

From the PARTITA experience (section 3.4), we gathered some guidelines:

- Tools that work on programs must have an internal representation which is convenient for analysis more than for mere edition. Therefore, instead of syntax trees, we prefer Call Graphs of Flow Graphs, as described in section 2.2.

- The internal representation must concentrate on the semantics of the program and eliminate or normalize what is less essential. In particular, the internal representation should be independent of the particular programming language used by the program.

- The internal representation must facilitate data flow analysis. Internal representation of variables is essential, and must be chosen very carefully. In particular, it must once and for all solve the tedious array region management due to constructs such as the FORTRAN `COMMON` and `EQUIVALENCE`.

- Many program analyses are indeed independent of Automatic Differentiation, and are identical to those needed by parallelizers. These analyses must form a fundamental layer, clearly separated from the AD specific part which is built above it.

- It is not necessary that a tool be restricted to mini-languages. Experience shows that real programs use all possibilities of their implementation language. If we want a usable tool, we must take care of every construct or possibility offered by the langage. Indeed, this is far less boring than one could fear!

Figure 4.34 summarizes the architecture of TAPENADE, built after these guidelines. To enforce a clear separation from the language of the programs,

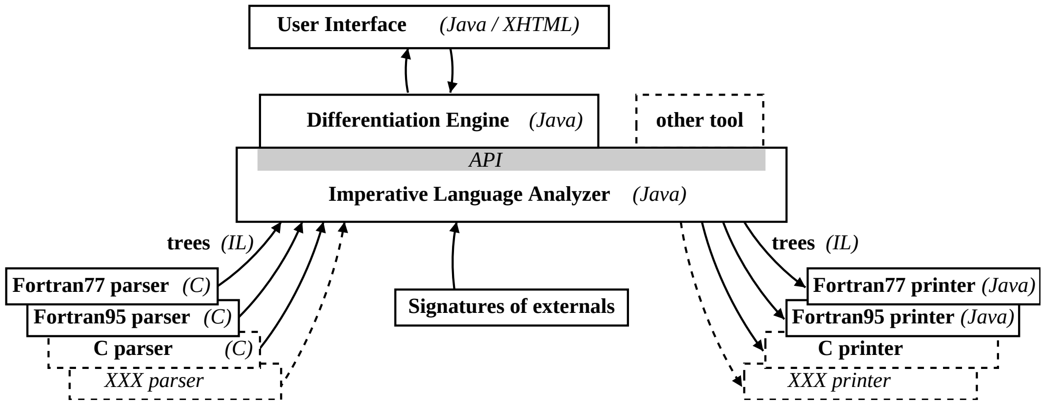


Figure 4.34: *Overall Architecture of* TAPENADE

we devised an Intermediate Language (IL), which should eventually contain all the syntactic constructs of imperative languages. IL has no concrete syntax, i.e. no textual form. It is an abstract syntax, that strives to represent by the same construct equivalent concrete constructs from different languages. IL currently covers all FORTRAN77, FORTRAN95, and C. Object-oriented constructs are progressively being added. The front-end for a given language, and the corresponding back-end, are put outside of TAPENADE, and communicate with it by a simple tree transfer protocol.

Real programs often use external libraries, or more generally "black-box" routines. The architecture must cope for that, because static analyses greatly benefit from compact, summarized information on these black-box routines. The end-user can provide TAPENADE with these signatures of black-box routines, in a separate file whose syntax is described in section 4.6.4.

The Imperative Language Analyzer performs general purpose analyses, independent from AD. Figure 4.35 shows these analyses with their relative dependencies. All of them run on Flow Graphs, according to their formal description as data flow equations (*cf* section 4.4). The result of these general analyses are of course used later by the AD level (*cf* figure 4.36), which performs Differentiable Dependency Analysis followed by Activity Analysis (section 4.4.1) and, specifically for the reverse mode, TBR Analysis (section 4.4.2) and Dead Adjoint Code detection (section 4.4.4). At the time when we write this report, the Pointer analysis as well as the Dead Adjoint Code detection are only prototypes, not functional in the distributed tool.

TAPENADE consists in a set of JAVA classes, and can be called directly as a command with arguments, or through a XHTML graphical user interface. Consequently, it was relatively easy to build a web server that runs TAPENADE on any file uploaded by a distant user.

4.6.2 Differentiation Model

In this section, we plan to describe precisely the actual differentiation model of TAPENADE. The goal is to gain a deeper understanding and familiarity with programs produced by TAPENADE. For this reason, contrary to the previous program "listings", we shall use the actual FORTRAN syntax produced by TAPENADE in the following pieces of code. In particular, symbol names will not be $\dot{\mathtt{x}}$ or $\overline{\mathtt{x}}$ any more.

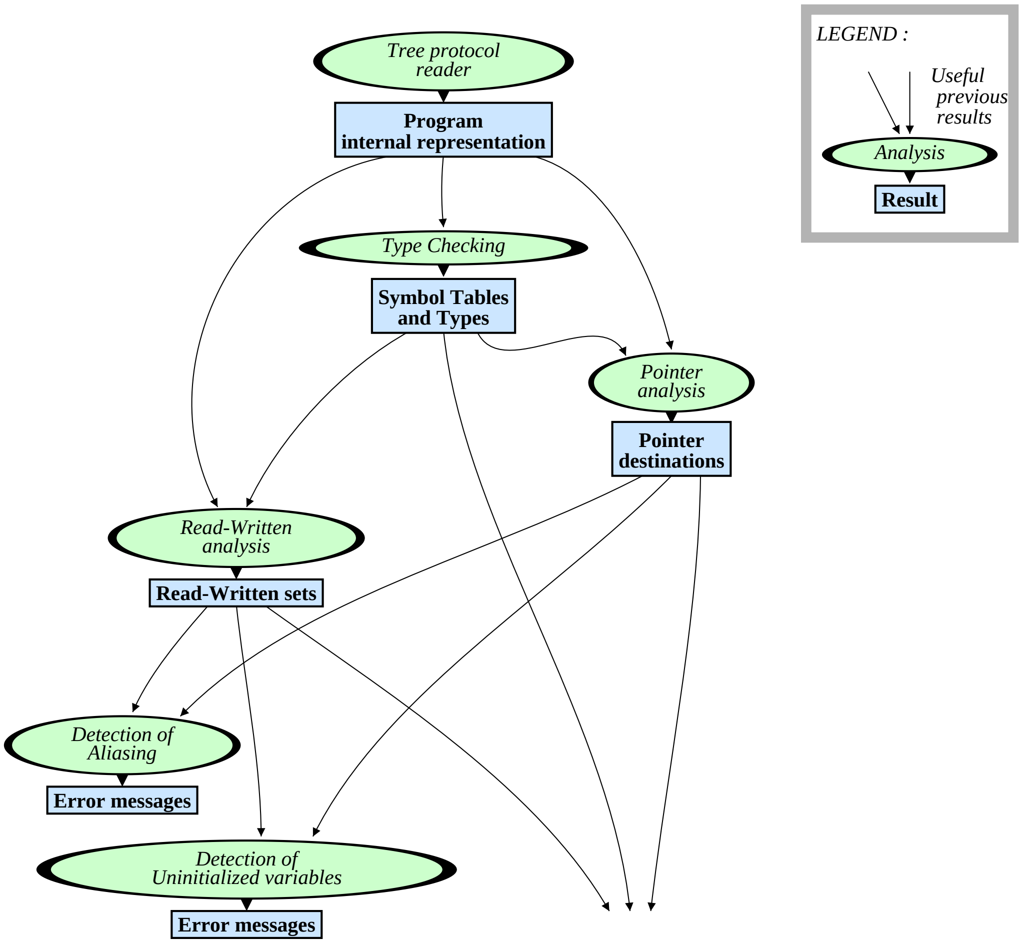


Figure 4.35: *Ordering of generic static data flow analyses in* TAPENADE

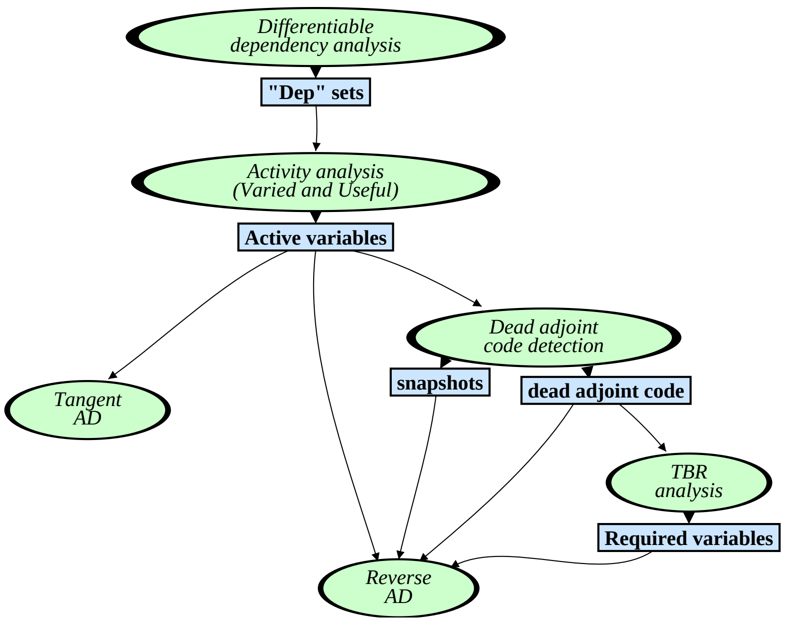


Figure 4.36: *Ordering of AD static data flow analyses in* TAPENADE

Symbol names

First consider symbol names. If a variable `v` is of differentiable type, and is currently active (see *activity* 4.4.1), this derivative is stored in a new variable of same type that TAPENADE names after `v` as follows: `vd` ("`v` *dot*") in *tangent* mode, and `vb` ("`v` *bar*") in reverse mode. Derivative names for procedures and `COMMONS` are built appending "`_D`" in *tangent* mode and "`_B`" in reverse mode. Figure 4.37 summarizes that. TAPENADE checks for possible conflicts

original program	TAPENADE **tangent**	TAPENADE **reverse**
`SUBROUTINE T1(a)`	`SUBROUTINE T1_D(a,ad)`	`SUBROUTINE T1_B(a,ab)`
`REAL a(10)`	`REAL a(10),ad(10)`	`REAL a(10),ab(10)`
`COMPLEX b(5)`	`COMPLEX b(5),bd(5)`	`COMPLEX b(5),bb(5)`
`REAL*8 c,d`	`REAL*8 c,cd,d,dd`	`REAL*8 c,cb,d,db`
`COMMON /cc/ b,c`	`COMMON /cc_d/ bd,cd`	`COMMON /cc_b/ bb,cb`
	`COMMON /cc/ b,c`	`COMMON /cc/ b,c`

Figure 4.37: *Differentiation of symbol names in* TAPENADE

with names already used in the program, in which case it appends `0`, then `1`, etc after the derivative name until conflicts disappear. Suffixes can be changed via command line options.

Simple instructions

Now consider an assignment I_k. In *tangent* mode (equation (4.4)), derivative instruction $\dot{I}_k$ implements $\dot{X}_k = f'_k(X_{k-1}).\dot{X}_{k-1}$, with initial $\dot{X}_0 = \dot{X}$. In *reverse* mode (equation (4.5)), derivative instruction(s) $\overline{I}_k$ implement $\overline{Y}_{k-1} = f'^{*}_k(X_{k-1}).\overline{Y}_k$, with initial $\overline{Y}_p = \overline{Y}$. Just like the original program *overwrites* variables, the differentiated program overwrites the differentiated variables, writing values $\dot{X}_k$ over previous values $\dot{X}_{k-1}$ in tangent mode, or writing values $\overline{Y}_{k-1}$ over previous values $\overline{Y}_k$ in the reverse mode. For example, if I_k is `a(i)=x*b(j) + COS(a(i))`,

$$\dot{I}_k \quad \text{implements} \quad \begin{pmatrix} \dot{\texttt{a}}\texttt{(i)} \\ \dot{\texttt{b}}\texttt{(j)} \\ \dot{\texttt{x}} \end{pmatrix} = \begin{pmatrix} \texttt{-SIN(a(i))} & \texttt{x} & \texttt{b(j)} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \dot{\texttt{a}}\texttt{(i)} \\ \dot{\texttt{b}}\texttt{(j)} \\ \dot{\texttt{x}} \end{pmatrix},$$

$$\overline{I}_k \quad \text{implements} \quad \begin{pmatrix} \overline{\texttt{a}}\texttt{(i)} \\ \overline{\texttt{b}}\texttt{(j)} \\ \overline{\texttt{x}} \end{pmatrix} = \begin{pmatrix} \texttt{-SIN(a(i))} & 0 & 0 \\ \texttt{x} & 1 & 0 \\ \texttt{b(j)} & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \overline{\texttt{a}}\texttt{(i)} \\ \overline{\texttt{b}}\texttt{(j)} \\ \overline{\texttt{x}} \end{pmatrix},$$

and therefore TAPENADE produces the derivative instructions shown on figure 4.38. Other simple instructions may have side-effects that affect deriva-

TAPENADE **tangent**	TAPENADE **reverse**
`ad(i) = xd*b(j)`	`xb = xb + b(j)*ab(i)`
`        + x*bd(j)`	`bb(j) = bb(j) + x*ab(i)`
`        - ad(i)*SIN(a(i))`	`ab(i) = -SIN(a(i))*ab(i)`

Figure 4.38: *Differentiation of a single assignment in* TAPENADE

tives. For example a `READ` I-O into a variable `v` forces the derivative of `v` to be reset to zero. TAPENADE automatically inserts these reset instructions. However, the end-user should check that this is the behavior wanted.

Activity of variables

TAPENADE lets the end-user specify that only some output variables (the "*dependent*") must be differentiated with respect to only some input variables (the "*independent*"). We say that variable `y` *depends on* `x` when the derivative of `y` with respect to `x` is not trivially null. A variable is said "*active*" if it depends on some independent *and* some dependent depends on it. Only the derivatives of the active variables need be computed. If variable `v` depends on no independent, then `vd` is certainly null and the value of `vb` does not matter. Conversely, if no dependent depends on `v`, then the value of `vd` does not matter, and `vb` is certainly null. TAPENADE automatically detects active variables and simplifies the differentiated program accordingly. In the

original program	TAPENADE **tangent**	TAPENADE **reverse**

`x = 1.0`
`z = x*y`
`t = y**2`
`IF (t .GT. 100) ...`
	`x = 1.0`
	`zd = x*yd`
	`z = x*y`
	`t = y**2`
	`IF (t .GT. 100) ...`
		`x = 1.0`
		`z = x*y`
		`t = y**2`
		`IF (t .GT. 100) ...`
		`...`
		`yb = yb + x*zb`

Figure 4.39: *Utilization of activity in* TAPENADE

example of figure 4.39, `x` does not depend any more on the independent, and `t` has no influence on any dependent. Therefore, TAPENADE knows that `xd` and `tb` are null: they can be simplified and never computed. We shall say that these derivatives are *implicit-null*. Symmetrically, `td` and `xb` are non-null but useless, and therefore need not be evaluated. Nevertheless, there are two special cases where TAPENADE explicitly resets implicit-null variables: (1) when the control flow merges and the other incoming flow has an explicit non-null derivative for this variable, and (2) when the end of the differentiated program is reached and the derivative is an output. Notice also that some of the user-given independent and dependent variables may turn out to be inactive. If so, TAPENADE removes them automatically. This can be checked on the differentiation comments that are put at the beginning of the differentiated program.

Control structure

Figure 4.40 illustrates how TAPENADE builds the control structure of the differentiated procedures. In *tangent* mode, equation (4.4) allows derivative instructions $\dot{I}_k$ to run along with the original I_k, indeed *just before* I_k because I_k may overwrite a part of X_{k-1} that is used by $f'_k(X_{k-1})$ in $\dot{I}_k$. The control structures are unchanged. In *reverse* mode, TAPENADE applies the *Store All* strategy (*cf* section 4.1.3), resulting in a *forward sweep* followed by a *backward sweep*. The forward sweep runs the original procedure, storing into a stack the variables potentially required by the derivatives. In addition, the forward sweep stores into the same stack the *control* information, used by the backward sweep to reproduce between the $\overline{I}_k$ the reverse of the original control flow. The stack is used classically through several `PUSH` and `POP` subroutines, which are adapted to the type of the value. Its internal representation of programs as Flow Graphs allows TAPENADE to use structured programming in the backward sweep like in the forward sweep, using very little memory space to store the control, and with no restriction on the original control (`GOTO`'s, alternate procedures or I-O returns,...). The principle is: the right time to store the control is when the original control flow *merges*, and what must be stored then is *where* the control actually *came from*.

Procedure calls

TAPENADE treats procedure calls differently from simple instructions, because a procedure call indeed represents a bunch of instructions, possibly with control. Therefore the differentiated instructions cannot be put *before* the original call, but rather *inside*, yielding a differentiated procedure, with additional arguments for the derivatives. The example on figure 4.41 illustrates this. In *tangent mode*, a call to `SUB` just gives a call to the differentiated `SUB_D`. In *reverse mode*, TAPENADE *checkpoints* the procedure call: the forward sweep calls the original `SUB` and the backward sweep calls the differentiated `SUB_B`, that gathers its own forward and backward sweeps. One principle of TAPENADE is procedure *generalization*, as opposed to *specialization*. Even if a procedure is called many times, with arguments sometimes active, sometimes not, only one differentiated procedure is built, i.e. for the most general activity of arguments. Thus, specific calls are sometimes given dummy derivatives, either to feed them with a null derivative input, or to receive a useless derivative result. Suppose `SUB` is called elsewhere with an

original program

```
SUBROUTINE S1(a, n, x)
...
DO i=2,n,7
  IF (a(i).GT.1.0) THEN
    a(i) = LOG(a(i)) + a(i-1)
    IF (a(i).LT.0.0) a(i)=2*a(i)
  END IF
ENDDO
END
```

TAPENADE **reverse:**
forward sweep

```
DO i=2,n,7
  IF (a(i).GT.1.0) THEN
    CALL PUSHREAL4(a(i))
    a(i) = LOG(a(i)) + a(i-1)
    IF (a(i).LT.0.0) THEN
      CALL PUSHREAL4(a(i))
      a(i) = 2*a(i)
      CALL PUSHINTEGER4(3)
    ELSE
      CALL PUSHINTEGER4(2)
    END IF
  ELSE
    CALL PUSHINTEGER4(1)
  END IF
ENDDO
CALL PUSHINTEGER4(i - 7)
```

TAPENADE **tangent**

```
SUBROUTINE S1_D(a, ad, n, x)
...
DO i=2,n,7
  IF (a(i).GT.1.0) THEN
    ad(i) = ad(i)/a(i) + ad(i-1)
    a(i) = LOG(a(i)) + a(i-1)
    IF (a(i).LT.0.0) THEN
      ad(i) = 2*ad(i)
      a(i) = 2*a(i)
    END IF
  END IF
ENDDO
END
```

TAPENADE **reverse:**
backward sweep

```
CALL POPINTEGER4(adTo)
DO i=adTo,2,-7
  CALL POPINTEGER4(branch)
  IF (branch .GE. 2) THEN
    IF (branch .GE. 3) THEN
      CALL POPREAL4(a(i))
      ab(i) = 2*ab(i)
    END IF
    CALL POPREAL4(a(i))
    ab(i-1) = ab(i-1) + ab(i)
    ab(i) = ab(i)/a(i)
  END IF
ENDDO
```

Figure 4.40: *Differentiation of control flow in* TAPENADE

original program	TAPENADE **reverse: forward sweep**

`x = x**3`
`CALL SUB(a, x, 1.5, z)`
`x = x*y`
	`CALL PUSHREAL4(x)`
	`x = x**3`
	`CALL PUSHREAL4(x)`
	`CALL SUB(a, x, 1.5, z)`
	`x = x*y`

TAPENADE **tangent**	TAPENADE **reverse: backward sweep**

`xd = 3*x**2*xd`
`x = x**3`
`CALL SUB_D(a, ad, x, xd,`
`          1.5, 0.0, z)`
`xd = y*xd`
`x = x*y`
	`xb = y*xb`
	`CALL POPREAL4(x)`
	`CALL SUB_B(a, ab, x, xb,`
	`          1.5, arg2b, z)`
	`CALL POPREAL4(x)`
	`xb = 3*x**2*xb`

Figure 4.41: *Differentiation of a procedure call in* TAPENADE

active 3rd argument, whereas the 4th argument is never active. This explains the "`0.0`" argument in tangent, and the "`arg2b`" in reverse.

In the reverse mode, checkpointing requires taking a *snapshot*. TAPENADE runs a preliminary Read-Written analysis to find a minimal snapshot, made of variables that are both *used* by the procedure and *overwritten* before the differentiated procedure is called. On the example of figure 4.41, the Read-Written analysis could prove that this is only the case for `x`.

Since there are no pointers in FORTRAN77, it is common practice to pass an individual element of an array to a subroutine, and then use it in the subroutine as a new array. In other words, the array element is considered as a pointer to the sub-array starting from this element. This is probably on the borderline with respect to the standard, but happens so often that TAPENADE has a specific behavior for this. Of course a message is printed, but moreover the type of the formal argument (an array), is "told" to the actual argument (an array element). This results in the entire sub-array being saved by the snapshot. There is an example of this behavior on figure 4.44, where not only `t(k)` is stored, but rather the seven array elements from `t(k)` to `t(k+7)`.

Sometimes a variable must be stored at part of the snapshot, and also because it is needed by derivatives of previous instructions. Instead of `PUSH`'ing the value twice, TAPENADE uses a special `LOOK` function, that reads the value from the stack without removing it, so that it can be `POP`'ed later.

TBR Analysis and Adjoint Instructions Merging

The reverse mode uses two specific improvements built in TAPENADE. We saw that intermediate values need to be stored before overwritten, *only* when they will be used by the differentiated instructions. A specific program static analysis does this in TAPENADE. On the example of figure 4.42, TAPENADE could prove that neither `x` nor `y` were needed by the differentiated instructions, and therefore did not store them on the stack. Also, many reverse differentiated instructions *increment* a differentiated variable. An internal *data-dependency* analysis allows TAPENADE to safely gather initializations and increments of the same differentiated variable, to make the code shorter. The result is closer to what one would write when programming an *adjoint* code by hand. This is the case for assignments to `ab` in figure 4.42.

original program	TAPENADE **reverse: naive bckward sweep**	TAPENADE **reverse: improved bckward sweep**

`x = x + EXP(a)`
`y = x + a**2`
`a = 3*z`
	`CALL POPREAL4(a)`
	`zb = zb + 3*ab`
	`ab = 0.0`
	`CALL POPREAL4(y)`
	`ab = ab + 2*a*yb`
	`xb = xb + yb`
	`yb = 0.0`
	`CALL POPREAL4(x)`
	`ab = ab + EXP(a)*xb`
		`CALL POPREAL4(a)`
		`zb = zb + 3*ab`
		`xb = xb + yb`
		`ab = 2*a*yb + EXP(a)*xb`
		`yb = 0.0`

Figure 4.42: *Improved reverse differentiation in* TAPENADE

The multi-directional AD model

The multi-directional AD mode, described in section 4.5.1, is similar to the tangent (*resp.* reverse) mode, except that derivatives are computed simultaneously for several vectors $\dot{X}$ (*resp.* $\overline{Y}$). We shall call these vectors the differentiation "directions". Multi-directional modes are sometimed called "*vector*" modes. Multi-directional modes therefore compute many derivatives, while running the original instructions only once.

The model, illustrated on figure 4.43, is therefore to put every derivative instruction into a loop. This loop has one iteration for each differentiation direction $\dot{X}$ or $\overline{Y}$. Data-Dependency analysis is used to fuse these loops whenenver possible, to reduce loop overhead. One consequence is that derivative variables become arrays, by adding an extra dimension which is the number of initial differentiation directions. This extra dimension should be deepest in the type structure, as can be seen for the `POINT` type, to let the program easily take one component of the structure along with all its derivatives. The multi-directional AD model makes the classical distinction between static size of arrays and the size actually used. Therefore, differentiated variables have a fixed static size in the "directions" dimension. This size should be given as a constant by the end-user in the `DIFFSIZES.inc` include file. However at run-time, the differentiated program may be given fewer differentiation directions. This run-time number of dimensions is an extra integer argument to the differentiated routine, a priori called `nbdirs`. Presently in TAPENADE, only the tangent mode can be done in multi-directional mode.

4.6.3 Performances and Limitations

TAPENADE has reached a state were it can be used routinely by academic and industrial users. We gave references to existing users at the beginning of this section. However there are still limitations, and probably there will always be.

Let's first discuss the speed of the differentiation itself. This is a secondary problem in theory because differentiation is done once, like compilation. Nevertheless in reality differentiation time matters when the application evolves rapidly. On a 1 GHz computer, differentiation takes about 5 seconds for 1000 lines of source, not counting comments. The most time consuming part of TAPENADE is the dependency analysis 4.4.1, which is quadratic with respect to the number of variables. On a rather extreme example, with about 2,000 variable names, this made differentiation time go up to 20 minutes.

The most crucial point is the speed of the differentiated program. An independent study [16] shows that TAPENADE creates the best tangent code, and is second to TAF only for the adjoint code. Even if we think that TAPENADE is indeed better than what this study shows, this is not the point here.

original program	TAPENADE **multi-directional tangent:**
`SUBROUTINE DECLS1(t1, n,`	`SUBROUTINE DECLS1_DV(t1, t1d, n, m,`
`              m, t4, t5)`	`            t4, t4d, t5, t5d, nbdirs)`
`TYPE POINT`	`INCLUDE 'DIFFSIZES.inc'`
`  REAL x`	`TYPE POINT_DV`
`  REAL y`	`  REAL x(NBDirsMax)`
`END TYPE`	`  REAL y(NBDirsMax)`
`PARAMETER (n1=500)`	`END TYPE`
`POINT t4(n1)`	`TYPE POINT`
`REAL*8 t1(n),a`	`  REAL x`
`REAL t5(2000)`	`  REAL y`
`...`	`END TYPE`
`t1(10) = a*t5(10)`	`PARAMETER (n1=500)`
`t1(5) = t4(5)%x + t4(5)%y`	`POINT_DV t4d(n1)`
`t4(2)%x = 3*t4(3)%y`	`POINT t4(n1)`
`CALL F(t4(1)%y)`	`REAL*8 t1(n), t1d(NBDirsMax, n)`
`...`	`REAL*8 a,ad(NBDirsMax)`
	`REAL t5(2000), t5d(NBDirsMax, 2000)`
	`...`
	`DO nd=1,nbdirs`
	`  t1d(nd, 10)=a*t5d(nd, 10)+ad(nd)*t5(10)`
	`  t1d(nd, 5) = t4d(5)%x(nd)+t4d(5)%y(nd)`
	`  t4d(2)%x(nd) = 3*t4d(3)%y(nd)`
	`ENDDO`
	`t1(10) = a*t5(10)`
	`t1(5) = t4(5)%x + t4(5)%y`
	`t4(2)%x = 3*t4(3)%y`
	`CALL F_DV(t4(1)%y, t4d(1)%y, nbdirs)`
	`...`

Figure 4.43: TAPENADE *multi-directional model*

Here are times taken from our biggest industrial example applications.

Code name (&type)	Original time:	Tangent time:	Adjoint time:	Adjoint stack size:
ALYA (CFD)	0.03 s	0.06 s	0.20 s	11.4 Mb
THYC (Thermodynamics)	2.63 s	5.57 s	11.94 s	37.9 Mb
LIDAR (Optics)	5.21 s	5.68 s	10.74 s	17.3 Mb
STICS (Agronomy)	0.16 s	0.35 s	42.64 s	478.5 Mb

On the average, tangent AD code is approximately 3 times slower than the original code, and adjoint AD is approximately 7 times slower. On the STICS code, the adjoint code is very slow due to useless storage of a very large number of variables. Nearly 96% of the time is spent in stack traffic (`PUSH` and `POP`). This can be solved by additional code analyses, described in sections 4.4.3 and 4.4.4, which are currently under development.

The current limitations that pose the biggest problems on real codes are

- Programs that store active variables into files or integer arrays are not differentiated correctly because TAPENADE does not keep the derivatives along with the values.
- There is no pointer analysis nor differentiation of dynamic memory allocation procedures. This should come with adaptation to C
- Programs that emulate pointers by passing an array element to a subroutine that expects a whole array may be differentiated wrong. Note that this does not conform with standards.
- Declarations are regenerated in a different order from the original file. In particular "include" files are expanded. This makes the resulting declarations somewhat hard to read.
- Sometimes TAPENADE needs to introduce new intermediate arrays, whose size is dynamic. This is forbidden in standard FORTRAN and therefore some help is required from the end-user.

4.6.4 A glimpse at the User Interface

TAPENADE can be installed on the local computer and run from the command line or from a `Makefile`, just like a compiler. Here is a typical call:

```
#> tapenade -reverse -head func -vars "x z" file1.f file2.f
```

Alternatively, the TAPENADE web server

`http://tapenade.inria.fr:8080/tapenade/index.jsp`

requires no installation and of course always runs the latest version. It can be triggered in a few clicks from most web browsers. All TAPENADE documentation, with tutorial and an ever-growing reference manual, is available at:

`http://www-sop.inria.fr/tropics/tapenade.html`

User input to TAPENADE consists in command-line options, directives in the original code, as well as configuration files. Consider for instance *black-box* procedures, i.e. procedures eventually called by the code to be differentiated, whose source is hidden (e.g. libraries). If nothing is known about a *black-box* procedure, the interprocedural analyses of TAPENADE will make conservative assumptions, and the code produced will be less efficient. TAPENADE lets the user specify, in a configuration file, summarized information about *black-box* procedures about parameters read and written and their relative derivatives. Here is an example. Consider a call to a black-box function `G`:

```
r = G(a,T(k),r2)
```

Suppose the end-user knows that `G` only uses its 1st and 2nd arguments and some global variable `x`, and overwrites its 1st and 3rd argument. Suppose also, to make things a little bit more complex, that `G`, although called with a scalar 2nd argument `T(k)`, actually expects a 2nd argument of dimension 7. This is a widespread practice in Fortran77, to emulate pointers. The end-user can specify this in the configuration file as follows. The specification file first enumerates the formal arguments expected by `G`, including globals, which we call the *shape* of `G`. Then, following this shape, the following information may be given: argument type, argument read or not, argument written or not, or dependency matrix, which is the sparsity pattern of the Jacobian matrix. All this is written in the following form:

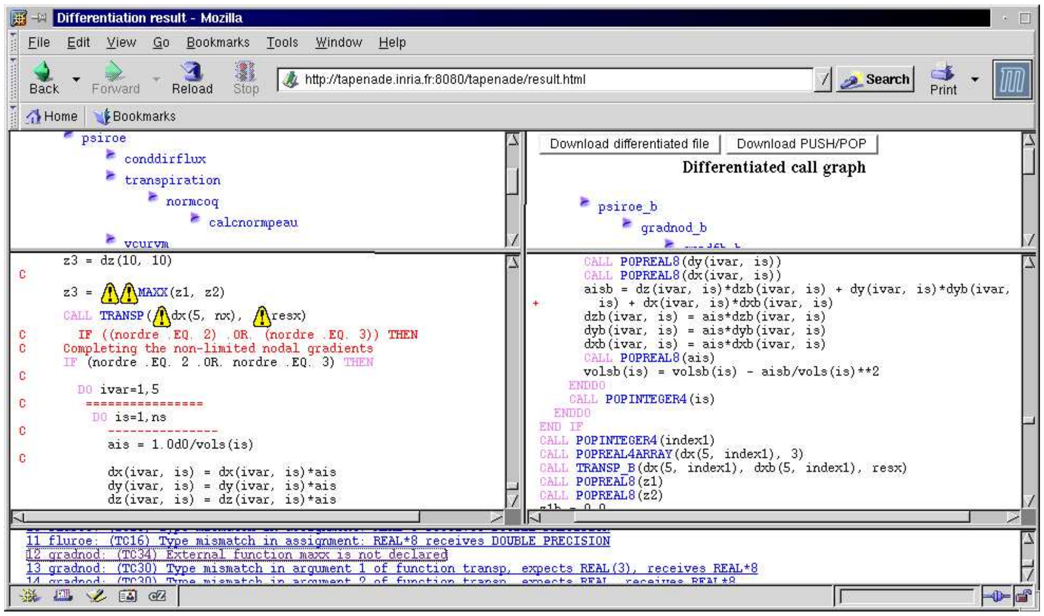
```
function G: external :
    shape:  (param 1, param 2, param 3, result, common /globals/x)
    type:  (real, real(1:7), real, real, real)
    R: (1 1 0 0 1)
    W: (1 0 1 1 0)
    deps:( 0 0 0 0 1
0 1 0 0 0
1 0 0 0 1
1 1 0 0 0
0 0 0 0 1)
```

Using this configuration file, the forward sweep takes a snapshot of the *In* arguments of `G`: `x`, `a`, and the 7 elements of array `T` from rank `k`. This requires that `k` be taken in the snapshot too. During the backward sweep, the snapshot is used to restore the necessary values, before the call to the differentiated routine `G_B`. This results in the code shown on figure 4.44.

original program	TAPENADE **tangent**	TAPENADE **reverse**
`...`	`...`	`...`
`a = a + 3*t(2)`	`a = a + 3*t(2)`	`CALL POPINTEGER4(k)`
`r = G(a,t(k),r2)`	`CALL PUSHREAL8(x)`	`CALL POPREAL8(a)`
`...`	`CALL PUSHREAL8ARRAY`	`CALL POPREAL8ARRAY`
	`        (t(k),7)`	`        (t(k),7)`
	`CALL PUSHREAL8(a)`	`CALL POPREAL8(x)`
	`CALL PUSHINTEGER4(k)`	`CALL G_B(a,ab,t(k),`
	`r = G(a,t(k),r2)`	`   tb(k),r2,r2b,rb)`
	`...`	`rb = 0.0`
		`r2b = 0.0`
		`tb(2) = tb(2) + 3*ab`
		`...`

Figure 4.44: *Improved* TAPENADE *output using Black-Box routine signature*

A graphical user interface, shown on figure 4.45, helps examine TAPENADE output, exhibiting correspondence between original and differentiated code. This user interface consists of HTML files, and is therefore accessible from the web server as well as from a local installation. In its bottom frame, the interface also lists error and warning messages found by TAPENADE, with location in the source. Source-code location is implemented with HTML links, which is very easy but slightly limited. Nevertheless, a click on a source line code generally makes the differentiated line appear in front, and vice-versa. A click of a node of a call graph makes the corresponding subroutine appear. A click on an error symbol makes the error text appear on the bottom frame, and conversely a click on an error text in the bottom frame scrolls the program to where the error occurs. One last word about error messages:


Differentiation result - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://tapenade.inria.fr:8080/tapenade/result.html
Search
Print
Home Bookmarks
Download differentiated file
Download PUSH/POP

```
psiroe
    conddirflux
    transpiration
        normcoq
            calcnormpeau
    vcurvm
```

```
      z3 = dz(10, 10)
C
      z3 = MAXX(z1, z2)
      CALL TRANSP(dx(5, nx), resx)
C       IF ((nordre .EQ. 2) .OR. (nordre .EQ. 3)) THEN
C     Completing the non-limited nodal gradients
      IF (nordre .EQ. 2 .OR. nordre .EQ. 3) THEN
C
        DO ivar=1,5
C        ================
          DO is=1,ns
C          ---------------
            ais = 1.0d0/vols(is)
C
            dx(ivar, is) = dx(ivar, is)*ais
            dy(ivar, is) = dy(ivar, is)*ais
            dz(ivar, is) = dz(ivar, is)*ais
```

Differentiated call graph

```
psiroe_b
    gradnod_b
```

```
          CALL POPREAL8(dy(ivar, is))
          CALL POPREAL8(dx(ivar, is))
          aisb = dz(ivar, is)*dzb(ivar, is) + dy(ivar, is)*dyb(ivar,
+           is) + dx(ivar, is)*dxb(ivar, is)
          dzb(ivar, is) = ais*dzb(ivar, is)
          dyb(ivar, is) = ais*dyb(ivar, is)
          dxb(ivar, is) = ais*dxb(ivar, is)
          CALL POPREAL8(ais)
          volsb(is) = volsb(is) - aisb/vols(is)**2
        ENDDO
        CALL POPINTEGER4(is)
      ENDDO
    END IF
    CALL POPINTEGER4(index1)
    CALL POPREAL4ARRAY(dx(5, index1), 3)
    CALL TRANSP_B(dx(5, index1), dxb(5, index1), resx)
    CALL POPREAL8(z1)
    CALL POPREAL8(z2)
```

```
11 fluroe: (TC16) Type mismatch in assignment: REAL*8 receives DOUBLE PRECISION
12 gradnod: (TC34) External function maxx is not declared
13 gradnod: (TC30) Type mismatch in argument 1 of function transp, expects REAL(3), receives REAL*8
14 gradnod: (TC30) Type mismatch in argument 2 of function transp, expects REAL, receives REAL*8
```

Figure 4.45: HTML *interface for* TAPENADE *output*

there are many types of messages, such as type conflicts, wrong number of arguments or dimensions, aliasing or variables used before initialized. Although the temptation is strong, these messages should not be ignored right away. Especially when AD is concerned, these messages may indicate that the program runs into one limitation of the AD technology. Generally speaking, compilers often permit to go against the standard with no visible harm. However, this often introduces errors into the program differentiated in reverse mode. This is particularly true for two such messages:

- **Aliasing:** one can get the following message
  ```
  (DF02) Potential aliasing in calling function F,
         between arguments ...
  ```
  Suppose a subroutine `F` has many arguments (i.e. formal parameters or commons or globals...). *Aliasing* happens at a given call of `F` when two different arguments receive the same actual argument, or at least two actual arguments that overlap in memory, *and* at least one of these two arguments is overwritten by `F`. Notice that such aliasing results in code that does not conform to the Fortran standard [38, Section 5.7.2].

Tapenade, like most static analysis tools, analyses each subroutine in its own local context, i.e. assumes that each formal parameter is a different memory location. This assumption leads to a differentiated program that will fail if there is aliasing. For example:

```
SUBROUTINE F(a, b)
  ...
  a = 3*b
```

would be differentiated in the reverse mode as:

```
SUBROUTINE F_B(a, ab, b, bb)
  ...
  a = 3*b
  ...
  bb = bb + 3*ab
  ab = 0.0
```

This works fine as long as the actual `a` and `b` are different variables. But if `a` = `b` = `x`, `F` does `x = 3*x`, and what you want for `F_B` is `xb = 3*xb`. This is not what the above `F_B` will do. Similar problems arise in most program transformations, such as parallelization, partial evaluation, etc. TAPENADE is able to detect this situation. Notice however that this is an undecidable analysis, and therefore TAPENADE may detect a case of aliasing when there is none in reality. However, when the end-user confirms that the aliasing is true, this must be fixed. Generally, this is done by introducing a temporary variable, so that memory locations of the parameters do not overlap.

- **Casting:** one can get the following messages

```
 (AD01) Actual argument ...  of F is active
        while formal argument is non-differentiable
 (AD02) Actual output ...  of F is useful
        while formal result is non-differentiable
 (AD03) Active variable ...  written by I-O to file ...
 (AD04) Useful variable ...  read by I-O from file ...
```

  Some programs save `REAL` variables `r1` into arrays of, say, `INTEGER`'s, or maybe just bytes. Later, they read the values from where they were saved, and put them back into `REAL` variables `r2`. We call this *casting* REAL's into and from another type. This is poor programming practice in general. We also consider writing into a file and reading back from this file as an extended sort of casting. The problem is that TAPENADE does not hold derivatives for INTEGER's. Therefore the chain of *activity* is broken between *r1* and *r2*, and the computed derivatives will be wrong. When TAPENADE detects this situation, one must do without this casting. Otherwise, if this is impossible, there is a special option in TAPENADE to actually attach derivatives to these strange "INTEGREAL's".

4.6.5 Further developments

Our goal is to promote the use of TAPENADE in the scientific computing community, and more importantly the use of the reverse mode of AD for optimization [12, 22, 35] and inverse problems [37]. Discussion with end-users drives our research very strongly.

We are currently extending TAPENADE in several directions. A new version that fully accepts FORTRAN95 is coming soon, and C in next on the list. Also, we are still unsure of the best program representation for object languages. Program static analyses will be developed further, particularly pointer analysis. There is also work to be done in the definition of directives used by TAPENADE to drive AD efficiently. Our research work, focused on the reverse mode, progressively suggests improvements into the tool. For example the dead adjoint code analysis still needs development to reach the results shown in section 4.4.4

Chapter 5

Conclusion

This report presents a summary of research done in the field of Software Engineering tools for Computer Science applications. Starting from research in automatic parallelization of programs, our goal is to transpose these parallelization techniques into the younger field of Automatic Differentiation, which is now our main research direction. We think we could bring new algorithmic ideas into AD, and there is still room for new research.

We shall conclude on the next open questions we envision for AD. As a relatively new technique, AD is still not an obvious choice for many specialists of scientific computing. Its development depends on the quality of both usage and computation of derivatives. Therefore, AD is still struggling on two main fronts:

- The numerical science front: now that AD provides analytic derivatives, convince numerical scientists to use those, exhibiting efficient way to use AD derivatives in scientific programs?
- The computer science front: What are the most efficient models to compute derivatives, for instance avoiding redundant computations of derivative expressions, or finding optimal tradeoffs between storage and recomputation for the reverse mode.

We didn't say much on the numerical science front, except in the AD illustration sections 4.2.1 and 4.2.2. One frontier of numerical science is the present jump from simulation to optimization of systems. This is by no means easy, but extremely important. It is actually so important that users are prepared to invest a lot of effort and computation time for it: just consider the current interest for "evolutionary" or "genetic" black-box approaches, which simply replace mathematical optimization by an extremely expensive clever exhaustive search. We believe gradient-based optimization is still an essential ingredient of any optimization method in scientific computation, and this is a frontier for AD, especially when iterative solving is used. Whereas AD differentiation of non-iterative pieces of code can follow mechanically the original control flow, researchers actively look for innovative models for AD of iterative loops. These models will probably combine purely automatic differentiation with application specific resolution tactics. Also, more efficient optimization schemes require not only the gradient of the cost function, but also its higher – mostly second – derivatives. Some AD tools provide some support for higher derivatives, but a unified framework is still missing.

Our main interest is the computer science front, in which we distinguish before algorithmic research and software tool development.

One challenge is to design a formalism that can capture both the *Store-All* and the *Recompute-All* strategies (*cf* section 4.1.3) for the reverse mode. This formalism should also capture strategies based on transformation to *single-assignment code* and allow detection of redundant derivative sub-expressions. With such a framework, we hope we can find optimal compromises returning adjoints of straight-line codes that are as good as some one would write by hand. Derivative sub-expressions are well captured by the *computation graph* (*cf* section 4.5.3), whereas the *dependence graph* is suited for manipulations of variables and instructions reordering. The new formalism should be a synthesis of these graphs.

Another challenge is to make checkpointing more flexible. Checkpointing is absolutely unavoidable on large codes, for example for optimization of unsteady processes. Checkpointing arbitrary pieces of code, designated by the end-user, is highly desirable. Also, successive checkpoints can share a part of their stored variables, and tradeoffs must be solved for this. There exist other flavors of checkpointing (*"reverse checkpoints"*) that we didn't investigate yet, that would deserve deeper studies.

The problem of differentiation of parallel programs that use asynchronous message-passing is still open. We still don't know how to reverse the control flow of an asynchronous execution.

Automatic Differentiation of programs around discontinuities is still an open problem. There is a theoretical description in [24, Chapter 11], and some run-time tests can be inserted by ADIFOR. We are studying a special differentiation mode that would evaluate the size of the neighborhood of the current input values in which no discontinuities occur.

About tool development, we know that TAPENADE can be largely improved, although it now surpasses the functions of its predecessor ODYSSÉE. TAPENADE still misses treatment of user directives for AD, and a decent analysis of pointers and dynamic memory. These problems will be even more important when C is an input language. Pointers are still a challenge for most AD tools, as well as differentiation of C. Nevertheless, the biggest challenge for AD tools is definitely differentiation of object-oriented languages (e.g. C++), which are speading more and more in Scientific Computing. Obviously, static knowledge of the control flow is weaker in object-oriented programs, and most AD models must be adapted to this new situation.

Bibliography

[1] L. Adhianto, F. Bodin, B. Chapman, L. Hascoët, A. Kneer, D. Lancaster, I . Wolton, and M. Wirtz. Tools for openmp application development: the post project. *Concurrency - Practice and Experience*, 12(12):1177–1191, 2000.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] J. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.

[4] B. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, 1977.

[5] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(1), 1986.

[6] C. Bischof, A. Carle, P. Khademi, and A. Maurer. The adifor 2.0 system for automatic differentiation of fortran77 programs. *IEEE Comp. Sci. & Eng.*, 3(3):18–32, 1996.

[7] P. Boulet. The bouclettes loop parallelizer. *Lecture Notes in Computer Sciences*, 1996. HPCN'96.

[8] A. Carle and M. Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University, 2000.

[9] D. Clément, J. Despeyroux, L. Hascoët, and G. Kahn. Natural semantics on the computer. In K. Fuchi and M. Nivat, editors, *France-Japan AI and CS Symposium, ICOT*, 1986. INRIA Research Report # 416.

[10] J.-F. Collard. *Reasoning about program transformations.* Springer, 2002.

[11] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann(editors). *Automatic Differentiation of Algorithms, from Simulation to Optimization.* Springer, 2001. Selected proceedings of AD2000, Nice, France.

[12] F. Courty, A. Dervieux, B. Koobus, and L. Hascoët. Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation. *Optimization Methods and Software*, 18(5):615–627, 2003.

[13] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(1):324–328, 1996.

[14] B. Creusillet. *Analyses de régions de tableaux et applications.* PhD thesis, Ecole des mines de Paris, 1996.

[15] B. Creusillet and F. Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, 1996.

[16] P. Cusdin and J.-D. Mueller. Improving the performance of code generated by automatic differentiation. Technical Report QUB-SAE-03-04, Queen's University Belfast, 2003.

[17] P. Dutto, C. Faure, and Fidanova S. Automatic differentiation and parallelism. In *Proceedings of Enumath 99, Finland*, 1999.

[18] C. Farhat and S. Lanteri. Simulation of compressible viscous flows on a variety of mpps : computational algorithms for unstructured dynamic meshes and performance results. *Computer Methods in Applied Mechanics and Engineering*, 119:35–60, 1994.

[19] C. Faure and U. Naumann. Minimizing the tape size. In *in [11]*, 2001.

[20] R. Giering. Tangent linear and adjoint model compiler, users manual. Technical report, 1997. http://www.autodiff.com/tamc.

[21] J.C. Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1:13–21, 1992.

[22] M.-B. Giles. Adjoint methods for aeronautical design. In *Proceedings of the ECCOMAS CFD Conference*, 2001.

[23] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[24] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Frontiers in Applied Mathematics, 2000.

[25] J. Grimm, L. Pottier, and N. Rostaing-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications and Tools*, pages 95–106. SIAM, 1996. rapport de Recherche INRIA 2794.

[26] L. Hascoët. Partial evaluation with inference rules. *New Generation Computing*, 6(2-3):187–209, 1988.

[27] L. Hascoët. A tactic-driven system for building proofs. In *7eme séminaire Programmation en Logique*, 1988. Rapport de Recherche INRIA 770.

[28] L. Hascoët. Specification of the partita tool. Technical report, EUREKA Report, EC Project 933 "EUROTOPS", 1994.

[29] L. Hascoët. Automatic placement of communications in mesh-partitioning parallelization. *ACM SIGPLAN Notices*, 32(7):136–144, 1997. Proceedings of 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

[30] L. Hascoët. The data-dependence graph of adjoint programs. Research report 4167, INRIA, 2001.

[31] L. Hascoët. A method for automatic placement of communications in spmd parallelisation. *Parallel Computing Journal*, 27:1655–1664, 2001.

[32] L. Hascoët, S. Fidanova, and C. Held. Adjoining independent computations. *in [11]*, pages 299–304, 2001.

[33] L. Hascoët, U. Naumann, and V. Pascual. Tbr analysis in reverse mode automatic differentiation. *Future Generation Computer Systems – Special Issue on Automatic Differentiation*, 2004. *to appear.*

[34] L. Hascoët and V Pascual. Tapenade 2.1 user's guide. Technical report 300, INRIA, 2004.

[35] L. Hascoët, M. Vázquez, and A. Dervieux. Automatic differentiation for optimum design, applied to sonic boom reduction. In V.Kumar et al., editor, *Proceedings of the International Conference on Computational Science and its Applications, ICCSA'03, Montreal, Canada*, pages 85–94. LNCS 2668, Springer, 2003.

[36] W. Kyle-Anderson, J. Newman, D. Whitfield, and E. Nielsen. Sensitivity analysis for navier-stokes equations on unstructured meshes using complex variables. *AIAA Journal*, 39(1):56–63, 2001.

[37] F.-X. le Dimet and O. Talagrand. Variational algorithms for analysis and assimilation of meteorological observations: theoretical aspects. *Tellus*, 38A:97–110, 1986.

[38] M. Metcalf and J. Reid. *Fortran 90/95 explained.* Oxford University Press, 1996.

[39] U. Naumann. Reducing the memory requirement in reverse mode automatic differentiation by solving TBR flow equations. In *Proceedings of the ICCS 2000 Conference on Computational Science, Part II*, LNCS. Springer, 2002.

[40] U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Prog.*, 2003.

[41] Feautrier P. Parametric integer programming. *RAIRO recherche opérationnelle*, 22:243–268, 1988.

[42] Feautrier P. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 1(1):23–51, 1991.

[43] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. Technical Report 91-50, Institute for Advanced Computer Studies, University of Maryland, 1991.

[44] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*. ACM, 2000.

[45] R. Sedgewick. *Algorithms*. Addison-Wesley series on computer science, 1988.

[46] Simulog S.A. *Foresys 2.0 reference manual*, 1998.

[47] J.-C. Sogno. The janus test: a hierarchical algorithm for computing direction and distance vectors. In *Proceedings of 29th Hawaii International Conference on System Sciences*, 1996.

[48] J.-C. Sogno. Analysis of multidimensional loops with non-uniform dependences. In *Proceedings of Int. Conf. on Advances in Parallel and Distributed Computing, APDC'97*, 1997.

[49] M. Vázquez, A. Dervieux, and B. Koobus. Aerodynamical and sonic boom optimization of a supersonic aircraft. Research report 4520, INRIA, 2002.