# Semantical Interprocedural Parallelization: An Overview of the PIPS Project

François Irigoin
Pierre Jouvelot
Rémi Triolet

CRI, Ecole des Mines de Paris, France
{irigoin,jouvelot}@ensmp.fr

## Abstract

PIPS is an experimental FORTRAN source-to-source parallelizer that combines the goal of exploring interprocedural and semantical analysis with a requirement for compilation speed. We present in this paper the main features of PIPS, i.e., demand-driven architecture, automatic support for multiple implementation languages, structured control graph, predicates and regions for interprocedural analysis and global nested loop parallelization, with an emphasis on its core data structures and transformation phases. Some preliminary results on the practical impact of our design choices are discussed.

This research is partially funded by DRET, under contract 87-017.

## 1 Introduction

Detecting the maximum level of parallelism in sequential programs requires a thorough understanding of their behavior. If numerous vectorizing and parallelizing compilers for scientific programs exist in both the academic ([ABCCF88], [KKLW84], [CCHKT88], [SG90], [AS89], [DLTKK90]) and industrial (VAST, FORGE, KAP) worlds, few are able to address the question of parallelism detection on a global basis, i.e. coupled with interprocedural information and comprehensive semantical analysis. The main goal of the PIPS (Parallélisation Interprocédurale de Programmes Scientifiques) project is to address this very issue and explore its effectiveness on real programs written in a real language.

PIPS is a source-to-source parallelizing compiler that transforms Fortran77 programs by replacing parallelizable nests of sequential DO loops with either Fortran90 [A90] vector instructions or DOALL constructs. It is not targeted towards a particular supercomputer, although only shared memory machines have yet been considered.

The principal characteristics of PIPS are:

- Interprocedural parallelization [T84]. It is at the core of PIPS: every step of the parallelizing process is able to cope with interprocedural information. A striking example of this drastic point of view is the parser, which encodes every assignment statement as a function call to a built-in = procedure with two arguments, the right and left hand side expressions. There is no "assign" node in the abstract syntax tree of programs and a user `CALL` would be encoded in the same way.
- Interprocedural analysis. To be most effective, a parallelizing compiler needs to gather as comprehensive as possible information about the behavior of programs. This is done in PIPS by a set of sophisticated (interprocedural) semantical analysis phases that compute side-effects (SDFI), regions [TIF86] and predicates in the style of [CH78].
- Efficiency. PIPS is fast enough to be used on real life programs. We use benchmarks provided by ONERA, a French research institute in aeronautics, and other more standard suites such as the PERFECT Club or LINPACK.

In the remainder of this paper, we discuss the general architecture of the PIPS parallelizer (Section 2), present the core data structures used within PIPS (Section 3), describe the different semantical analysis phases (Section 4), dependence tests (Section 5) and transformations (Section 6) applied by PIPS, give the current status of the project (Section 7) and conclude (Section 8). Given the breadth of this presentation, no particular section is dedicated to a discussion of the related work; relevant references are provided within each section.

## 2 PIPS General Design

Since PIPS is mainly a research project, we wanted its architecture to be modular and evolutive enough to adapt to the wide variety of people involved in the project (researchers, students, numerical analysts,...) and to the requirements of interprocedural analysis and interactivity. This dictated a structure in *phases*, all of which manipulate a common intermediate representation of programs that has been carefully designed and whose updates are under top supervision (see figure 1).

This architecture is hardly new. But a major issue is then the determination of the order according to which these different phases of the parallelizing process have to be scheduled to maintain consistency. PIPS adopts a relatively uncommon approach to this problem; the ordering is demand-driven, dynamic and automatically deduced from the declarative specifications of the dependencies between phases, by a program called `pipsmake`. Users are allowed to send requests to `pipsmake`, either to get a particular data structure or to apply a specific function.

Every data structure used, produced or transformed by a phase in PIPS, such as the abstract syntax tree, the control flow graph or the use-def chains, are considered to be *resources* which are under control of a data base manager, called `pipsdbm`. The place where the data actually reside (memory or file) is left to `pipsdbm` which can optimize these transfers according to the current memory space available. Data structures destroyed in memory by side effects performed by a given phase can be retrieved on disk, as well as data structured saved by previous PIPS runs, if they are still consistent.

Called by `pipsmake`, any phase begins by requesting some resources, via the `db_get` function provided by `pipsdbm`, performs some computation and declares the availability of its result via `db_put`. Every data structure is linked to its associated program unit. Currently, the units known by `pipsmake` are the whole `PROGRAM`, a given `MODULE` (i.e. a Fortran77 subroutine or function), the `CALLEES` (i.e., the list of all the modules that are called by a given module) and the `CALLERS` (i.e., the list of all the modules that call the given module). Via the notion of callees and callers, `pipsmake` manages the interprocedurality of PIPS.

Every phase in PIPS, such as the parsing of a source program or the privatization process, denotes a *function*; it may use and/or produce some resources. Every phase that exists in PIPS is declared via production rules; these are stored in a configuration file which is used by `pipsmake` to schedule the execution of each function, according to what the user and each phase (recursively) request. We give below an excerpt of this configuration file:

```
proper_effects  > MODULE.proper_effects
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.summary_effects

cumulated_effects       > MODULE.cumulated_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects

summary_effects > MODULE.summary_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
```

Here, for instance, the phase `cumulated_effects` computes the accumulated effects (such as a read on a given variable, called *entity* in PIPS) of all the statements of a `MODULE`. It needs the definition of all the entities of the `PROGRAM` - recall that PIPS is interprocedural -, the code of the module and its proper effects, i.e. the effects of the subroutine calls.

This architecture is demand-driven, since data structures are computed only when needed by some phase, and flexible, since each phase does not have to worry about the others and only requests resources at its outset. This organization permits the interproceduralty to be almost transparent to the programmer of a given phase since he can ignore how resources are computed. Each necessary resource is requested from `pipsdbm` and the appropriate function has previously been automatically activated by `pipsmake`, if the rules are correct. This allows for easy implementations of both top-down and bottom-up algorithms on the call graph of a program. Note however that it relies heavily upon the non-recursivity of Fortran77. It also provides the database structure necessary to combine interprocedural analysis and the speed necessary for interactivity: objects are computed only when they are needed and when they are not already available and up-to-date.

## 3 Data Structures

PIPS is organized around a core data structure that implements the abstract syntax trees of Fortran77. Since we expected to look at other languages than Fortran77 in the future, we carefully designed our Intermediate Representation (IR) to avoid to stick too closely to Fortran77 idiosyncrasies. Based upon experience, we also knew that numerous constructs in Fortran77 were mere syntactic sugar and could be abstracted as function calls, thus avoiding a huge set of cases that would have to be treated separately, even though they are all of the same. Occam's razor has been applied to its most extreme; there are only three cases of expressions and five cases of statements ! Even so, almost all Fortran77[1] has been embedded inside this IR.

The data structures that implement the IR used in PIPS are defined with the software engineering tool NewGen [JT89], developed by Pierre Jouvelot and Rémi Triolet. NewGen allows the definition of data *domains* in terms of basic ones, like `integer` or `float`, that can be combined to form products, sums, lists or sets. From these definitions, expressed in a simple language, NewGen generates a set of creation, manipulation and destruction functions on objects and values of these domains. We give below, as an example of NewGen compactness and PIPS philosophy, the definition of the expression domain used in PIPS:

```
expression = reference + range + call ;
reference = variable:entity x
            indices:expression* ;
range = lower:expression x
        upper:expression x
        increment:expression ;
call = function:entity x
       arguments:expression* ;
```

Here, we can see that a node for a `call` expression, representing a Fortran77 `FUNCTION` call, includes the `function` entity and the list of `arguments` expressions. As said above, an assignment is encoded in the same way, with a pseudo-intrinsic function = and two arguments denoting the left hand side and the right hand side of the assignment. This is possible since the implicit call mechanism is by reference.

The functions and macros generated by NewGen can be in either C or CommonLISP, the two languages currently

[1] The most significant restrictions are multiple entry points together with assigned and computed gotos, which can, most of the time, be desugared into more standard constructs. The very common BUFFER IN and BUFFER OUT extensions have been added to accept benchmark programs.

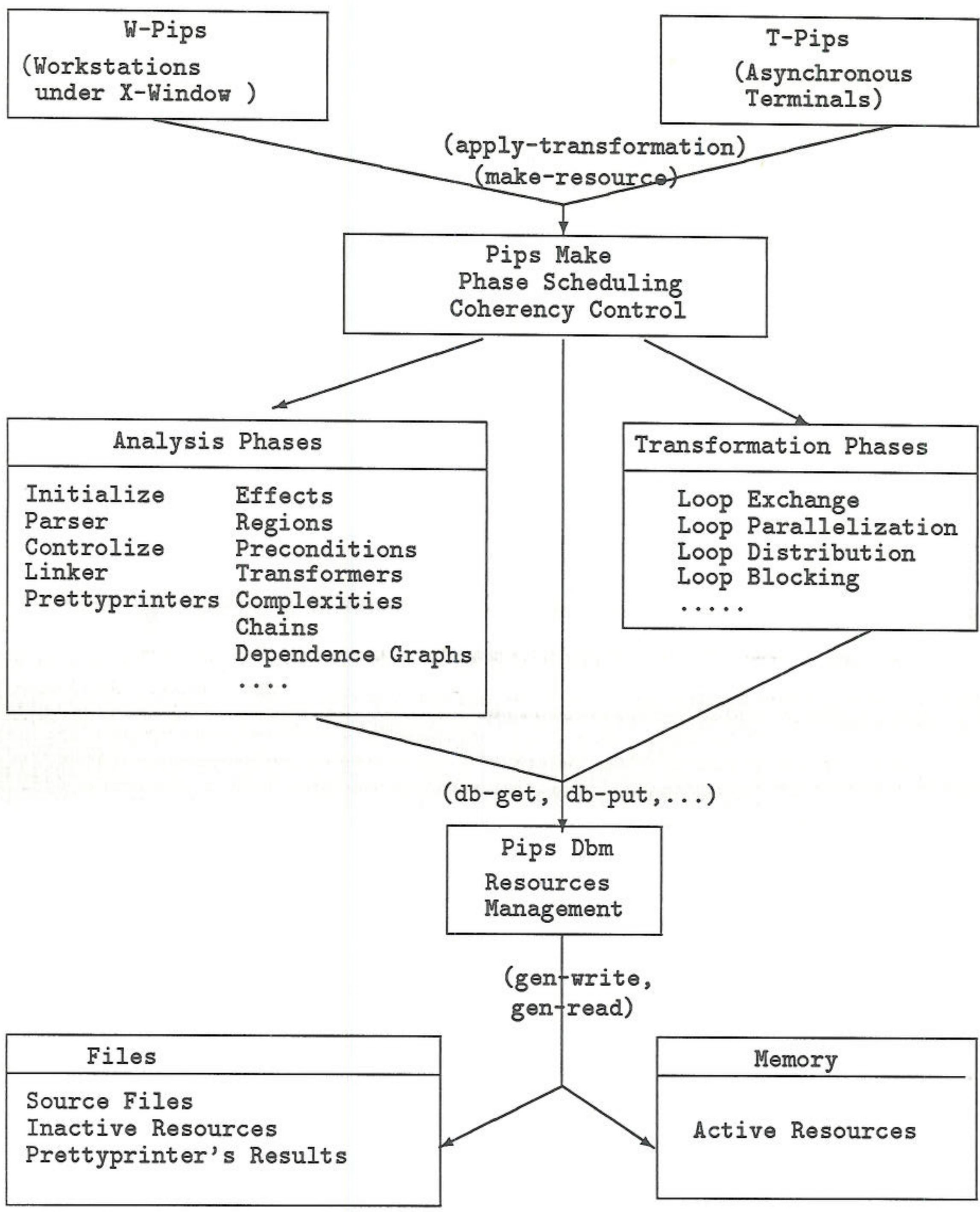


Figure 1: PIPS Structure

supported. These two versions are compatible at the file system level since NewGen automatically generates functions that can read and write data structures on files in a language-independent format. This permits a programming style reminiscent of *persistency*, a feature heavily exploited by pipsdbm.

CommonLISP is used as a prototyping language. For instance, a given exploratory phase can thus be quickly programmed in this highly expressive language before being efficiently recoded in C, once the algorithms have been designed and polished in the Lisp environment. The incremental linker of PIPS was first programmed in CommonLISP and rewritten in C. The implementation of *generalized reduction* detection [JD89] is being carried out in CommonLISP.

## 4 Semantical Analysis

We discuss in this section the control graph used in PIPS, together with the interprocedural computation of memory effects, regions [T84] and predicates, based on the intraprocedural computation of the same information on the statements of modules.

### 4.1 Control Graph

Semantical analysis[2] of programs is performed on their Hierarchical Structured Control-flow Graph (HSCG). This data structure is unique to PIPS among parallelizers[3] and its inception can be traced back to the notion of *control effect masking* described in [JG89]. The basic premise is that useful parallelism is only found in structured parts, which are at the same time easier and faster to manipulate and analyze, and very common in scientific programs like the benchmarks we were given. Most analysis algorithms for structured code have a simple recursive description by structural induction on the abstract syntax tree definition of the programming language. The presence of random branches destroys this nice property and imposes the computation of fixed points by iteration on the control graph.

The *controlizer* phase of PIPS, which computes the hierarchical structured control flow graph of a program, tries to keep the influence of branches as local as possible so that if, for instance, a label is only used within and from within a loop body, the loop will appear as a structured construct from the outside; control vagaries are masked. The HSCG is thus a layered data structure where each layer is either a structured construct or a graph-based description of a non-structured piece of code. The algorithms that manipulate HSCGs are successively defined by recursion and fixed point iteration. In particular, note that the presence of local branches in loops does not prevent parallelization, if the data dependences do not create conflicts.

The control graph is defined by the domain `unstructured` whereas structured statements are either basic commands altering the store with side-effects (`call`), sequences (`block`), tests (`test`) or DO loops (`loop`). The NewGen declarations, slightly simplified for the purpose of exposition, look like the following:

```
statement = label:entity x
            number:int x
            comments:string x
            instruction ;

instruction = block:statement* + test + loop +
              call + unstructured ;
test = condition:expression x
       true:statement x
       false:statement ;
loop = index:entity x
       range x
       body:statement x
       label:entity ;

unstructured = control x exit:control ;
control = statement x
          predecessors:control* x
          successors:control* ;
```

All the semantical analysis phases are defined by induction on the `statement` domain.

### 4.2 Effects

Effects describe the memory operations performed by a given statement. Besides the reference on which the operation is performed and its kind (read or write), PIPS distinguishes between effects that are always performed and the ones that may be performed.

*Proper* effects are memory references local to a given statement, such as a write on the left-hand side of an assignment or on the index variable of a `DO` loop construct. *Cumulated*[4] effects take into account all effects of a given statement, including those of its substatements (recall that the definition of the PIPS abstract syntax tree is recursive). Contrarily to proper effects, only the variable that is referenced is kept; in particular, in case of array references, index expressions are lost in this aggregation. *Summary* effects abstract the cumulated effects of a module by masking the effects performed on entities local to the module.

The proper effects of a call statement are derived from the cumulated effects of the body of the callee. Proper and cumulated effects are computed by an (automatic) bottom-up scan of the call tree, as can be seen in the following excerpt of the pipsmake rules for effects we saw previously:

```
proper_effects > MODULE.proper_effects
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.summary_effects

summary_effects > MODULE.summary_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects

cumulated_effects > MODULE.cumulated_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
```

[2] Semantical analysis denotes any kind of abstract interpretation of program and not only type checking or overloading removal as in classical compilers.

[3] High-Level Data Flow Analysis was proposed in [R77] and [S80] for classical compilers. Most academic parallelizers use either a classical low-level control flow graph or a program dependence graph [FOW87] or both. Another approach is to restructure the control flow graph before analysis are performed [TF89].

[4] Cumulated effects are also known as *use-mod* or sdfi information.

To give a better flavor of pipsmake's behavior, suppose no effects are available when the proper effects of some module are requested. Let P be this module and assume it calls Q, a leaf of the call graph. pipsmake will find the program entities and P's code on disk or in memory if parsing and linking have been performed[5] but not Q's summary effects. It will push them as a new goal and try to activate the rule for summary effects. But Q's cumulated effects are not available either, and the new goal becomes Q's proper effects, which can be evaluated because Q has no callees (Q was assumed to be a call graph leaf).

### 4.3 Predicates

To improve the accuracy of the effect information, to provide more information for dependence testing [TIF86] and to select better program transformations, statements are labelled with *predicates* that express constraints on the integer scalar variables, which are often used in references and loop bounds. This is most useful for array references, for instance to precisely define which subpart of an array is referenced by a memory operation and to refine cumulated effects into *regions* [T84][TIF86]. It is also used to improve dependence testing when inductive variables and triangular loop bounds are involved, and to select one vector loop or one parallel loop among many dependence free loops.

PIPS predicates are abstract commands, mapping a store to a set of stores [S77]. They are relations between the values of integer scalar variables in an initial store and a final store. The relations considered are polyhedra, represented by systems of linear inequalities and equalities and, sometimes, internally by generating systems, as in [CH78], because they provide a sound and general mathematical framework. Constant propagation, as well as inductive variable detection, linear equality detection [K76] or general linear constraints computation [CH78], can be performed in this framework using different operators to deal with PIPS basic control structures: sequences, tests, loops and control graphs (the so-called `unstructured`). More accurate operators are usually slower and PIPS users can interactively choose which ones are best for their program or module after viewing the results.

Unlike most analysis, we are dealing with abstract commands instead of abstract stores for two reasons. First, variables appearing in references cannot be aggregated to build cumulated and summary effects unless they denote the same value, i.e., they refer to the same store. The module's initial store is a natural candidate to be the unique reference store and, as a result, a relationship between this store and any statement's initial store is needed. Second, dependence tests are performed between *two* statements. Although a relationship between each statement store would be more useful than two stores, each relative to one statement, this would not be realistic because too many predicates would have to be computed. Thus a common reference to the same initial store seems to be a good trade-off between accuracy and complexity.

The drawback for using abstract commands instead of abstract stores is that the polyhedron dimensions are doubled, which is bad news for the exponential algorithms used to deal with polyhedra [CH78]. This was made less of a problem by exploiting the hierarchical structured control graph to compute predicates in an as small as possible environment.

A two-phase algorithm was designed and implemented. The first phase computes abstract commands bottom-up from the `call` statements to the sequences, tests and loops up to whole modules, and bottom-up from leave procedures to the main module. This is the most CPU intensive part but it benefits from locality. Each procedure is analyzed only once. The second phase is a top down phase. It propagates information about the initial store of the main program downwards to the `call` statements and to the called procedures, down to the simplest statements of the leaf procedures. Each procedure again is analyzed only once and receives as initial store the convex hull of all stores of its call sites.

It is difficult to compare this analysis with previously published methods either intraprocedural, like [CH78][6], or interprocedural, like [CCKT86], for two reasons. The first one is that many lattices are used, even though some algorithms are lattice independent. The second one is that different sets of operators can be used for a given lattice producing more or less accurate results. To put it in a very simplified way, return functions à la [CCKT86] are computed over a [CH78] lattice in the first phase, while jump functions are computed in the second phase. Example 1 of [CCKT86] could be handled if the assignment in *ralph*:

```
b=a*c/2000;
```

were linear.

### 4.4 Regions

Regions were defined [T84][TIF86] and implemented a first time in Parafrase [T85] by Rémi Triolet. Since then many different methods have been published and sometimes implemented. They belong to two broad classes [C90]. In the first class, the effects of call statements are somehow aggregated within each procedure using some lattice, at the cost of an accuracy loss and of a dependence test extension, but with the benefit of a reduction in the number of dependence tests to be performed to parallelize the callers [T84][C87][CK88][BK89]. In the second class, these elementary effects are simply gathered, which may benefit accuracy, but have to be tested independently to parallelize each caller, which induces a time penalty [L89] and, sometimes, requires a dependence test modification [BC86].

The region method as described in [T85] et [TIF86] did not look very attractive mainly because the dependence test used was shown to be very slow[7] and because the convex hull algorithm used to aggregate regions is potentially slow too. It was decided to implement it anyway in PIPS because worst-case exponential algorithms are not necessarily exponential on practical cases [TIF86]. This is especially true if restrictive assumptions are made on the region shapes as in [BK89]; once it has been shown that most regions belong to some subset, algorithm complexity should be evaluated on that subset. Experimental results are needed here to usefully compare methods.

[5] If not, pipsmake will activate the parse and link phases whose rules have been omitted here.

[6] In his PhD thesis Halbwachs gave hints about interprocedural extensions and program structure exploitation but no implementation seems to have ever been done.

[7] Slowliness reported in [T85] was also due to the implementation. Rémi Triolet had very little time to finish his work at CSRD.

From a theoretical point of view, regions could (but should they?) be improved by preserving some integer lattice information when possible, instead of giving up by systematically using convexity to reduce the information amount. For instance, convexity reduction in `T(2*I)` will lose the even parity information of the index expression. This could easily, from a mathematical point of view, be done by characterizing subparts of an array as affine images of polyhedra.

From a practical point of view, region implementation in PIPS is underway and only very simple cases have been handled up to now. Our key benchmarks were also analyzed by hand. Three out of four contain some interprocedural parallelism within DO loop bodies. Since regions would not give better results than array slices, subarray-based detection was quickly implemented as an extension of effect translation, the process that maps the summary effects of the callee to the proper effects of the CALL statement. At each CALL site, the summary effects of the callee are checked and ranges are used to express effects on subarrays, such as columns or set of columns.

## 5 Dependence Tests

The dependence test algorithm [B88] is a critical part for any parallelizer. Under the usual linear assumptions on loop bounds and array index expressions, integer programming provides an exact test at a very high cost [TF89]. Historically, simpler tests have been developed to speed up dependence testing because the number of tests to perform increases as the square of the number of references in the statements of interest.

Recently people have tried to develop multi-precision tests. Fast tests are used first. If a positive or negative conclusion is obtained, the test is finished. If a *don't know* answer is returned, a more sophisticated test is applied.

Although we know it could be easily improved, we used the Fourier-Motzkin pairwise elimination technique described in [TIF86], with slight variations. First of all, equations can be solved exactly before inequalities are combined and can show dependence or independence very often. Second, the whole dependence system built with the array references and the execution contexts (called *predicates* in PIPS) does not have to be solved for each dependence level [AK87] or for each dependence direction vector [W89].

It is sufficient to project[8] the dependence system on the dependence subspace once and for all, and either to quickly conclude when a constant dependence vector is found or when the equations have no solution, or to add additional constraints to the small system obtained in the dependence space. Its dimension is bounded by the number of enclosing loops[9] and additional constraints are very simple since they are either simple equalities, like $di = 0$, or simple inequalities, like $di > 0$ ($i$ is assumed to be a loop index and $di$ the corresponding dependence direction).

Also, the algorithm complexity seems to adapt to the dependence system intrinsic difficulty although it is very simple to program. Diagonal systems, which do not need *simultaneous testing* [BC86], are solved linearly and constant dependences are detected before inequalities are involved. Finally, measurements show that dependence testing does not take a noticeable amount of time compared to the whole parallelization process, at least in our implementation.

[8] Linear programming, as used in [LT85], or integer programming techniques are faster to decide emptiness but do not preserve enough information for projection.

[9] Depending on previous program transformations, inductive variables and the like may have to be taken into consideration, increasing the sub-system size.

## 6 Transformations

A number of transformation phases have been implemented or planned inside the PIPS parallelizer. We will discuss below the following ones : privatization, detection of induction variables and reductions, distribution of loops and loop rescheduling.

### 6.1 Privatization

Some variables are used as local temporaries inside loop bodies. They are assigned on loop entries and their values on body exit are not used by subsequent iterations. This condition is checked in the *privatization* phase by looking at the use-def chains; all variables that satisfy this condition are flagged as local to the loop. Whenever such a loop is prettyprinted, a `PRIVATE` pragma is inserted into the output and the related dependences are deleted. Privatization increases the potential for parallelism detection, without the memory penalty of scalar expansion [W89].

### 6.2 Induction Variables and Reductions

Loop invariants, induction variables and reductions such as inner products are special cases of the so-called *generalized reductions*. A prototype implementation of the technique described in [JD89] is being written. It is based on symbolic evaluation of loop bodies and pattern-matching of the resulting symbolic store against a database of known cases.

This phase could be used to replace references to induction variables inside a loop body by appropriate expressions built over loop counters. By eliminating spurious dependences created by these variables, this would improve the potential for parallelism detection. Since inductive variables are also found by the semantic analysis, it will be interesting to compare the two method effectiveness.

The eventual goal is to detect reductions on real variables and to replace sequential loops by parallel reductions or by a proper library call.

### 6.3 Loop Distribution

The algorithm used in PIPS to generate parallel code is based on the technique described in [AK87]. The dependence graph is structured into strongly connected components, each of which is recursively analyzed with an incremented dependence level. Depending on whether the target architecture has a vector facility or not, it can be interesting to replace parallel loops that have more than one assignment statement in their bodies by a set of single assignment loops. These simple loops can then be replaced by vector instructions by the code generator. This transformation, called *loop distribution*, is provided by PIPS. It is coupled with the prettyprinter which is able to generate instructions with a syntax compatible with Fortran90.

### 6.4 Nested Loop Parallelization

Even if the initial order in which iterations of a loop nest body are executed prevents the straightforward parallelization of these loops, there may be ways to perform a change

of basis in the corresponding iteration space to exhibit some loop parallelism. Based on the dependence directions between loop statements, the *hyperplane* method [L74] is one of the numerous techniques, like loop interchange and loop skewing, that perform a unimodular change of basis and that belong to this class of transformation.

A first prototype of global loop nest parallelization was implemented as an experimental phase in the PTRAN project [I88], using dependence direction vectors [W89]. The unimodular change of basis choice was based on a heuristic and designed for a vector multiprocessor like the IBM 3090 VF. A loop direction with as many as possible contiguous memory accesses was chosen as inner vector loop and another parallel loop was chosen as parallel outer loop to define parallel tasks. Other parallel loops were kept sequential. A more general version, using dependence cones to characterize the loop dependence set, was presented in [IT88b].

A PIPS implementation of this technique is under development. Program transformations are performed automatically, but the choice of the transformation is still left to the user at the time being.

## 7 Status and Preliminary Results

PIPS has been under development for the last two years and required six man-years of design, programming and testing. It is written in about 50k lines of C, although some prototype phases were written in CommonLISP. A window-based user interface has been added, written on top of the X11 Window System with the XView Toolkit. It runs on Sun 4 and Sparc workstations, under the SunOS operating system.

Parts of our test suite include four medium-sized (from one to three thousand lines of sparsely commented code) numerical programs from ONERA (Office National d'Etudes et de Recherche Aéronautiques), a government-funded agency for aerospace research. They were run on PIPS to assess both its performance and its efficiency.

The performance of the whole system is difficult to assess since it depends on input-output file transfers implicitly performed by `pips_dbm`. Currently, all the data structures are saved on disk, which means that a significant performance penalty is incurred. A more intelligent manager would cache some of them in memory and delay the disk operations to the point where they are required.

A few conclusions can be drawn from our preliminary results. First, interprocedural parallelism detection would benefit from better programming practice :

- Formal arrays should be dimensioned as accurately as possible. Declarations like `A(1)` should be prohibited and declarations like `A(*)` should be avoided as often as possible. Some parameter `N` is usually passed to loop over `A` and should be used to declare `A(N)`.
- Temporary arrays should be avoided. In ONERA benchmarks, many copies could be replaced by better procedure call or index expressions.
- Implicit conditions on key parameters read from disk should be explicitly checked. The program would be safer and semantical analysis could automatically propagate this extra information where it is needed.

Second, new compilation techniques are needed to achieve a reasonably good automatic interprocedural parallelization, while others are not as useful as expected :

- Regions seem to be overprecise to summarize the effects of modules. More attention should be paid to effect translation, from effects on formal parameters to effects on actual ones.
- Interprocedural semantical analysis gathers potentially useful information but, often, a key piece of information is missing.
- Array expansion [F88] or privatization are needed.
- Kill information about arrays would be useful to mask procedure effects on static arrays.
- General cases of array reshaping should be handled.

## 8 Conclusion

The PIPS project strives to reach three objectives: (1) confirm interprocedural analysis as a useful basis for parallelization, (2) justify the use of sophisticated semantical analysis techniques to improve the effectiveness of parallelism detection and (3) show the practicality of these two approaches with actual programs written in full Fortran77. By using both standard software engineering tools such as Lex and Yacc and the NewGen program generator, developed in-house, these ambitious requirements can be fulfilled with a limited development team of three people.

The PIPS parallelizer is still under development and but preliminary experiments on real programs have been successfully carried out. The effectiveness of parallelism detection cannot be evaluated without hand analysis, almost impossible on real programs, or without another parallelizer. The project evaluation is also hampered by the lack of a target machine; we expect to tune PIPS for Cray-like machines in the near future.

New parallelization techniques are also going to be introduced in or derived from it, such as code partitioning [IT88a] and data movement generation [An90] for shared memory and, eventually, distributed memory supercomputers.

## References


[A90] ANSI X3J3/S8.115. Fortran90. June 90

[An90] Ancourt, C. *Génération Automatique de Codes de Transfert pour Multiprocesseurs à Mémoires Locales.* PhD thesis, Université Paris 6, 1991

[ABCCF88] Allen F., Burke M., Charles P., Cytron R., and Ferrante J. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing* , Vol. 5, No 5, Oct. 1988.

[AK87] Allen, R., and Kennedy K. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Langages and Systems*, Oct. 87.

[AS89] Appelbe, B., and Smith, K. Start/Pat: A Parallel Programming Toolkit. *IEEE Software*, Jul. 89.

[B88] Banerjee, U. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988

[BC86] Burke, M., and Cytron, R. Interprocedural Dependence Analysis and Parallelization. In the *Proceedings of the ACM Symposium on Compiler Construction*, 1986

[BK89] Balasundaram, V., Kennedy, K. A Technique for Summarizing Data Access and its Use in Parallelism Enhancing Transformations. In the *Proceedings of the ACM Symposium on Programming Languages Design and Implementation*, 1989

[C87] Callahan, D. A Global Approach to Detection of Parallelism. PhD Thesis, Rice University, 1987

[C90] Chassany, P. Les méthodes de parallélisation interprocédurale. Tech. Rep. EMP-CAI-I E/129, Ecole des Mines, 1990

[CCHKT88] Callahan, C. D., Cooper, K. D., Hood, R. T., Kennedy, K., and Torczon, L. Parascope: A Parallel Programming Environment. In the *Inter. J. of Supercomputer Applications*, Winter 88.

[CCKT86] Callahan, C. D., Cooper, K. D., Kennedy, K., and Torczon, L. Interprocedural Constant Propagation. In the *Proceedings of the ACM Symposium on Compiler Construction*, 1986

[CH78] Halbwachs, N., and Cousot, P. Automatic Discovery of Linear Restraints Among Variables of a Program. In the *Conference Record of the Tenth ACM Annual Symposium on Principles of Programming Languages*, 1978

[CK88] Callahan, D., and Kennedy, K. Analysis of Interprocedural Side Effects in a Parallel Programming Environment. *Journal of Parallel and Distributed Computing*, v. 5, n. 5, 1988

[DLTKK90] Dehbonei, B., Laurent, C., Tawbi, N., and Kulkarni, R. & S. PMACS: An Environment for Parallel Programming. In the *Proceedings of the International Workshop on Compilers for Parallel Computers*, Paris, Dec. 90.

[F88] Feautrier, P. Array Expansion. In the *Proceedings of the ACM International Conference on Supercomputing*, St-Malo, 1988.

[FOW87] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The Program Dependence Graph and it Use in Optimization. *ACM TOPLAS*, 1987

[I88] Irigoin, F. Loop Reordering with Dependence Direction Vectors. In the *Journées FIRTECH Systèmes et Télématique Architectures Futures: Programmation Parallèle et Intégration VLSI*, Paris, 1988

[IT88a] Irigoin, F., and Triolet, R. Supernode Partitioning. In the *Proceedings of the ACM Symposium on Principles of Programming Languages*, San-Diego, 1988

[IT88b] Irigoin, F., and Triolet, R. Dependence Approximation and Global Parallel Code Generation for Nested Loops. In the *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, Bonas, 1988

[JD89] Jouvelot, P., and Dehbonei, B. A Unified Semantic Approach For The Vectorization And Parallelization Of Generalized Reductions. In the *Proceedings of the ACM International Conf. on Supercomputing*, Crete, 1989

[JG89] Jouvelot, P., and Gifford, D. K. Reasoning about Continuation with Control Effects. In the *Proceedings of the ACM Conference on Programming Languages Design and Implementation*, 1989

[JT89] Jouvelot, P., and Triolet, R. NewGen: A Language Independent Program Generator. Rapport Interne CAII 191, 1989

[K76] Karr, M. Affine Relationships among Variables of a Program. *Acta Informatica*, 1976

[KKLW84] Kuck, D. J., Kuhn, R. H., Leasure, B., and Wolfe, M. J. The Structure of an Advanced Retargetable Vectorizer. In *Supercomputers: Design and Application*, IEEE Comp. Soc. Press, 1984

[L74] Lamport, L. The Parallel Execution of DO Loops. *Communications of the ACM*, 1974

[LT85] Lichnewsky, A., and Thomasset, F. Techniques de base pour l'exploitation automatique du parallélisme dans les programmes. INRIA Report 460, 1985

[L89] Li. Intraprocedural and Interprocedural Data Dependence Analysis for Parallel Computing. CSRD Report 910, 1989

[R77] Rosen, B. K. High-Level Data Flow Analysis. *Communications of the ACM*, 1977

[S77] Stoy, J. E. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, 1977

[S80] Sharir, M. Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers. *Computer Language*, 1980

[SG90] Shei, B., and Gannon, D. SIGMACS: A Programmable Programming Environment. In the *Proceedings of the International Workshop on Programming Languages and Compilers for Parallel Computers*, Irvine, Aug. 90.

[T84] Triolet, R. Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédure, PhD Thesis, Université Pierre et Marie Curie, 1984

[T85] Triolet, R. Interprocedural Analysis for Program Restructuring with Parafrase. CSRD Report 538, 1985

[TF89] Tawbi, N., and Feautrier, P. Parallélisation automatique de programmes pour ordinateurs multiprocesseurs à mémoire partagée. Tech. Rep. MASI, Université Pierre et Marie Curie, 1989

[TIF86] Triolet, R., Irigoin, F., and Feautrier, P. Direct Parallelization of Call Statements. In the *Proceedings of the ACM Symposium on Compiler Construction*, 1986

[W89] Wolfe, M. J. *Optimizing Supercompilers for Supercomputers*. MIT Press, 1989