CENTRE DE RECHERCHE EN INFORMATIQUE

Enhanced Static Evaluation of Fortran Programs in PIPS Environment

Lei ZHOU

December 17, 1991

Document EMP-CRI E/160/CRI

Enhanced Static Evaluation of Fortran Programs in PIPS Environment

Lei Zhou

Centre de Recherche en Informatique
École Nationale Supérireure des Mines de Paris

E-mail: zhou@cri.ensmp.fr

February 12, 1992

Abstract

In order to choose the relatively best optimized version of a real Fortran program, we have to compare the execution times of different optimized versions of the same Fortran program. Since the real scientific Fortran programs can run for hours on the expensive machines, it's useful to perform a static analysis without lack of the accuracy. In this paper, we propose to use the complexity, which is composed of polynomial and statistics, to present the time evaluations of the Fortran program. Then, one real example is presented in the PIPS programming environment. Finally, we shall mention what we plan to do about complexity in the near future.

Contents

Chapter 1

Introduction

Complexity plays an important role in computer science. For an algorithm, the cost of the algorithm is always presented, because we are interested in not only complexity but also results. For a machine, how fast it works is one of the key features to show its ability.

In this report, we'll talk about complexity...

1.1 PIPS project

PIPS is French acronym which stands for Paralléliseur Interprocédural de Programmes Scientifiques, developped at Centre de Recherche en Informatique de l'Ecole Nationale Supérieure des Mines de Paris.

PIPS is a source-to-source parallelizing compiler that transforms Fortran77 programs by replaceing parallelizable nests of sequential DO loops with either Fortran90 vector instructions or DOALL constructs. It is not targeted towards any particular supercomputers, although only shared machines have been considered. The principal characteristics of PIPS are:

1. Interprocedural parallelization.
2. Interprocedural analysis
3. Relative efficiency

See [IJT90] for more information.

1.2 Complexity in PIPS

Static evaluation is always useful to predict the performance of a program. Automatic parallelizers apply the program transformation to detect the hidden parallelism which is sometimes *forgotten* by the programmers or sometimes very difficult to find by human being. Among these tramsformations, some lead to an increase of parallelism, some, in the contrary, don't. Now comes a question: Which transformation would we use? In some

cases, we choose one ; in some other cases, we choose the other. A static comparison would help to make the decision.

There are several ways we can use to get the performance evaluation of a certain algorithm or certain Fortran program.

1. The easiest way is to run it on the real parallel machine with varing number of processors, such as Cray, Transputer, CM. it's not a good way sinply because it's the most expensive one.

2. The cheapest one is statically evaluate or simulate.

3. The most approriate one (at least I think) is to well balance the former two ways to get the best tradeoff.

Here is a brief sketch of what's in the remaining chapters appendices:

- Chapter 2 shows the previous status of complexity program, explains the algorithm of complexity.

- Chapter 3 describes what we have added for the complexity program, that is the enhanced part of the program.

- Chapter 4 shows the result on a real example tmines.f, tells the difference between the results obtained by hand and by our enhanced complexity evaluator.

- Chapter 5 describes what we can do in the near future about the evaluator.

Chapter 2

Previous Status of Complexity

Static evaluation is always useful to predict the performance of the program. See [Bert] for more information.

2.1 Complexity's Components in PIPS

The complexity of PIPS project is composed of two elements: a polynomial and statistics counters about the variables.

2.1.1 Polynomial

The polynomial library has been created by several people of CRI, especially Pierre Berthomier. It's the component of C3 library. It's made from the monomes, which are made from vectors. The vector library is the essential part of C3 library.

2.1.2 Statistics

Although it's almost impossible to get the precise execution time of a given set of instructions, we can expect a good approximation. The statistics counters contain three kinds of counters to summarize the different sorts of approximations performed during the evauation.

Variable Counting

1. **symbolic** counts the variable which appears literally and needs not to be evaluated. What we need to know is its type.
2. **guessed** counts the variable which has to be evaluated and whose value can be found.
3. **bounded** counts the variable which has to be evaluated and whose value can not be found exactly. In this case, we chose the worst estimation.
4. **unknown** counts the variable which is totally unknown.

Range Counting

1. **profiled** counts the loop range whose range is measured with some profiling. (not implemented)
2. **guessed** counts the loop range whose bound contains variable of the kind *symbolic* and *guessed.*
3. **bounded** counts the loop range whose bound contains variable of the kind *bounded.*
4. **unknown** counts the loop range which is totally unknown.

If Counting

1. **profiled** counts the test whose probability was measured. (not implemented)
2. **computed** counts the test whose probability was computed.
3. **halfhalf** counts the test that we know nothing about and whose probability was supposed to be fifty-fifty.

2.2 Target machine model

Each computer manufacturers has its own way to deal with the numerical computations. As computer technology advances, the supercomputer is getting more powerful. We need to know the approximate cost for each operation, memory accesses, etc.

2.2.1 Cost Table

We present here the ideal complexity of the basic operators. It is the first step towards the real world. Our purpose is to let it work first, so the values have not the importance (surely, it will have in the future). For the time being, we give the complexity unit for every operation.

```
# cost_basic_ops_1  - Unity cost for basic operations

#       int     float   double  complex double-complex    <-- type of argument(s)

+       1       1       1       1       1
-       1       1       1       1       1
*       1       1       1       1       1
/       1       1       1       1       1
--      1       1       1       1       1
**      1       1       1       1       1
```

Equally, we have the other cost tables for memory access, array element address computation and transcendental functions which have the same format as the above one.

2.2.2 Implemented machines

The simplest model has been chosen for a prototype implementation. The machine has one ALU and one FPU, and they don't overlap. There is neither cache memory, nor virtual memory system. With some more time, we would have included a parametrable number of processors: either a constant, or a symbolic pseudo-variable that would have appeared in the expression of the complexity. For the time being, the evaluator only processes sequential programs. Vector processing isn't either taken into account. File inputs/outputs can only receive a static constant cost, as well as each Fortran intrinsic (static in the sense that it doesn't depend on the environment in the source code). For the time being, we ignore the kind of cost. In the case of mathematical functions, a cost is defined for each type of the arguments: integer, floating-point, double precision, complex numbers.

2.3 Complexity Algorithms

We initially started the evaluation of the complexity in terms of floating-point operations; then it gradually appeared that integer operations could be equally important, and also that after all, memory accesses could as well be the real bottleneck, so we decided to count them all.

2.3.1 Complexity of a variable

The cost of a variable depends upon its type, then we can find the cost in the cost table of memory.

2.3.2 Complexity of an operator

To know the cost of an operator, one must first know the types of its arguments to determine which kind of operation takes place, then find the relative cost for the operator in the cost table.

2.3.3 Complexity of an expression

The evaluation of an expression cost is complicated by the overloading of most arithmetic operators. To know the cost of an addition, for instance, one must first know the type of its arguments. So we evaluate the expression bottom-up, beginning with the leafs of the syntax tree (constants, variables or function calls) which type is known. The types of sub-expressions are propagated towards the root. For example, $IM * JM$ will have the cost 3 according to our current cost table.

2.3.4 Complexity of a statement

We have differnet complexity for different statements.

Complexity of an assignment

We refer to it as *expression* = *expression*, so the total complexity equals the sum of the two expressions. For example, the following assignment has the the cost 5.

```
C                                                          5 (STAT)
      JJ = I+J-2
```

In PIPS, the assignment is considered as a call. Because the memory access has been costed, so the assignment is considered to have zero cost.

Complexity of a call

The complexity of a call is exactly the summary complexity of the corresponding subroutine or function.

Complexity of a Structured IF

$$\text{IF } boolexpr \text{ THEN } stat_{true} \text{ ELSE } stat_{false} \text{ ENDIF}$$

This statement is structured if there is no `GOTO` jumping in or out of $stat_{true}$ or $stat_{false}$. Let's call p the probability that *boolexpr* is true, q the probability that it is false, $p+q=1$. We use the following probabilistic definition of the complexity:

$$C(IF...) = C(boolexpr) + p.C(stat_{true}) + q.C(stat_{false})$$

If *boolexpr* isn't particular, as a first coarse approximation, awaiting for run-time measures, we use the values $p = q = \frac{1}{2}$.

Complexity of a Sequential DO

Suppose that we have a loop:

$$\begin{array}{c}\text{DO } index = lower, upper, increment \\ body \\ \text{ENDDO}\end{array}$$

The evaluation of *lower*, *upper*, *increment* may call functions, whose complexity must be summed to the overall execution time. The complexity of the body may depend on the index, so we must integrate it on the index rather than multiply it by the range width. We hereafter include in C_{body} the complexity of the loop index test and index incrementation.

$$C(DO...) = C_{lower} + C_{upper} + C_{increment} + \sum_{index=lower}^{upper} C_{body} \tag{2.1}$$

This formula applies if we are able to properly evaluate *lower*, *upper* and *increment* as polynomials.

For the sequential loop as well as the parallel ones, the statistics of the loop are computed by adding those of the expressions *lower*, *upper*, *increment*, and *body*.

Complexity of a Parallel DO

$$\begin{array}{c}\text{DOALL } index = lower, upper \\ body \\ \text{ENDDO}\end{array}$$

The complexity of the parallel loop is equal to the complexity of its largest iteration:

$$C(DOALL...) = C_{lower} + C_{upper} + \max_{lower \leq index \leq upper} C_{body} \tag{2.2}$$

This maximum is impossible to find in the general case. But note that thebest performance is obtained when all iterations last the same time, for all processors to run together. So when we are unable to compute the maximum, we can approximate it with the complexity of the first iteration.

2.3.5 Complexity of a block

$$\begin{array}{c} statement_1 \\ statement_2 \\ \dots \\ statement_n \end{array}$$

Once the control graph is computed, the construction of PIPS internal representation of programs garantees that there is no `GOTO` jumping in or out of the middle of a block, so that the n statements are always executed sequentially. The complexity of the block is simply:

$$C(block) = \sum_{i=1}^{n} C(statement_i)$$

The statistics of the whole block is the sum counter to counter of the statistics of its statements. PIPS detects only loop parallelism (nor `COBEGIN ... COEND` neither `FORK, JOIN`).

2.3.6 Complexity of an unstructured control flow graph

Control flow graph and PIPS programs representation

Any Fortran program, even those containing `GOTO`s, can be represented by a graph whose nodes are structured or elementary instructions, and whose edges stand for `GOTO` jumps. Most of nodes have one outgoing edge, `IF-GOTO-ELSE-GOTO` nodes have two, and the computed and assigned `GOTO` can have more. This graph is called a control flow graph ([ASU86]); PIPS' outline internal representation is such a graph. Moreover, little unstructured pieces of code are encapsulated into one node in such graphs and viewed from the rest of the program as if they were single, structured blocks of instructions.

Complexity of an unstructured control flow graph

Let's now define a complexity for an unstructured program represented by its control flow graph. Let $\{S_1, S_2, \ldots, S_n\}$ be the set of the graph nodes, and S_1 the entry node. We are sure there is only one, because of the definition of the graph: if there were more than one entry point, that would mean there would be `GOTO`s reaching from outside of the graph into the middle of it; the origin vertex would actually be a part of it.

Let $c_1, c_2, \ldots, c_n$ be the complexities associated with $S_1, S_2, \ldots, S_n$, and supposed known. Let's call p_{ij} the probability to go to node S_j when you are in node S_i. The probability to go to any node, from node S_i, is 1, so $\sum_{j=1}^{n} p_{ij} = 1$. We'll at last associate to each node S_i the average complexity g_i of the code still to be executed between S_i and the exit of the graph (g_i is a sort of "global cost"). Our goal is the evaluation of g_1 as it is the average complexity of the code executed between the first node S_1 and the exit node.

Here is a recursive definition of the g_i: the global cost of a node is its proper cost c_i plus the sum of the global costs of its successors g_j weighted by the associated probabilities p_{ij}.

$$g_i = c_i + \sum_{S_j \text{ successor of } S_i} p_{ij} \cdot g_j \tag{2.3}$$

Chapter 3

New Features of Complexity

3.1 Complexity Zero

The complexity consists of two elements : polynomial and statistics. Ideally, we give the statement `CONTINUE` zero as complexity value. It has the same idiosyncrasis as arithmetic zero. When the polynomial is null, we refer to it as the complexity 0. There was an implementation problem at the beginning, because it was just a symbol to present complexity zero, it had not the memory space. So we have to define complexity zero to deal with this specific case.

3.2 Sorted Complexity Output

To make the complexity output more readable, we sorted the complexity output according to the following criteria.

1. sum of the power number
2. alphabetical order about the variable names
3. constant is always at the end (power equals zero)

The following is the complexity output of an example.

```
C                    2*M^2.N + 15*M.N^2 + 5*N^2 + 9*N + 3*K^(-1) - 8 (SUMMARY)
```

You see that, although the first and second term have the same sum power 3, we put `M^2.N` because the M is alphabetically smaller than `N`. The second to the last has power -1, we put it the second to the last because there's a constant.

3.3 Utilization of the preconditions

All the preconditions depend heavily upon the C3 libraries, especially upon the systeme constraint library where we can get all the information we want about the preconditions.

3.3.1 Exploitations of constants

Example:

```
n = 10
do i = 1, n
   ...
enddo
```

In this loop, N is replaced automatically by 10.

3.3.2 The check of the effects

Example:

```
n = f(..)
do i = 1, n
   ...
enddo
...
n = g(..)
do i = 1, n
   ...
enddo
```

Note that N has been changed between the two loops. We have to use different values for each loop. If they are unknown, the complexity result can not be added.

3.3.3 Final complexity result using formal parameters

When a subroutine is called, maybe several formal parameters have been passed. The complexity results should contain the formal parameters as output and at the same time, delete the *middle* variable(s). The same thing for the COMMON variables and explicitly-asked variables.

Look at the following example:

```
      subroutine sub(a,m,n)
      real a(m)
      integer m,n
      k = 3 * n + 2
      do 10 i = 1, k
         t = t + 1.0
10    continue
      return
      end
```

For this example, the complexity result should contain n instead of k , for k is a private integer and is not known by the outside world and is called *middle* variable.

We'll point out that no matter how complex the function is , as long as the function is linear, the final complexity result always contains k as its component.

3.3.4 Nested-loop case

One of the most tricky things comes here. When inner loop bound is a linear function of outer loop index.

Example:

```
      subroutine sub2(m)
      integer m
      do 10 i = 1, m
         ii = i + 1
         do 20 j = ii, m + 2
            jj = i + j - 2
            do 30 k = jj + 10, 100
               t = t + 1.0
               u = u + 1.0
30          continue
20       continue
10    continue
      return
      end
```

There are three loops here loop 10, loop 20 and loop 30 respectively, one is embedded into the other. For the innermost loop 30, its lower bound `jj + 10` is dependent on the outer loop 20's index, which is determined by the two outer loop indexs. Remember that we use the bottom-up method to accumulate the complexity. confined in this innermost loop 30, we don't know the relation between this loop and outer loop*s*. So we must obtain what we need in the innermost loop. Naturally preconditions comes into use.

We give the output of complexity result for this example in the following.

```
C                                                    29*M^2 + 98*M + 1 (SUMMARY)
SUBROUTINE SUB2(M)
INTEGER M
C                                                    29*M^2 + 98*M + 1 (DO)
      DO 10 I = 1, M                                                  0003
C                                                                   3 (STAT)
         II = I+1                                                     0004
C                          9*I^2 - 12*I.M + 3*M^2 - 84*I + 72*M + 135 (DO)
         DO 20 J = II, M+2                                           0006
C                                                                   5 (STAT)
            JJ = I+J-2                                               0007
```

```
C                                                          -6*I - 6*J + 6*M + 70 (DO)
            DO 30 K = JJ+2, M+10                                                0009
C                                                                             3 (STAT)
               T = T+1.0                                                      0010
C                                                                             3 (STAT)
               U = U+1.0                                                      0011
C                                                                             0 (STAT)
30             CONTINUE                                                       0012
C                                                                             0 (STAT)
20          CONTINUE                                                         0013
C                                                                             0 (STAT)
10       CONTINUE                                                            0014
C                                                                             0 (STAT)
         RETURN
         END
```

Our method is to put each induction variable into a hash table to let it not be evaluated by the complexity program when encountering a loop, and delete that induction variable when going out of that loop. So for the innermost loop 30, what we can see about the complexity contains only outer loops' induction variables `i`, `j` and formal parameter `m`. For loop 20, the result contains `i` and `m`, for outermost loop 10, the result contains merely the formal parameter `m`.

According to our complexity algorithm, The total complexity for the loop is `6*((M+10)-(JJ+2)+1)+4` that equals `-6*I - 6*J + 6*M + 70` by substituting `JJ` with the expression `JJ = I+J-2` . Where 6 is loop body complexity, 1 is added because `K` is read each time in the loop, and 4 is the range complexity, that is, 2 for the expression `JJ+2` (variable `JJ` and plus sign, each has one complexity unit.) The same thing for the expression `M+10`.

3.4 User Interface for complexity

As a part of the PIPS project, one can get the complexity output using the PIPS interfaces, that is TPIPS for "terminal" PIPS interface and WPIPS for multi-window PIPS interface. See [Baron] for more details.

The original interface for complexity in PIPS environment is to call `complexity` directly. After using TPIPS command `Init` to prepare everything, then we can use `complexity` .

The following session will help you run complexity program.

```
linz /home/users/pips/Pips/Development/Lib/complexity: cat s.f
        subroutine sub
        n = 10
        do 10 i = 1, n
           t = t + 1.0
 10     continue
        end
linz /home/users/pips/Pips/Development/Lib/complexity: Init -f s.f s
linz /home/users/pips/Pips/Development/Lib/complexity: complexity -p s SUB
```

The option `-p` means to save the complexity result in the database. When execution is done, we can find the complexity result in s.database/SUB.comp .

```
linz /home/users/pips/Pips/Development/Lib/complexity: cat s.database/SUB.comp
```

You can also use interative TPIPS to get result directly:

```
linz /home/users/pips/Pips/Development/Lib/complexity: Display -m sub comp
```

It calls the complexity program to obtain the complexity result and prints that in the screen.

3.5 Debugging tools for complexity

When the complexity evaluator is developped, at the same moment, two useful by-products came into existence too —— Tracing tool and Printing internal preconditions values.

3.5.1 Tracing tool

This tool is triggered by adding `-t` option to the executable complexity program. With it, we can know where we are when we have problems, it can print out useful information, including statement numbering; statement name etc. We use the above example s.f to show the tracing tool as the following.

```
complexities SUB
  >statement @, ordering 1
    >instruction
    | >unstructured
    |   >statement @, ordering 65537
    |     >instruction
    |     | >block
    |     |   >statement @, ordering 65538
    |     |     >instruction
    |     |     | >call TOP-LEVEL:=
    |     |     |   >arguments
    |     |     |     >expression
    |     |     |     | >syntax
    |     |     |     |   >reference SUB:N
    |     |     |     |     >indices
    |     |     |     |     <
    |     |     |     |   <
    |     |     |     | <
    |     |     |     <
    |     |     |     >expression
    |     |     |     | >syntax
    |     |     |     |   >call TOP-LEVEL:10
    |     |     |     |   <
    |     |     |     | <
    |     |     |     <
    |     |     |   <
    |     |     | <
```

```
|      |      <
|      |    <
|      |    >statement @, ordering 65539
|      |      >instruction
|      |      | >loop SUB:I label SUB:@10
|      |      |   >range
|      |      |     >expression
```

Omitted some parts of output.

```
|      |      |   <
|      |      | <
|      |      <
|      |    <
|      |    >statement @00000, ordering 65543
|      |      >instruction
|      |      | >call TOP-LEVEL:RETURN
|      |      |   >arguments
|      |      |   <
|      |      | <
|      |      <
|      |    <
|      |  <
|      <
|   <
|  <
 <
  <
<
```

3.5.2 Printing internal preconditions values

If we want to make sure that the preconditions are good enough , we can append the `-i` option to complexity program to print out the current internal precondition values. It will help us to check where the error emerges, in the C3 library or complexity program itself.

Chapter 4

A Real Example

In this chapter, we will let our complexity program run on a real Fortran program *tmines.f*, which is a compilable program from ONERA. (Office National pour l'Etude et Recherche de Aerospace).

4.1 Introduction to Fortran program TMINES

The Fortran program tmines.f, coded be Marc BREDIF, calculates the potential flow in a tube in varing rectangular section. It resolves potential equation and calculats irrotational flow. See [Emad] for more details.

Figure 4.1 shows the Interprocedural control flow graph. Among the most important subroutines, we can say:

- calmat matrice constitution
- romat decomposition
- calcg application

4.2 Manual Results

In order to check the results obtained by our parallelizer PIPS is consistent to the original ones, we need to know all the information about the Fortran program *tmines.f*. The work has been done by Nahid Emad. She has got the all the complexity results for all procedures of tmines.f. (See [Emad] for more details)

4.2.1 Programme TMINES.

Suppose *ni, nj* and *nk* be parameters used for the dimensions of the table in the program. Let :
$\alpha = ni * nj * nk, \beta = nj * nk, \gamma = ni * nj, \zeta = ni * nk$
$\tilde{\alpha} = (ni-1) * (nj-1) * (nk-1), \tilde{\beta} = (nj-1) * (nk-1)$
$\tilde{\gamma} = (ni-1) * (nj-1), \tilde{\zeta} = (ni-1) * (nk-1)$

The following table shows the floating point complexity of each routine called by TMINES or by one of its procedures :

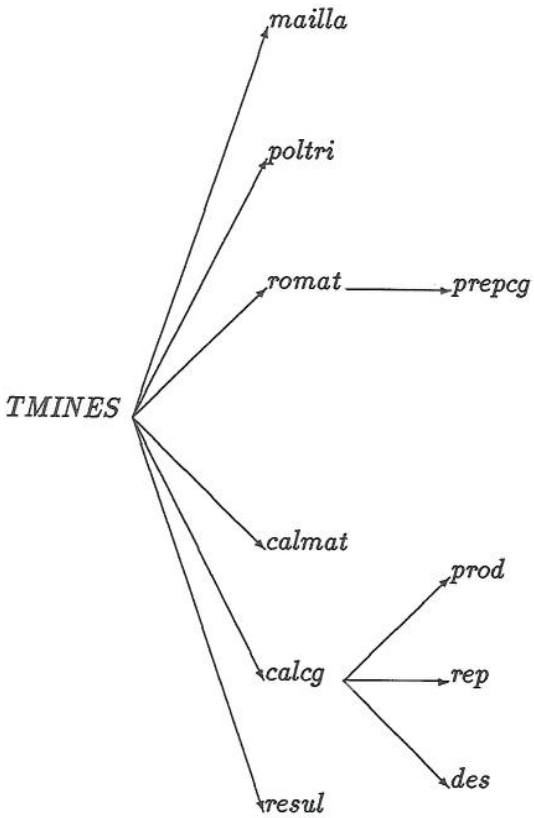

Figure 4.1: Interprocedural control flow graph of TMINES

procedure	C_{fl}
mailla	$7 * \alpha - 3 * \beta + 543 * ni - 517$
calmat	$8584 * \tilde{\alpha} + \tilde{\beta}$
prod	$54 * \alpha - 36 * \beta$
poltri	7277
romat	$600 * \tilde{\alpha} - 56 * \zeta - 26$
prepcg	$129 * \alpha - 56 * \zeta$
calcg	$nmax * (118 * \alpha - 36 * \beta - 36 * \zeta + 8)$
resul	$45 * ni - 28$
des	$27 * \alpha - 18 * \zeta$
rep	$27 * \alpha - 18 * \zeta$

4.3 Complexity Results

Generally our complexity result contains not only floating point complexity, but also one of integers, etc. So in order to check our result, we have to make some small changes of cost table to calculate merely floating point complexity.

4.3.1 Changed Cost Table

You have seen the general cost table in Chapter 2.

```
# cost_basic_ops_1  - Unity cost for basic operations

#       int     float   double  complex double-complex    <-- type of argument(s)

+       0       1       0       0       0
-       0       1       0       0       0
*       0       1       0       0       0
/       0       1       0       0       0
--      0       1       0       0       0
**      0       1       0       0       0
```

4.3.2 Complexity Output

The following complexity results are obtained by the complexity program with changed cost table. Notice that IM is ni, JM is nj and KM is nk.

procedure	C_{fl}
mailla	$7 * IM.JM.KM - 3 * IM.KM + 431 * IM + 5 * U\_RANGE - 422$
calmat	$5664 * IM.JM.KM - 5664 * IM.JM - 5664 * IM.KM - 5663 * JM.KM + 5664 * IM + 5664 * JM + 5664 * KM - 5664$
prod	$54 * IM.JM.KM - 36 * JM.KM$
poltri	7277
romat	$489 * IM.JM.KM - 369 * IM.JM - 425 * IM.KM - 369 * JM.KM + 369 * IM + 369 * JM + 369 * KM - 349$
prepcg	$120 * IM.JM.KM - 56 * IM.KM$
resul	$55 * IM - 46$
des	$27 * IM.JM.KM - 18 * IM.KM$
rep	$27 * IM.JM.KM - 18 * IM.KM$

For *calcg*, there is a line

```
NPMAX = IM*JM*KM
```

and NPMAX serves as upper loop bound. But this is not a linear function, so output of complexity program is very bad, we ignore that.

For mailla,

```
I2=IM/2 + 1
```

and I2 serves as upper loop bound. The U_RANGE in the table means I2.

You can see that for *prod*, *poltri*, *des*, *rep*, the results are identical. For the *prepcg*, I think that it was a typo of Nahid Emad, it should be 120 instead of 129. For *mailla*, the two largest terms are identical. For the *calmat*, *romat*, *resul*, there are several differences and I don't know who made the mistakes.

4.3.3 DES procedure

We choose the DES subroutine to show the whole complexity result.

```
C                                                  27*IM.JM.KM - 18*IM.KM (SUMMARY)
      SUBROUTINE DES(IM,JM,KM,T,B)
      INTEGER IM,JM,KM,JMM,KMM,K,KU,I,IL,IU,J
      REAL*8  T(1:IM,1:JM,1:KM),B(1:14,1:IM,1:JM,1:KM),TIK(1:25)
C                                                  27*IM.JM.KM - 18*IM.KM (UNSTR)
C                                                  27*IM.JM.KM - 18*IM.KM (BLOCK)
C                                                                0 (STAT)
      JMM = JM-1                                                  0002
C                                                                0 (STAT)
      KMM = KM-1                                                  0003
C                                                  27*IM.JM.KM - 18*IM.KM (DO)
C
C----------------DESCENTE-----------------------------------------------
C
      DO 100 K = 1, KM                                                  0005
```

```
C                                                          27*IM.JM - 18*IM (BLOCK)
C                                                                        0 (STAT)
C
      KU = MINO(KM, K+1)                                                  0006
C                                                          27*IM.JM - 18*IM (DO)
      DO 1 I = 1, IM                                                      0008
C                                                               27*JM - 18 (BLOCK)
C                                                                        0 (STAT)
C
         IL = MAXO(I-1, 1)                                                0009
C                                                                        0 (STAT)
         IU = MINO(I+1, IM)                                              0010
C                                                                 3*JM - 3 (DO)
C
         DO 10 J = 1, JMM                                                0012
C                                                                        3 (BLOCK)
C                                                                        1 (STAT)
            TIK(J) = T(I,J,K)*B(1,I,J,K)                                 0013
C                                                                        2 (STAT)
10          T(I,J+1,K) = T(I,J+1,K)-B(2,I,J,K)*TIK(J)                    0014
C                                                                        1 (STAT)
         TIK(JM) = T(I,JM,K)*B(1,I,JM,K)                                 0015
C                                                                   8*JM (DO)
         DO 11 J = 1, JM                                                 0017
C                                                                        8 (BLOCK)
C                                                                        0 (STAT)
            T(I,J,K) = TIK(J)                                            0018
C                                                                        2 (STAT)
            T(IU,J,K) = T(IU,J,K)-B(4,I,J,K)*TIK(J)                      0019
C                                                                        2 (STAT)
            T(IL,J,KU) = T(IL,J,KU)-B(6,I,J,K)*TIK(J)                    0020
C                                                                        2 (STAT)
            T(I,J,KU) = T(I,J,KU)-B(8,I,J,K)*TIK(J)                      0021
C                                                                        2 (STAT)
11          T(IU,J,KU) = T(IU,J,KU)-B(10,I,J,K)*TIK(J)                   0022
C                                                                 8*JM - 8 (DO)
         DO 12 J = 1, JMM                                                0024
C                                                                        8 (BLOCK)
C                                                                        2 (STAT)
            T(IU,J,K) = T(IU,J,K)-B(3,I,J+1,K)*TIK(J+1)                  0025
C                                                                        2 (STAT)
            T(I,J,KU) = T(I,J,KU)-B(7,I,J+1,K)*TIK(J+1)                  0026
C                                                                        2 (STAT)
            T(IL,J,KU) = T(IL,J,KU)-B(11,I,J+1,K)*TIK(J+1)               0027
C                                                                        2 (STAT)
12          T(IU,J,KU) = T(IU,J,KU)-B(13,I,J+1,K)*TIK(J+1)               0028
C                                                                 8*JM - 8 (DO)
```

```
      DO 13 J = 2, JM                                                    0030
C                                                                      8 (BLOCK)
C                                                                      2 (STAT)
         T(IU,J,K) = T(IU,J,K)-B(5,I,J-1,K)*TIK(J-1)                     0031
C                                                                      2 (STAT)
         T(I,J,KU) = T(I,J,KU)-B(9,I,J-1,K)*TIK(J-1)                    0032
C                                                                      2 (STAT)
         T(IL,J,KU) = T(IL,J,KU)-B(12,I,J-1,K)*TIK(J-1)                 0033
C                                                                      2 (STAT)
13       T(IU,J,KU) = T(IU,J,KU)-B(14,I,J-1,K)*TIK(J-1)                 0034
C                                                                      0 (STAT)
1     CONTINUE                                                         0035
C                                                                      0 (STAT)
C
100   CONTINUE                                                         0036
C                                                                      0 (STAT)
      RETURN
      END
```

The complexity program output is exactly the same as the result obtained by hand.

Chapter 5

Future work and Conclusion

5.1 Machine model – Cost table

It is necessary for the performance evaluator to be aware of the relative amount of time required by different operations. For instance on the IBM RISC System/6000 computer, handfuls of floating-point multiplication and additions can be performed in the time it takes to do a single fixed-point multiply or to service a cache miss or even to get the result of a comparison to the branch unit. In many instances, compilers cannot exploit these facts. So we need several tables to present the machine's characteristics.

Because the arithmatic calculation is very machine-dependent, so the factor depends heavily upon which machine it runs. We should fetch the factors at the first time when PIPS is installed.

5.2 Implementation – Half static and half dynamic

Both static and dynamic evaluations have the shortcomings that we can't solve it separately. We have to find a way to combine the advantages of the two evaluations and get rid of the drawbacks as most as possible. We think that method of half static and half dynamic is a good way to evaluate the performance of program. The aim of this method is to get rid of static evaluation problems, while keeping advantages of the symbolic expression of complexity. This method runs in three steps.

1. It begins with a first pass of static evaluation, accumulating informations about its failures: the locations of the IF tests which probability could not be computed (almost all), and the locations of the DO loops which ranges were not exactly computed. These are the only counters needed to complete the stastic complexity evaluation.

2. Then it makes a copy of the program, inserting counters at all places where a failure occured during the first pass. The second step is to execute the modified program. At the end of the run, the counters are written in a file that will now be exploited by a second pass of "static" evaluation.

3. The last step is to rerun the "static" evaluation with all the counters obtained by the second step to get the overall complexity of the program.

The only use of the first pass of complexity evaluation is to insert counters only where it is necessary, to gain time on the execution of the modified program. Actually, it may be more interesting to skip it and choose to spy every IF test probability and every DO loop range width.

An advantage of this method is that the profiling execution can run on any machine (once again, having the same floating-point numbers internal representation). As it uses much less profiling information than the dynamic approach, and because the output evaluation is parametric, its results are less sensitive to the choice of the data set provided for this particular sample run.

Bibliography

[ASU86] A.V. Aho, R. Sethi, J.D. Ullman, *"Compilers: Principles, Techniques, and Tools"*, Addison-Wesly, 1986.

[Baron] B.Baron *PIPS User Guild Version 1.0*, Sep. 1990

[Ber66] A.J. Berstein, *Analysis of programs for parallel processing"*, IEEE Transactions on Electronic Computers, Vol.15, No 5, OCT66.

[Bert] **P. Berthomier *Static Comparison of Different Program Versions*, DEA systèmes Informatiques - Université PARIS VI, Parallélisation et Vectorisation Automatiques.** Sep. 1990 Document EMP-CAI-I 130

[BKV] J. Bentley, B. Kernighan, C. Van Wyk *"An Elementary C Cost Model"* UNIX Review, Feb. 1991

[BWJ90] F. Bodin, D. Windheiser, W. Jalby, D. Atapattu, M. Lee, D. Gannon, *"Performance Evaluation and Prediction for Parallel Algorithms on the BBN GP100"*, Proceedings of the ACM International Conference on Supercomputing, Amsterdam, JUN90.

[DOZ90] H.G. Dietz, M.T. O'Keefe, Abderrazek Zaafrani. *"An introduction to Static Scheduling for MIMD Architecture"*, Proceedings of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, California, AUG90.

[Emad] Nahid Emad *"Détection de Parallélism et Production de Programmes Parallèles* CRI report Jan. 1991 Document EMP-CAI-I 150

[Fato90] R.A. Fatoohi, *"Vector Performance Analysis of the NEC SX-2"*, Proceedings of the ACM International Conference on Supercomputing, Amsterdam, JUN90.

[IAM87] I. Ion, R. Arhire, M. Macesann, *"Program complexity: comparative analysis, hierarchy, classification"*, SIGPLAN NOTICES, Vol.22, No 4, AVR87.

[IJ91] F. Irigoin, P. Jouvelot *PROJET PIPS: Manuel Utilisateur du Paralléliseur (version 2.3)* Centre d'Automatique et d'Informatique - Section Informatique: Rapport interne, 1991. Document EMP-CAI-I E144

[IJT90] F. Irigoin, P. Jouvelot, R. Triolet, *"PIPS: Représentation intermédiaire"*, Centre d'Automatique et d'Informatique - Section Informatique: Rapport interne, 1990.

[JT89] P. Jouvelot, R. Triolet, *"NewGen: A Language-Independant Program Generator"*, Rapport interne EMP-CAI-I A/191, Mines de Paris, JUL89.

[Meta90] Daniel Le Metayer, *"ACE: An Automatic Complexity Evaluator"*, ACM TRANSACTIONS on Programming Languages and Systems, Vol.10, No 2, AVR90.

[MR90] A,D. Malony, D.A. Reed, *"Hardware-Based Performance Monitor for the Intel iPSC/2 Hypercube"*, Proceedings of the ACM International Conference on Supercomputing, Amsterdam, JUN90.

[Nich90] K.M. Nichols, *"Performance Tools"*, IEEE Software, MAY90.

[PIPS] Rapports DRET No 103 à 128, CAII, Fontainebleau.

[Port89] A.K. Porterfield, *"Software Methods for Improvement of Cache Performance on Supercomputer Applications"*, Rice University, Computer Science Technical Report, Rice COMP TR89-93, MAY89.

[Sark89] Vivek Sarkar. *"Determining Average Program Execution Times and their variance"*, ACM SigPlan P.L.D.I, Portland 1989.

[Tawbi90] Nadia Tawbi, *Parallélisation Automatique : Estimation des Durées d'Exécution et Allocation Statique de Processeurs*, Thèse à l'université Paris VI. e-mail: nt@dpx-nt.crg.bull.fr.

[Zima] H. Zima and B. Chapman *Supercompilers for Parallel nad Vector Computers* ACM Press, Frontier Series 1991