CENTRE DE RECHERCHE EN INFORMATIQUE

Complexity Estimation in the PIPS Parallel Programming Environment

Lei Zhou

February 1992

Document EMP–CRI A/221

Complexity Estimation in the PIPS Parallel Programming Environment

Lei Zhou
Centre de Recherche en Informatique
Ecole Nationale Supérireure des Mines de Paris

E-mail: zhou@cri.ensmp.fr

February 1992

Abstract

In order to choose the 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 real scientific Fortran programs can run for hours on expensive machines, it's useful to perform a static analysis that accurately predicts running time.

In this paper, we propose to use our complexity model, which is composed of a library of polynomial models of program performance and dynamically-derived program statistics, to present running time predictions of Fortran programs. Then, a real example from the PERFECT Club is presented in the PIPS programming environment.

1 Introduction

PIPS[1] is a source-to-source parallelizing compiler that transforms Fortran77 programs by replacing parallelizable nests of sequential DO loops with either Fortran90 vector instructions or DOALL constructs. It is not targeted towards any particular supercomputer, although only shared memory machines have been considered. The principal characteristics of PIPS are:

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

See [IJT91] for more information.

One part of the PIPS project is a performance tool to predict the complexity of a program. This tool can be based on a variety of machine models which will allow us to predict the performance of a program on different machines. There are several methods we can use to predict the performance of a given algorithm or Fortran program.

1. The easiest way is to run it on the real parallel machine such as a Cray, Transputer or CM-5, with varying number of processors and varying problem sizes. This is not the best way, not only because it's the most expensive one, but also we can not see the relationship between the result and parameters.

[1]PIPS is French acronym which stands for Paralléliseur Interprocédural de Programmes Scientifiques, developed at Centre de Recherche en Informatique (CRI) de l'Ecole Nationale Supérieure des Mines de Paris (ENSMP).

2. The cheapest one is static evaluation, but it is not computable. As there is no way to predict the outcome of a branch statement in static evaluation, we are forced to guess or choose forced probabilities.

3. The most appropriate method should be a combination of the former two approachs to get the best tradeoff. We call it "half static and half dynamic". This method allows us to get around static evaluation problems, while keeping the advantages of symbolic expressions of complexity. The advantage of this method is that the profiling execution can run on any machine.

Our goal is the "half-static and half-dynamic" model. For the time being, only the first step has been done, that is, only a pure static evaluator has been implemented and only sequential Fortran programs are taken into account.

In order to calculate how many times a loop body should be executed, the values of the loop bound and increment are needed also. Hence, we have to make full use of preconditions on scalar integer variables. If we fail to get exact values of loop bounds, we use the largest complexity instead. We will get UNKNOWN_RANGE when no information is available. This is described in [Bert].

2 Complexity Components

The complexity tool of the PIPS project is composed of two elements: a library of polynomial mathematical models and counters which measure statistical information about each variable.

2.1 Polynomial

In our model, the complexity of every statement of Fortran program can be expressed as a polynomial. To support this, a polynomial library has been created by several people at CRI. It is the component of the C3 library. It is made from monomes, which are made from vectors. The vector library is the essential part of the C3 library. It has the same features as mathematical polynomial. There is only a restriction on division, the divisor must be a monome or a constant. Both developers and users are interested in this information.

2.2 Statistics

Although it is 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 information measured during the evaluation. This kind of information is reserved for further development of this estimator.

2.2.1 Variable Statistics

1. **symbolic :** counts variables that appear explicitly and do not need to be evaluated. What we need to know is its type.

2. **guessed :** counts variables that must be evaluated and whose value can be calculated.

3. **bounded :** counts variables that must be evaluated and whose value can not be calculated exactly. In this case, we chose the worst case estimation.

4. **unknown :** counts variables which are totally unknown.

2.2.2 Range Statistics

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

2.2.3 If Statistics

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.

3 Target machine model

Each computer manufacturer has its own way to deal with 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.

3.1 Cost Table

In our complexity model, we divide the entire cost table into five separate parts, each is stored under the COMPLEXITY_COST_DIR directory, namely operation, index, memory, transcend and trigo.

Initially, we will test our system on a simple machine model. Later more accurate estimate of instruction will be used.

Here we present the *all-one* tables for *operation*, which is the ideal complexity of the basic operation. It is first step towards the real world.

```
# operation  cost for basic operations

#       int     float   double  complex double-complex

+        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
```

In addition, we use similar cost tables for memory access, array element address computation and transcendental functions which have the same format as the one above.

3.2 Machine Models Implemented

The simplest model has been chosen for a prototype implementation. The machine has one ALU and one FPU, and they do not overlap. There is neither cache memory, nor a virtual memory system. For the time being, the evaluator only processes sequential programs. Vector processing is not taken into account. File inputs/outputs are modelled as a static constant cost, as one Fortran intrinsic operator (static in the sense that it doesn't depend on the environment in the source code). More accurate estimates of these costs will be introduced later. In the case of mathematical functions, a cost is defined for each type of argument: integer, floating-point, double precision, complex numbers.

4 Complexity Algorithms

We initially started the evaluation of the complexity in terms of floating-point operations; then it gradually appeared that integer operations can be equally important. Furthermore, memory accesses can also be a real bottleneck. Hence, all of these are included in our analysis. Below we describe how the polynomial library, the dynamically determined program statistics and the machine model are combined to determine the complexity of a program.

4.1 Complexity of a variable

The cost of a variable depends upon its type. For each variable we can find the cost in the table of *memory* cost.

4.2 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 performed. The cost of operation can then be derived from the *operation* cost table.

4.3 Complexity of an expression

The evaluation of the cost of an expression is complicated by the use of overloading for 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 leaves of the syntactic tree (constants, variables or function calls) where the type is known. The types of sub-expressions are propagated towards the root. For example, $IL * JL$ will have the cost 3 according to our current *all-one* cost table, one for each variable and one for the multiplication.

4.4 Complexity of a statement

We have different complexity for different types of statements.

4.4.1 Complexity of an assignment

In PIPS, the assignment operation is considered as a call. It can be viewed as *expression* = *expression*, so the total complexity equals the sum of the two expressions. There is no explicit charge for the assignment operation because the cost is already associated with the memory access to the variable at the left-hand side. For example, the following assignment has the the cost five, three variables and two operators.

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

4.4.2 Complexity of a Call

The complexity of a call is exactly the total complexity of the corresponding subroutine or function. There is no extra charge for a call.

4.4.3 Complexity of a Structured IF

$$\text{IF } \mathit{boolexpr} \text{ THEN } \mathit{stat}_{true} \text{ ELSE } \mathit{stat}_{false} \text{ ENDIF}$$

This statement is structured if there is no GOTO jumping into or out of $stat_{true}$ or $stat_{false}$. Let us call p the probability that *boolexpr* is true and 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 the branch probability cannot be determined at compile-time, as a first approximation, prior to run-time measures, we use the values $p = q = \frac{1}{2}$.

4.4.4 Complexity of a Sequential DO

Suppose that we have a loop:

```
DO index = lower, upper, increment
    body
ENDDO
```

The evaluation of *lower*, *upper*, *increment* may involve function calls, whose complexity must be added to the overall execution time. The complexity of the body may depend on the index, so we must integrate it over 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. Ideally, we consider that C_{body} is the sum of complexity of loop body and complexity unit one.

$$C(DO...) = C_{lower} + C_{upper} + C_{increment} + \sum_{index=lower}^{upper} C_{body} \tag{1}$$

This formula applies if we are able to properly evaluate *lower*, *upper* and *increment* as polynomials. The cost of the upper and lower bounds is the cost of evaluating each expression. Our experience shows $C_{increment}$ is negligible because of register operation.

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*.

4.4.5 Complexity of a Parallel DO

```
DOALL index = lower, upper
    body
ENDDO
```

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

$$C(DOALL...) = C_{lower} + C_{upper} + \max_{lower \leq index \leq upper} C_{body} \tag{2}$$

This maximum is impossible to find in the general case. But note that the best performance is obtained when all iterations have the same duration, avoiding load imbalance. So when we are unable to compute the maximum, we can approximate it with the complexity of the first iteration.

4.5 Complexity of a block

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

Once the control graph is computed, PIPS' internal representation of programs guarantees 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 of the statistics of its statements. PIPS detects only loop parallelism (but not `COBEGIN ... COEND` or `FORK, JOIN`).

4.6 Complexity of an unstructured control flow graph

Any Fortran program, even those containing GOTOs, can be represented by a graph whose nodes are structured or elementary instructions, and whose edges stand for jumps. Most 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 ; PIPS' internal representation is such a graph. Moreover, small unstructured code fragments 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.

Let us now define the complexity of an unstructured program represented by its control flow graph. Let $\{S_1, S_2, \dots, S_n\}$ be the set of the nodes, where S_1 is 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 GOTOs reaching from outside of the graph into the middle of it.

Let $c_1, c_2, \dots, c_n$ be the complexities associated with $S_1, S_2, \dots, S_n$, and supposed known. Let us call p_{ij} the probability of going to node S_j when you are in node S_i. The probability of going to some nodes, from node S_i, is 1, Finally, we require that $\sum_{j=1}^{n} p_{ij} = 1$. We'll at last associate with 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, since 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 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}.

$$g_i = c_i + \sum_{S_j \text{ successor of } S_i} p_{ij} \cdot g_j \qquad (3)$$

5 Current Status

We have said that, for the time being, only a pure static evaluator has been implemented and only sequential Fortran programs are taken into account. In this section, we present the current implementation and describe its important features.

5.1 Cost Table

We provide a way to choose the cost table desired, or to build a brand-new one if needed. For example, if you only want floating-point operations to be accounted for, you can provide a new

COMPLEXITY_COST_DIR, which is stored in the file *properties.rc* of the working directory. An *operation* table of our *fp-one* model might look like this:

```
# operation   - Unity cost for floating-point

#       int     float  double  complex double-complex

+        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
```

5.2 Unevaluated Variables

Sometimes, we need to keep some variables unevaluated, to let them appear in the complexity output. These variables may be specified by the user. If the user doesn't want several variables to be evaluated, he can put these variable names into COMPLEXITY_PARAMETERS which is stored in the file *properties.rc* of the working directory.

In addition, PIPS may identify variables which must remain unevaluated. If COMMON variables appear in the module, that means they are assigned somewhere else. These variables should be kept in the module's complexity result.

5.3 Preconditions

PIPS uses preconditions, that is information available before the statement, to resolve analysis problems. All the preconditions depend heavily upon the C3 libraries, especially upon the system constraint library where we can get all the information we want about the preconditions. One major source of precondition information is constants. An example can be seen in the following program:

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

In this loop, n is replaced automatically by 10. However, it is important to make sure that the preconditions are up to date. Consider the following 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 two complexity results can not be added.

5.4 Simplifing Complexity Results

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 *intermediate* variable(s), which are local to the subroutine. Look at the following example:

```
        subroutine sub1(a,n)
        real a(1000)
        integer m,n
        k = 3 * n + 2
        do 10 i = 1, k
           a(i) = a(i) + 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. We call such variables *intermediate* variables.

We'll point out that no matter how complex the function is , as long as k is a linear function of n, the final complexity result always contains n as its component. Let us look at a more complex 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 embedded loops here loop 10, loop 20 and loop 30 respectively. For the innermost loop, its lower bound `jj + 10` is dependent on the index of loop 20, which is determined by the two indices of the outermost loop. Remember that we use a bottom-up method to accumulate the complexity. Confined in this innermost loop, we can not know the relation between this loop and outer loop. So we must obtain what we need in the innermost loop. Naturally preconditions come 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                                               0002
C                                                                3 (STAT)
         II = I+1                                                  0003
C                           9*I^2 - 12*I.M + 3*M^2 - 84*I + 72*M + 135 (DO)
         DO 20 J = II, M+2                                         0005
```

```
C                                                                5 (STAT)
          JJ = I+J-2                                                  0006
C                                           -6*I - 6*J + 6*M + 70 (DO)
          DO 30 K = JJ+2, M+10                                        0008
C                                                                3 (STAT)
             T = T+1.0                                                0009
C                                                                3 (STAT)
             U = U+1.0                                                0010
C                                                                0 (STAT)
30           CONTINUE                                                 0011
C                                                                0 (STAT)
20        CONTINUE                                                    0012
C                                                                0 (STAT)
10     CONTINUE                                                       0013
C                                                                0 (STAT)
      RETURN
      END
```

Our method is to put each induction variable into a hash table when encountering a loop, and delete that induction variable when leaving the loop. This allows us to save a copy of the variable, which has not been evaluated by the complexity program. So for the loop 30, the complexity information contains only the outer loops' induction variables `i`, `j` and formal parameter `m`. For loop 20, the complexity result is given in terms of `i` and `m`, for loop 10, the result contains merely the formal parameter `m`.

According to our complexity algorithm, shown in (1) of Section 4.4.4, the total complexity for the loop is `2 + 2 + 0 + 6*( (M+10) - (JJ+2) + 1 )` which equals `-6*I - 6*J + 6*M + 70` when substituting JJ with the expression `JJ = I+J-2` . Here 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.) and the same thing for the expression `M+10`.

6 A Real Example

In this section, we execute our complexity program on a real Fortran program. We chose a medium-sized one *TFS.f* from the PERFECT Club. See [Cybenko] for more information. It has almost 2000 lines and its application area is fluid dynamics. In order to focus on the main functions, we have gotten rid of the irrelevant routines, such as plotting a graph and timing. This leaves 27 routines, and we succeeded in evaluating 23 of them. The benchmark was evaluated using both the *all-one* and *fp-one* models. The results are shown in Table 1 and 2 respectively.

IL and JL are COMMON variables, so we must keep them in the complexity results. I2, J2, II2, JJ2, LPRNT and NRES are formal parameters passed by the coresponding callers. In the routine ADDX, U_RANGE which is short for UNKNOWN_RANGE appears. This is due to the variable J1, which serves as upper loop bound. It depends on the input, so we can not know it at compile-time.

We choose one routine COLLC to present the entire complexity output using *all-one* cost table:

```
C                    60*I2.J2 + 94*IL.JL + 62*IL + 4*J2 + 56*JL + 103 (SUMMARY)
C     NAME=COLLC
      SUBROUTINE  COLLC (I2,J2,W,VOL,II2,JJ2,WW,WR)
C     COLLECTS RESIDUALS FOR THE NEXT MESH
      COMMON/LIM/ IL,JL
```

Table 1: Complexity Results Using *all-one*

Module	Complexity
ADDX	159*I2.J2+32*II2.JJ2+95*IL.U_RANGE+158*IL+11*J2+2*JJ2-4*U_RANGE+22
BCFAR	240*IL-101
BCWALL	253*IL-113
CIRCLE	21*IL.JL+IL+2
COLLC	60*I2.J2+94*IL.JL+62*IL+4*J2+56*JL+103
COORD	13*IL+16*JL+7
CPLOT	36*IL+308
DFLUX	995*IL.JL-775*IL-553*JL+387
DFLUXC	459*IL.JL-326*IL-305*JL+197
EFLUX	490*IL.JL-445*IL-315*JL+281
FORCF	87*IL-65
GRAPH	31*IL+32
GRID	32*IL.JL+14*IL+23*JL+14
INIT	33*I2.J2+J2+1
INTPL	458.93
MESH	230
METRIC	52*IL.JL+14*I2-52*IL-37*JL+39
PRNTFF	93*I2.J2.LPRNT^(-1)+186*I2-91*I2.LPRNT^(-1) -93*J2.LPRNT^(-1)+91*LPRNT^(-1)
PRNTXY	17*IL.JL.LPRNT^(-1)+2*IL.LPRNT^(-1)+4
PSMOO	400*IL.JL-416*IL-416*JL+478
RPLOT	37*NRES+51
STEP	244*IL.JL+28*I2-244*IL-213*JL+222
XPAND	5*IL.JL+5*IL+7*JL+9

Table 2: Complexity Results Using *fp-one*

Module	Complexity
ADDX	28*I2.J2+2*II2.JJ2+12*IL.U_RANGE+43*IL-12*U_RANGE - 3/2
BCFAR	36*IL + 25*IL - 36
BCWALL	55*IL - 38
CIRCLE	5*IL.JL
COLLC	4*I2.J2 + 10*IL.JL + 6*IL
COORD	4*IL + 5*JL + 4
CPLOT	8*IL
DFLUX	106*IL.JL - 96*IL - 83*JL + 78
DFLUXC	48*IL.JL - 34*IL - 34*JL + 23
EFLUX	70*IL.JL - 64*IL - 41*JL + 35
FORCF	25*IL - 19
GRAPH	8*IL + 8
GRID	8*IL.JL + 2*IL + 4*JL
INIT	4*I2.J2
INTPL	88
MESH	80
METRIC	8*IL.JL - 8*IL - 8*JL + 8
PRNTFF	17*I2.J2.LPRNT^(-1)+34*I2-17*I2.LPRNT^(-1) -17*J2.LPRNT^(-1)+17*LPRNT^(-1)
PRNTXY	0
PSMOO	48*IL.JL - 56*IL - 56*JL + 78
RPLOT	13*NRES + 8
STEP	48*IL.JL - 48*IL - 48*JL + 48
XPAND	0

```
      COMMON/ADD/ DW(194,34,4)
      COMMON/MGR/ KODE,MODE
      DIMENSION   W(I2,J2,4),VOL(I2,J2),WW(II2,JJ2,4),WR(II2,JJ2,4)
C                    60*I2.J2 + 94*IL.JL + 62*IL + 4*J2 + 56*JL + 103 (UNSTR)
C                                                                   3 (STAT)
      IIL = II2-1                                                    0001
C                                             35*IL.JL + 10*JL + 8 (DO)
      DO 10 N = 1, 4                                                 0003
C                                                                   1 (STAT)
         JJ = 1                                                      0004
C                                             9*IL.JL + 5/2*JL + 1 (DO)
         DO 10 J = 2, JL, 2                                          0006
C                                                                   3 (STAT)
            JJ = JJ+1                                                0007
C                                                                   1 (STAT)
            II = 1                                                   0008
C                                                         18*IL + 1 (DO)
            DO 10 I = 2, IL, 2                                       0010
C                                                                   3 (STAT)
               II = II+1                                             0011
C                                                                  32 (STAT)
               WR(II,JJ,N) = DW(I,J,N)+DW(I+1,J,N)+DW(I,J+1,N)+DW(I+1   0012
     &         ,J+1,N)                                               0012
C                                                                   0 (STAT)
10             CONTINUE                                              0013
C                                              60*I2.J2 + 4*J2 + 4 (DO)
      DO 20 N = 1, 4                                                 0015
C                                                15*I2.J2 + J2 + 1 (DO)
         DO 20 J = 1, J2                                             0017
C                                                         15*I2 + 1 (DO)
            DO 20 I = 1, I2                                          0019
C                                                                  15 (STAT)
               DW(I,J,N) = VOL(I,J)*W(I,J,N)                         0020
C                                                                   0 (STAT)
20             CONTINUE                                              0021
C                                             59*IL.JL + 46*JL + 8 (DO)
      DO 30 N = 1, 4                                                 0023
C                                                                   1 (STAT)
         JJ = 1                                                      0024
C                                             15*IL.JL + 12*JL + 1 (DO)
         DO 30 J = 2, JL, 2                                          0026
C                                                                   3 (STAT)
            JJ = JJ+1                                                0027
C                                                                   1 (STAT)
            II = 1                                                   0028
C                                                         30*IL + 1 (DO)
            DO 25 I = 2, IL, 2                                       0030
C                                                                   3 (STAT)
               II = II+1                                             0031
C                                                                  56 (STAT)
               WW(II,JJ,N) = (DW(I,J,N)+DW(I+1,J,N)+DW(I,J+1,N)+DW(I+   0032
     &         1,J+1,N))/(VOL(I,J)+VOL(I+1,J)+VOL(I,J+1)+VOL(I+1,J+1)   0032
     &          )                                                          0032
C                                                                       0 (STAT)
25              CONTINUE                                                   0033
C                                                                       9 (STAT)
            WW(1,JJ,N) = WW(IIL,JJ,N)                                      0034
C                                                                       9 (STAT)
            WW(II2,JJ,N) = WW(2,JJ,N)                                      0035
C                                                                       0 (STAT)
30          CONTINUE                                                       0036
C                                                              62*IL + 80 (DO)
      DO 40 N = 1, 4                                                       0038
C                                                                       1 (STAT)
         II = 1                                                            0039
C                                                               16*IL + 1 (DO)
         DO 35 I = 2, IL, 2                                                0041
C                                                                       3 (STAT)
            II = II+1                                                      0042
C                                                                      28 (STAT)
            WW(II,JJ2,N) = (DW(I,J2,N)+DW(I+1,J2,N))/(VOL(I,J2)+VOL(I   0043
     &      +1,J2))                                                        0043
C                                                                       0 (STAT)
35          CONTINUE                                                       0044
C                                                                       9 (STAT)
         WW(1,JJ2,N) = WW(IIL,JJ2,N)                                       0045
C                                                                       9 (STAT)
         WW(II2,JJ2,N) = WW(2,JJ2,N)                                       0046
C                                                                       0 (STAT)
40       CONTINUE                                                          0047
C                                                                       0 (STAT)
      RETURN
      END
```

7 Conclusion

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.

Neither static nor dynamic evaluation is sufficiently powerful to solve performance predication. We have to find a way to combine the advantages of the two methods of evaluation and get rid of the drawbacks as much as possible. We think that the method of half-static and half-dynamic evaluation is a good way to evaluate the performance of a program. The aim of this method is to get rid of static evaluation problems, while keeping the advantages of the symbolic expression of complexity. This method runs in three steps.

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

2. In the second step, dynamic analysis is used to address these shortcomings. A copy of the program is made, inserting counters at all places where a failure occurred during the first pass. The modified program is executed. 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.

An advantage of this method is that the profiling execution can run on any machine (once again, having the same internal floating-point 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.

In this paper we have shown how a library of polynomial models and a machine description can be used to construct the static analysis tool described in step one. This is also the foundation for step three. Our simple model allowed us to show that this approach works. In the future, we plan to combine this tool with the dynamic analysis described step two.

Furthermore, 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 measure every IF-test probability and every DO-loop range width.

References

[Bala] V. Balasundaram et al. *A Static Performance Estimator to Guide Data Partitioning Decisions* Rice U. COMP TR90-136, Oct. 1990

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

[Bert] P. Berthomier *Static Comparison of Different Program Versions*, DEA systèmes Informatiques - Université PARIS VI, Parallélisation et Vectorisation Automatiques. Sep. 1990 Report EMP-CAI-I 130, Sep. 90

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

[CKPK] G.Cybenko, L. Kipp, L. Pointer and D. Kuck *Supercomputer Performance Evaluation and the Perfect Benchmark*, CSRD Rpt. No. 965

[Cybenko] G. Cybenko *Supercomputer Performance Trends and thee Perfect Benchmark* Supercomputing Review, April 1991

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

[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. Report EMP-CAI-I E144

[IJT91] F. Irigoin, P. Jouvelot, R. Triolet, *"PIPS overview"*, ICS'91

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

[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.

[Tawbi90] Nadia Tawbi, *Parallélisation Automatique : Estimation des Durées d'Exécution et Allocation Statique de Processeurs*, Thèse à l'université Paris VI. April 1991

[Zhou] Lei Zhou *Enhanced Static Evaluation of Fortran Programm in PIPS Environment* Report EMP-CRI E/160/CRI Dec. 1991

Luc Bouge

CONPAR 92
VAPP V

The past decade has seen the emergence of two highly successful series of CONPAR and of VAPP conferences on the subject of parallel processing.
The Vector and Parallel Processors in Computational Science meetings were held in Chester (VAPP I, 1981), Oxford (VAPP II, 1984) and Liverpool (VAPP III, 1987). The International Conferences on Parallel Processing took place in Nürnberg (CONPAR 81), Aachen (CONPAR 86) and Manchester (CONPAR 88). Thereafter the two series joined and the CONPAR 90 - VAPP IV conference was organised in Zurich.

The next event in the series CONPAR 92 - VAPP V will be organised in 1992 at the Ecole Normale Superieure de LYON (FRANCE) from September 1, 1992 to September 4, 1992.

The format of the joint meeting will follow the pattern set by its predecessors. It is intended to review hardware and architecture developments together with languages and software tools for supporting parallel processing and to highlight advances in models, algorithms and applications software on vector and parallel architectures.

<u>It is expected that the program will cover:</u>

✱ languages / software tools
✱ automatic parallelization and mapping
✱ hardware / architecture
✱ performance analysis
✱ algorithms
✱ applications
✱ models / semantics
✱ paradigms for concurrency
✱ testing and debugging
✱ portability

Final Call for Papers

The proceedings will be published by Springer Verlag in the Lecture Notes in Computer Science Serie.
Original papers are invited for the conference. Five copies of the full paper (maximum of 15 pages) are to be submitted no later than Feb 29, 1992 to :

Secretariat CONPAR92/VAPPV

Ecole Normale Supérieure de Lyon
Laboratoire de l'Informatique du Parallélisme
46, allée d'Italie - 69364 Lyon Cedex 07 - France

+33/72/72/80/37 - fax +33/72/72/80/80
e-mail conpar92@frensl61.bitnet

Schedule

Return Form of Intent	Now please !
Submission of paper	before Feb 29,1992
Notification of acceptance	May 15,1992
Final Program	June 15,1992

Awards

❶ Best technical contribution
❷ Best student contribution
❸ Best presentation
❹ Best poster

C.FRABOUL	(Toulouse - France)
D.GANNON	(Bloomington - USA)
J.L.GAUDIOT	(L.Angeles - USA)
A.GERASOULIS	(Rutgers Univ. - USA)
W.GILOI	(TU Berlin - FRG)
G.HAINS	(Montréal - Canada)
R.N.IBBETT	(Edinburgh - UK)
H.IMAI	(Tokyo - Japan)
C.JESSHOPE	(Surrey - UK)
H. JORDAN	(Colorado - USA)
O.LANGE	(TU Hamburg-Harburg - FRG)
M.LANGSTON	(Tenessee - USA)
N.N.MIRENKOV	(Acad. of Scien. USSR)
H.MULHENBEIN	(Augustin I - FRG)
Y.MURAOKA	(Tokyo - Japan)
P.NAVAUX	(Porto Allegre - Brasil)
D.A. PADUA	(Illinois - USA)
D. PARKINSON	(Queen Mary College - UK)
B.PHILIPPE	(IRISA Rennes - France)
R.H. PERROTT	(Belfast - UK)
G.R. PERRIN	(Besancon - France),
B.PLATEAU	(Grenoble - France)
R.PUIGJANER	(Baleares - Spain)
P.QUINTON	(Rennes - France)
V.RAMACHANDRAN	(Austin - USA)
K.D.REINARTZ	(Erlangen-Nürnberg FRG)
G.REIJNS	(Delft - Netherlands)
Y.ROBERT	(ENS Lyon - France)
D.ROOSE	(Leuven - Belgium)
W.RYTTER	(Warszawski - Poland)
S.SEDUKHIN	(Novosibirsk - USSR)
B.SENDOV	(Sofia-Bulgaria)
S.W.SONG	(Soa-Paulo - Brasil)
O.SYKORA	(Bratislava - CSFR)
M.TCHUENTE	(Yaounde - Cameroun)
E.E.TYRTYSHNIKOV	(Moscow - USSR)
D.TRYSTRAM	(Grenoble - France)
M.VALERO	(Barcelona - Spain)
M.VANNESCHI	(Pisa - Italy)
M.VAJTERSIC	(Bratislava - CSFR)
P.VITANYI	(CWI and Amsterdam - Netherlands)
U.VISHKIN	(Maryland - USA and Tel Aviv - Israël)
R.WAIT	(Liverpool - UK)
H.P. ZIMA	(Wien - Austria)

CONPAR 92 - VAPP V

Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon, France

September 1-4, 1992

FINAL CALL FOR PAPERS

Sponsored by

BCS-PPSG, CNRS, GI-PARS, Institut IMAG, Programme de Recherches Coordonnées C^3

in cooperation with

IFIP WG10.3, IEEE, ACM, AFCET, SI-PARS, INRIA