Visualizing the Iteration Space in PEFPT *

Qi Wang, Yu Yijun and Erik D'Hollander

University of Ghent
Dept. of Electrical Engineering
St.-Pietersnieuwstraat 41
B-9000 Ghent
wang@elis.rug.ac.be
Tel: +32-9-264.33.75
Fax: +32-9-264.35.94

Abstract. Sufficient and precise semantic information is essential to interactive parallel programming. In this paper, we present a feasible implementation of the iteration space dependence graph and discuss relevant technical problems. Moreover, we give a further prospect of interactive loop optimization guided by the graph.

1 Introduction

In the past decade, high performance computing has become a critical technique for scientists and engineers. Parallel processing is considered as the essential way to speedup massive computational application, and large-scale parallel architectures composed of microprocessors become a fashion. To exploit parallelism and memory hierarchy effectively for these machines, the compiler must be able to get data dependence precisely and distribute the computations among the processors accordingly. Unfortunately, the results of recent research have been both encouraging and disappointing[3,4]. One of principal drawback is the inaccuracy of current dependence analysis techniques: symbolic expression, procedure calls, induction and reduction variables, and complex control flow, all of them introduce conservative assumption, so as to invalidate the feasibility of parallelism.

A tradeoff is to provide an interactive programming or optimizing environment, sharing part of the responsibility with the user under the guidance of the system, which the user can afford to. This compromise is accepted widely by the user and the researcher. There have been much significant experiments in this area, such as *Parafrase-2*[5] and *Parascope*[6].

1.1 Motivation

We devote ourselves to design a new programming environment – *PEFPT*[7] [2] (the Parallel Programming Environment for *FPT*, the Fortran Parallel Transformer), based on the research of *FPT*[7], proposing to develop and integrate

* This work was supported in part by European Community under grant ITDC'94-164 and by the Ministry of Education, project CHIN9504

[2] A joint project between the universities of *Ghent*(B) and *Fudan*(PRC)

series of meaningful tools and methods, and testing them through practice.

It is well known that most of the parallelism in a program dwells in loops, and current parallel architectures also pay great emphasis on developing coarse-grained parallelism.

Therefore, it is vital to exhibit loop-carried dependence information. *Parascope* provides a table to show the dependencies between statements in different loops. This allows the user to inspect and potentially remove false dependencies. However, a table is not very elegant to analyze the parallelism on iteration level. In *PEFPT*, we employ an *iteration space dependence graph* as a supplement illustration for the user.

1.2 Data dependence

The analysis of precedence constraints on the execution of the statements is a fundamental step in parallelizing the program.

There are four types of data dependence [1,2] between two statements, S_1 and S_2:

True (flow) dependence occurs when S_1 writes a memory location that S_2 later reads, and there is no S_3 write this location between S_1 and S_2.

Anti dependence occurs when S_1 reads a memory location that S_2 later writes, and there is no S_3 write this location between S_1 and S_2.

Output dependence occurs when S_1 writes a memory location that S_2 later writes again, and there is no S_3 write this location between S_1 and S_2.

Input dependence occurs when S_1 reads a memory location that S_2 later reads, and there is no S_3 write this location between S_1 and S_2.

1.3 Iteration space and Iteration space dependence graph

Suppose $I \subseteq R^n$, $i_1, \ldots, i_n$ are the iteration indices, $(L_1, U_1), \ldots, (L_n, U_n)$ are the respective loop bounds[1,2] and both L_l and U_l are linear functions of iteration indices $i_1, \ldots, i_{l-1}$, so that the iteration space is:

$$I = \{(i_1, \ldots, i_n) | L_1 \leq i_1 \leq U_1, \ldots, L_n(i_1, \ldots, i_{n-1}) \leq i_n \leq U_n(i_1, \ldots, i_{n-1})\}$$

Moreover, the *iteration space dependence graph* of the loops is an iteration space picture with dependence arrows, which draw from the point corresponding to iteration $I = (i_1, \ldots, i_n)$ to the point corresponding to iteration $J = (j_1, \ldots, j_n)$ whenever there exist statements $S_1(I)$ and $S_2(J)$ in the loop body such that S_1 dependent on S_2, where $I \neq J$.

2 Analysis of the iteration space dependence graph

A straight solution to get the dependence relationship between iterations is to record all variable reference information occurred during the execution of the loops, then to analyze them using the definition of dependence. For each *read operation*, the iteration space is searched from this point until a write operation to the same memory location is found. If read and write are not in the same iteration, a dependence results. It is a flow dependence if the write comes lexicographically before the read, otherwise it is an anti-dependence.

Output dependence occurs when a write operation is followed lexicographically by another write operation.

2.1 Static analysis of the iteration space

At first, we implement such an idea in *PEFPT* using a strategy of static simulated execution on the syntax tree. The user selects particular loops and gives necessary data (especially the values of symbolic variables). Throught calculating all loop indices and array subscripts for each iteration on the lexicographic order, we get a real iteration space with all reference information of variables. Then the technique described above is used to get all dependence arrows between iterations. In order to handle complex control flow, especially GOTO statements, a powerful algorithm [7] exists within FPT, which converts GOTO statements into structured code.

Sometimes nested loops have large iteration distances, and a loop body may be very complex. In that case, the symbolic execution is time consuming and becomes impossible when there are call statements in the loop body.

2.2 Runtime simulation

In order to solve these limits and deal with more general cases, we propose to insert a marker for each variable reference in the source program. This marker will store the read/write behavior during the execution. After running the program, the required data are available.

The question is how to implement the marker to process the raw data quickly at runtime, and to minimize the information records needed by the post-processor.

For each analysis variable, two integer shadow variables are declared with the same dimension as the original: RS (read shadow) and WS (write shadow), each element of them records the iteration ID which occurs last read/write operation on this element. See the tracer algorithm in figure 1.

2.3 Generation of the test program

In order to generate the dependence test program automatically, we need do more work and more information. We divide all process into five steps:

Initialization: For each variable A, declare two shadow variables: RsA and WsA, and set RsA($i_1, \ldots, i_m$)=WsA($i_1, \ldots, i_m$) = ϕ

Input: For each reference site in the loop, give an instance of array A's subscripts $S_1, \ldots, S_m$, enclosed in n-nested loops with indices $I_1, \ldots, I_n$

IF it is a *read operation* do:
    SET This_iteration = IS_Encode($I_1, \ldots, I_n$)
    IF WsA($S_1, \ldots, S_m$) = ϕ THEN
        no dependence
    ELSE
        IF WsA($S_1, \ldots, S_m$) $\neq$ This_iteration THEN
            There exists a *flow dependence* arrow from iteration
            IS_Decode(WsA($S_1, \ldots, S_m$)) to ($I_1, \ldots, I_n$)
        ENDIF
    ENDIF
    IF RsA($S_1, \ldots, S_m$) =ϕ THEN
        no dependence
    ELSE
        There exists an *input dependence* arrow from iteration
        IS_Decode(RsA($S_1, \ldots, S_m$)) to ($I_1, \ldots, I_n$)
    ENDIF
    SET RsA($S_1, \ldots, S_m$) = This_iteration
ENDIF
IF it is a *write operation* do:
    SET This_iteration = IS_Encode($I_1, \ldots, I_n$)
    IF RsA($S_1, \ldots, S_m$) = ϕ THEN
        no dependence
    ELSE
        There exists an *anti dependence* arrow from iteration
        IS_Decode(RsA($S_1, \ldots, S_m$)) to ($I_1, \ldots, I_n$)
    ENDIF
    IF WsA($S_1, \ldots, S_m$)= ϕ THEN
        no dependence
    ELSE
        There exists an *output dependence* arrow from iteration
        IS_Decode(RsA($S_1, \ldots, S_m$)) to ($I_1, \ldots, I_n$)
    ENDIF
    SET RsA($S_1, \ldots, S_m$) = ϕ
    SET WsA($S_1, \ldots, S_m$) = This_iteration
ENDIF

* IS_Encode and IS_Decode are functions that convert $N^m \rightleftharpoons$ N, real iteration address $\rightleftharpoons$ an unique integer iteration ID.
* Just see scalar variable as one element array.

Fig. 1. Iteration dependence tracer algorithm

1. *Prepare required syntax information*
   Record all variable nodes of syntax tree, which appear in the selected loops and belong to both read reference set and write reference set.
2. *Generate runtime test program which is appended necessary function calls*
   Generate a call statement with necessary information for each variable node on dataflow order. If there exists call statement in the loop body, we have to duplicate this subroutine, treat it as a new one, then pass actual arguments with their shadow variables together. In subroutine body, we do the same thing and insert marker functions for all variable reference sites. Of course, there are also need a few initialization and clean-up functions.
3. *Compiler and run the program*
   Use normal Fortran compiler and predesigned library, we get an executable program. Run it and record intermediate data in a temporary file, the data like ($iteration_1$, $iteration_2$, dependence kind, memory location).
4. *Post process the first-hand data*
   By now, we get accurate flow and output dependencies. For anti dependence, it is not as precise as the former. The cure algorithm see Figure 2.
5. *Visualize iteration space dependence graph*
   Due to the complexity both on iteration space dependence graph itself and presentation, we only show it in two dimensions within a predefined region.

Now, we illustrate it using a sample(see fig. 3).

For all records do:
  If it is an *input dependence* from iteration I_1 to I_2, just insert it into a list.
  If it is a *flow dependence* or *output dependence*, delete all elements in the list which relate to this memory location.
  If it is an *anti dependence* from iteration I_1 to I_2, then retrieve all records in the list which relate to the same memory location from J_1 to J_2 , and convert it to an anti dependence arrow from I_1 to J_2.

Fig. 2. Post process of intermediate data

3 Experiments

In a sense, as compared with dependence graph, the iteration space dependence graph (see fig.4 - 5) is more comprehensible to the user, which summarizes the dataflow restricted relationship between iterations and gives a vivid picture of the loop-carried data dependencies of a given nested loops. It should be more easily accepted by the user.

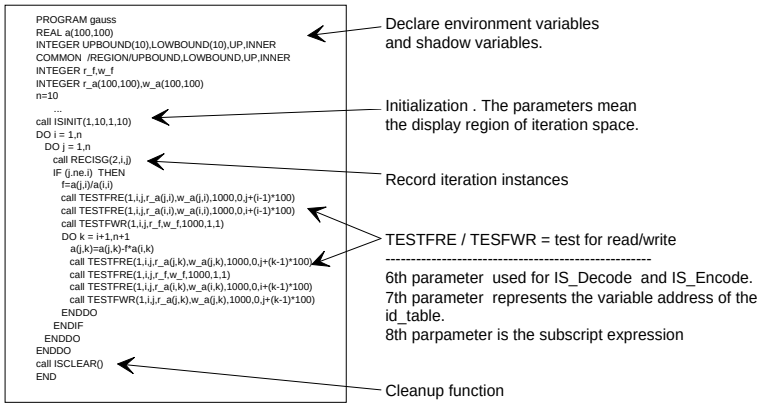

Fig. 3. The source code of "Gauss" test program

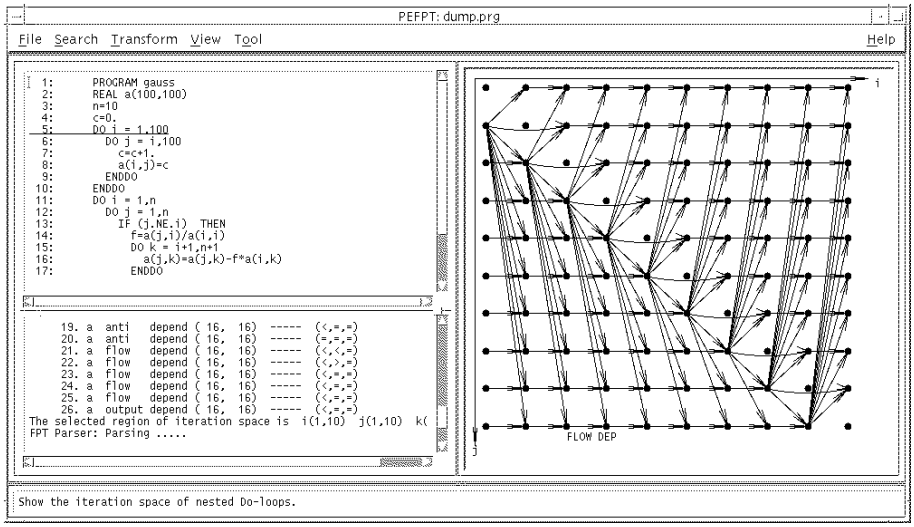

Fig. 4. The iteration space dependence graph of "Gauss" on variable `f`, consider loop I and J with bounds $(1,10) \times (1,10)$

In order to help the user to understand the semantics and guide the user to parallelize the loops, we not only distinguish between different dependencies by respective colors, but also design a dependence filter to get a partial picture of various types of dependencies on specific variables.

Generally, the user cannot violate and eliminate existent flow dependencies, unless he adopts a new algorithm to rewrite the loops. However, the other dependencies can be eliminated using appropriate techniques, such as scalar expansion and variable privatization. In a way, it will simplify the graph and give more possibilities to parallelize the loops.

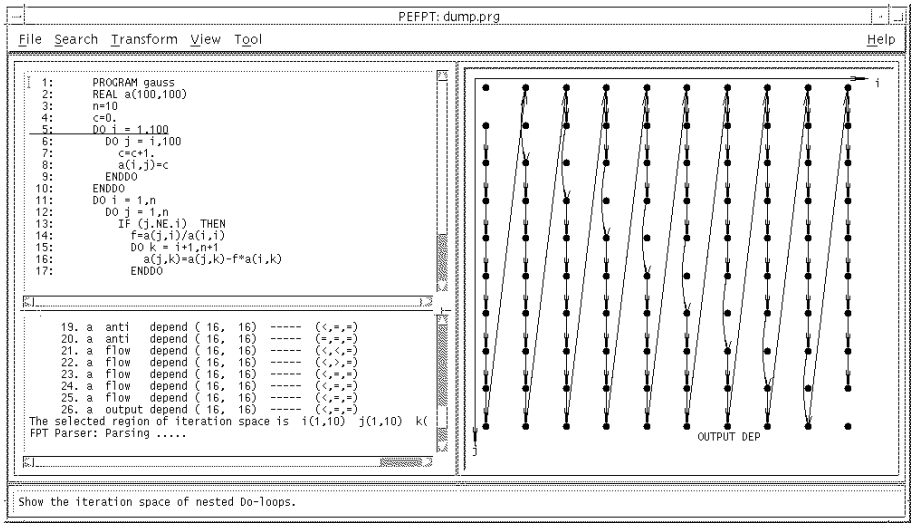

Fig. 5. The iteration space dependence graph of "Gauss" on variable "a", consider loop I and J with $(1, 10) \times (1, 10)$

Although there have already deposited many known techniques on loop transformations for decades, which can result in better speedup through rearranging or partitioning iteration space, most of them are only suitable for certain loop models. Unfortunately, the compiler does not realize it easily by itself.

Moreover, in order to verify the correctness for certain case, most of them need strict dependence information. For example, the premise of well-known unimodular transformation is to get constant dependence distance vectors. It is another barrier that there is no known optimal order in which these transformations should be applied. The iteration space dependence graph gives the user more opportunities to play a part in optimization. Through the graph, the user will easily find out some solution.

For example, see fig.4, the arrows represent the overwrites of scalar variable *f*, it exposes the lexicography of the loops. The fig.5 tells the user that each iteration on vertical direction does not exist restraint, all arrows cross left line to right line. Therefore, the inner loop can be parallelized.

For more complicated cases, the user needs more sophisticated interactive method to describe and perform desired rearrangement and partition. Unimodular transformations are a potential area of interest here. E.g. the user composes suitable unimodular matrix under the guidance of the graph, then the system automatically calculates new loop indices and bounds.

4 Conclusion

Iteration space dependence graph is an attractive compiler information. It promises the user new opportunities to exploit more parallelism, which are normally abandoned by the compiler for inaccuracy and complexity of analysis. In a way, the method we implemented is effective and efficient, especially using runtime simulation, which effectively avoids the drawback of dependence analysis we mention before.

References

1. Michael Wolfe, *"Optimizing Supercompilers for Supercomputers"*, Ph.D. thesis, University of Illinois, 1982.
2. Utpal Banerjee, *"Dependence Analysis for Supercomputing"*, Kluwer Academic Publishers, 1988.
3. K.McKinliey, *"Evaluation Automatic Parallelization for Efficient Execution on Shared Memory Multiprocessors"*, ICS'94, pp.54-63, 1994.
4. W.Blume and R.Eigenmann, *"Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs"*, IEEE Transaction on Parallel Distributed Systems, 3(6), pp. 643-656, Nov. 1992.
5. K. Cooper et al., *"The ParaScope Parallel Programming Environment"*, Proceedings of the IEEE, 81(2), Feb. 1993.
6. C.D.Polychronopoulos et al, *"Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors"*, Inter. Conference on Parallel processing pp. II-39–II48 1989.
7. F.B.Zhang *"The FPT Parallel Programming Environment"*, Ph.D. thesis, University of Gent, 1996.
8. Q.Wang, Y.J.Yu and E.H.D'Hollander, *"Interactive Programming using* PEFPT*"*, Syllabus of the Parallel Computing Seminar, T.U.Delft, pp. 125-130 1996.

This article was processed using the LaTeX macro package with LLNCS style