Interactive Programming using *PEFPT*[1]

Wang Qi, Yijun Yu and Erik D'Hollander

University of Ghent
Dept. of Electrical Engineering
St.-Pietersnieuwstraat 41
B-9000 Ghent
wang@elis.rug.ac.be

Abstract

Current automatic parallel technology cannot detect and exploit quite a part of parallelism in real Fortran programs, for lack of application specific knowledge and its inaccuracy of dependence analysis. So it is meaningful to investigate the technique and the methodology of interactive parallel programming. In a joint project between the universities of *Ghent*(B) and *Fudan*(PRC) called *PEFPT* (the Parallel Programming Environment for *FPT*, the Fortran Parallel Transformer), we are developing an integrated toolkit to help scientific programmers to implement correct and efficient parallel programs.

1 Introduction

Parallel processing is more and more considered as the essential way to speedup massive computational application. But it is also proven extremely difficult for users to develop, maintain and transplant. An ideal solution is to ask compilers to automatically parallelize and optimize programs written in conventional language such as F77, so as to take advantage of particular parallel architecture. Many efforts have been done on this subject, such as *SUIF* (Stanford university[4]), *Polaris*(Illinois university[5]), *FPT*(Gent university[9]). The results have been both

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

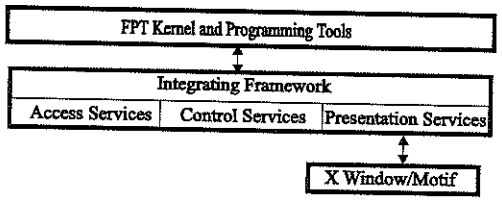

Figure 1: The System Architecture of PEFPT

encouraging and disappointing[1, 2]. One of principal drawback is the inaccuracy of their dependence analysis using current technique: symbolic expression, procedure calls, induction and reduction variables, and complex control flow, all of them introduce conservative assumption. The other way is to develop new generation parallel languages, such as *HPF*, *PCF*, etc., but it seems also very difficult for the user to reach heartening results.

A tradeoff is to provide an interactive programming or optimizing environment, sharing part of responsibility with the User under the guide of the system, which the user can afford. There are limited works on this region, such as *Parafrase-2*[6] and *Parascope*[7].

As a result, we propose to design a new programming environment – *PEFPT*, based on the work of *FPT*, developing and integrating a series of tools and methods, and testing them through practice.

2 Overview of the system

2.1 System architecture

One of goals of *PEFPT* is to construct a consistent user interface that is easy to learn and use, so a well-designed system architecture is indispensable (see figure 1). In this respect, we fully think about the difficulties of *FPT* itself and further expansion, through defining three level services (*access, control*, and *presentation services*) to integrate into a consistent and independent context.

2.2 Semantic information browser

In order to give the user an opportunity to intervene with the process of parallelization, it is much important to expose correct and complete semantic information. Unfortunately, there are too much data necessary for correct transformation, and they are very complicated and tedious indeed. Presenting them carelessly only

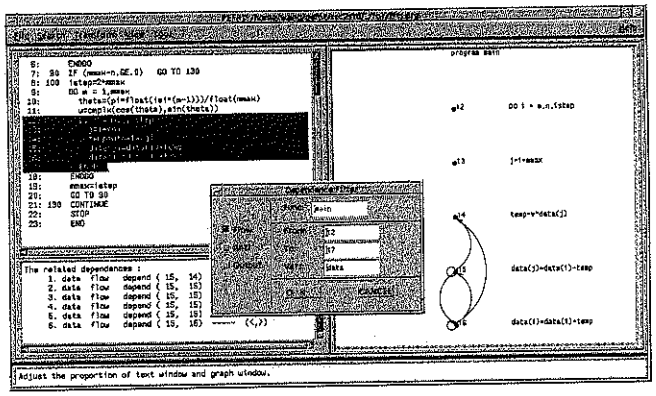

Figure 2: A Dependence graph of "FFT"

as statement pair, variable name, type of dependence, direction vector) in the message pane.

- *Task graph:* A task is a set of instructions that must be executed sequentially while different task can be executed concurrently through synchronization. *PEFPT* presents a picture of fine functional partition tasks using *Fork* and *Join* primitives[3][9].

- *Iteration space dependence graph:* Normally, the user pays more attention to the dependencies which cross the loop, namely loop-carried data dependencies, which prevent loop iterations from executing parallel. At this point, the iteration space dependence graph is just right, it exposes the execution constraints among iterations clearly.

  In *PEFPT*, we implement two different strategies to analyze and gather necessary data. One of them is to use a run-time solution, try to overcome the weakness of traditional dependence analysis algorithms on complicated control flow and subscript expression. The following figure 3 is an example of "FFT", which displays the iteration space dependence graph of the nested loops after projecting on the plane of loop "K1" and "I" [4].

  Compared with dependence graph, to a great extent, it is more summary and understandable to the user, and it is accurate too. Similar to the

[3]Task graph is used to explain VPS code

[4]The loop "K1" is derived from "Goto" removal– a transformation provided by *PEFPT; the filter condition is flow dependence of array "data".*

frustrate and fuzzy the user. Therefore, the fundament of programming environment is to determine which interactions are profitable, which data are necessary for such interactions, and how to exhibit this information so that the user can accept it.

By now, the system has presented four information graphs. They are call graph, dependence graph, task graph, and loop iteration dependence graph.

- *Call graph:* In the system, we see a *function* as a base unit to be analyzed and transformed. Therefore, the call graph is the default one in the canvas pane which shows the relationship between caller and callee in the program. We provide a navigate operation on it, click on the function node, then unparse the relevant function from internal syntax-tree into the text pane, and maintain consistence of environment data.

- *Dependence graph:* The analysis of precedence constraints on the execution of the statements is a fundamental step toward program parallelization.

  There are four types of data dependence [3] 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.

  Most of parallelization technology is based on dependence information. Therefore, it is critical to maintain and present dependence graph. An illustration of it is given in figure 2, which is a part of dependence graph of "FFT" under specific conditions[2].

  In this figure, the dependence filter dialog is used to define the category the user wishes to deal with, which includes the program segment, the class of dependence and a variable list. This feature is needed because there are often too many dependencies for the user to effectively comprehend. In order to give more convenience, we define two operations on the dependence graph: click on the edge or node to highlight statements in the text pane which involve in the dependencies, and list detail information (such

[2]the filter condition is from statement 12 to 17, flow dependence and array *"data"*

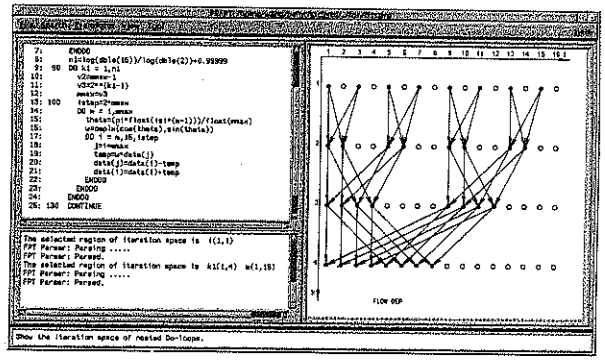

Figure 3: An iteration space dependence graph of "FFT". Black nodes indicate the iterations executed.

former, the system uses the same filter utility to decompose the constraints into different types of dependence edges.

A lot of loop transformations can take advantage of such information. We can use the variable privatization technology to eliminate most of anti and output dependencies. As to flow dependencies, it is possible to gain considerable parallelism through iteration space re-arrangement or iteration partition, such as loop skewing, wavefront and unimodular transformation. Anyway, it will inspire the user an idea to do something himself. For example, the figure 3 indicates that in vertical direction every line of iterations must be executed sequentially, and in horizontal direction, the iterations in one line can run concurrently.

2.3 Program transformation

PEFPT has implemented a series of transformations to expose and make use of inherent parallelism in the program, which includes: program restructuring, loop unrolling, SSA(Static Single Assignment), loop parallelization, and unimodular transformation[8][9].
Unlike an automatic system, which possibly use command-line switch to control the transformation, *PEFPT* adopts a more flexible strategy to integrate them, apply on demand, guided by semantic information and operator's own decision. Namely, the user specifies a proper transformation to be performed, and the system carries out the mechanical details and maintains the consistence and correctness of IR (Intermediate Representation). Otherwise, systems give a reason if such action can not work.

2.4 Code generation

The efficiency of the target program depends too much on the architecture of the target machines. Unfortunately, there is still no prevalent one. So it is vital to automatically generate high efficient target code for specific architecture to help the user make full use of its feature. It alleviates the burden of the user to a great extent on porting and optimizing data and communication. In particular, *PEFPT* currently supports *Fortran/MP*, *Multi-threaded* code for Sun Sparc, *PVM*(Parallel Virtual Machine) and *VPS* (Virtual Processor System)[9].

3 Future enhancement

Although *PEFPT* is proven that it has good features to help the user to improve his work, it is also obvious that the current functions implemented are not enough to deal with real programming. Moreover, we need more sophisticated information organization, more effective mechanism to maintain the consistence between IR derived from the system and user assertions, and more practice on parallel programming.
As a result, we propose to analyze and transform some real programs to gain further results using both *PEFPT* and *FPT*, and compare them to get knowledge of necessary improvement for *PEFPT*, especially on what automatic system cannot work perfectly but the user can make up. On the other hand, we can verify the value of current features implemented in *PEFPT*.

4 Conclusion

PEFPT was designed in the context of the *FPT*. It intends to overcome shortage of completely automatic systems through interactive programming. We believe it provides the user for such a solution: it permits the user to develop program on his application specified knowledge, aided by full analysis information the system gathers, and it correctly carries out a set of transformations to enhance parallelism under the user's decision. All of them will lead to high performance output. On the other hand, *PEFPT* is also an aid to object teacher which help the user to understand the terms and behavior of compiler and parallel processing.

References

[1] K.McKinliey, *"Evaluation Automatic Parallelization for Efficient Execution on Shared Memory Multiprocessors"*, ICS'94, pp54-63, 1994.

[2] 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.

[3] G. Goff, K. Kennedy, and C.-W. Tseng, "Practical Dependence Testing" Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI'91), SIGPLAN Notices, 26(2), June 1991.

[4] Stanford Compiler Group, *"The SUIF Parallelizing Compiler Guide"*, Technique Report 1994.

[5] *"Polaris: The Next Generation in Parallelizing Compilers"*, Technique Report.

[6] K. Cooper et al., *"The ParaScope Parallel Programming Environment"*, Proceedings of the IEEE, 81(2), Feb. 1993.

[7] C.D.Polychronopoulos et al, *"Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors"*, Inter. Conference on Parallel processing 1989, pp. II-39–II48 1989.

[8] E.H.D'Hollander, *"Partitioning and labeling of loops by unimodular transformations"*, IEEE Transaction on Parallel Distributed Systems, 3(4), pp. 465-476, 1992.

[9] F.B.Zhang *"The FPT Parallel Programming Environment"*, Ph.D. thesis, University of Gent, 1996.