PMACS : An Environment for Parallel Programming

B.Dehbonei, C.Laurent, N.Tawbi
Bull Corporate Research Center
Louveciennes, France

R., S.Kulkarni
PSI Data Systems
Bangalore, India

Abstract

In this paper, we present a parallel programming environment called PMACS. The central structure of this environment is the Parametrized Dependence Graph that gives a very precise representation of dependencies. It also facilitates the recompilation of dependencies whenever the user is requested to introduce information about the program. A unique framework for the interprocedural constant propagation and the interprocedural side-effect analysis, based on symbolic evaluation, is used. A powerful tool for displaying graphs has also been carried out. It allows the user to understand the detected inherent parallelism of his program. A first release of PMACS written in C is already implemented.

1 Introduction

Using programming environment for parallel programming is believed to be the best way to produce the most correct and efficient programs. A parallel programming environment is a collection of tools that collaborate in order to transform and generate efficient programs dedicated to the execution on parallel machines. Many parallel programming environments have been carried out by research centers : Parascope [3], developped at Rice University, is based on a editor that enables the user to transform the program in order to remove dependencies. Faust [11], developped at CSRD, is based on the Sigma editor, which allows the user to restructure his program, according to the target machine on which he wishes to execute his program. In Ptran [1] project at IBM Research, the environment takes into account the issue of executing parallelized programs on target machines.

PMACS is an experimental parallel programming environment. The novelty of PMACS is the semantic analysis it performs. This analysis is based on a powerful symbolic evaluation technique that enables PMACS to detect very complicated cases of loop invariant, induction variables and reduction operations [12].

This technique has been extended for performing a symbolic-based flow-sensitive interprocedural data-flow analysis [9]. A unique framework deals with both the interprocedural constant propagation [6] and the interprocedural side-effect analysis [7]. The symbolic data-dependence analysis is also another important feature of PMACS. It allows the environment to compute the Dependence Graph more precisely and also to update it more efficiently.

We give a brief presentation of the PMACS architecture. Then, we emphasize four of the interesting features of PMACS : The editor, the Symbolic Dependence Analysis, the graph displayer and the interprocedural data-flow analysis.

2 Architecture

PMACS is composed of a set of tools that compute information and store them in a central database. A program is composed of modules : main program, subroutines and functions. Each module has a set of attributes, stored in the database of the environment. These attributes are information that can be computed without taking into account the calling context of a module. Therefore, any attribute associated with the whole program is calculated with a combination of the attributes of all the modules that compose this program. Each of these attributes is computed by a tool.

For instance, the intermediate representation form of a module is automatically computed by the editor that calls a parser for Fortran 77. This important feature allows PMACS to perform some preliminary analysis whenever a module is saved in the editor.

The editor edits modules and compose them in order to create programs. The parallelization can be applied to a whole program or a module. The result of the interprocedural data-flow analysis is associated with each module in the database. In order to understand the result of the analysis performed by the parallelizer, the user can consult the Dependence Graph by means of the Graph displayer and the selection tool (see below). These tools communicate through the database. Tools are activated by the *process manager* which is written in Emacs-Lisp, the programming language of the Gnu Emacs editor. The code of the process manager is accessible to the user. Consequently, he is able to reconfigurate the way tools are activated like in SIGMACS [13]. Another flexibility comes from the window manager chosen in PMACS. GWM has a programming language that enables the user to change the buttons associated with the main window of PMACS. One can imagine to change the callbacks procedures associated with buttons, or to design its own window decoration. The Parametrized Dependence Graph is the central structure of PMACS. The main goal of the environment is to reduce as much as possible the number of edges of this graph.

3 Editor

The Pmacs Editor is based upon the standard editor Emacs and more precisely upon a recent version called Epoch, which is especially interfaced with the X Window System. The use of the GWM window manager which is programmable introduces flexibility inside the environment design. The language used to program GWM is an interpreted Lisp-like language called WOOL (Window Object Oriented Lisp). Communication between the window manager and the Editor is carried out with the help of Emacs-Lisp functions.

An important function provided by the editor is the selection function. It allows users to focus on a specific loop they want to parallelize. Clicking with the mouse on any statement generates the highlight of the most closely surrounding loop in the Emacs window. Parallelization can then be activated only for that loop.

4 Symbolic Dependence Analysis

The dependence analysis is the kernel of any parallelizing system. The result of the dependence analysis is the dependence graph (DG). The nodes of this graph are the statements of the program. An edge (S_1, S_2) means that statement S_1 must be executed before statement S_2 otherwise the program semantics could be modified. The dependence analysis between two statements that manipulate array expressions is very complicated. Several *dependence tests* have been proposed in order to find out if two array subscript expressions reference the same memory locations. Among these tests, one can mention the Wolfe-Banerjee test [2], the Fourier-Motzkin method [8]. In PMACS, the dependence test is PIP (Parametric Integer Programmaing) [10]. It is designed to find the solution in the integer domain of a set of linear inequalities, therefore it can be used as a dependence test. It is accurate and according to the nature of subscripts in [14] about 85.3% of the studied case could be exactly solved by PIP. Furthermore, PIP can deal with unknown variables (called *parameters*) involved in subscript expressions. Consider as an exemple the following program sketch:

```
do i = 1,n
   a(i,i) = ....          (s1)
   ...    = a(i, n-i+1)   (s2)
end do
```

The existence of a dependence between s_1 and s_2 depends on the existence of integer solutions for the system:

$$\begin{aligned} i &= i' \\ i &= n - i' + 1 \\ 1 \leq\ & i \leq n \\ 1 \leq\ & i' \leq n \end{aligned}$$

where i and i' are the variables and n is the parameter. The answer as given by PIP is:

$$i = i' = \text{if } 2 \times \frac{n+1}{2} - n - 1 \geq 0 \text{ then } \perp \text{ else } \frac{n+1}{2} \quad (1)$$

where a fraction indicates integer division and the sign $\perp$ indicates no solution. The above result clearly says that there is a dependence iff $n+1$ is even. The result of the symbolic dependence analysis is the *Parametrized Dependence Graph* (PDG) where edges are labelled by conditions such as (1). Additional information about

parameters may be available either by performing interprocedural analysis or by interaction with the user. According to these information, the PDG is updated. Since the PDG is computed independently from the call site, it is computed once for any module and used at any call site. A special structure has been chosen in order to reduce significantly the recompilation time whenever new information about parameters are available.

5 Graph Displayer

As PMACS has an interactive Parallelizer, it is useful to provide the user with a display of the dependence graph. The Graph Displayer depicts the dependence graph pictorially and provides some additional features as detailed below. The nodes should be placed in such a way that the edge crossings are minimum. The problem of displaying a graph has been widely discussed and many of the approaches attempt to get a logical planar mapping of the graph and perform a logical to physical translation. In [15], a hierarchy is identified and defined in the nodes to obtain the logical mapping. Our logical mapping of the graph is the *attributed depth first search tree* of the graph. New attributes are defined for nodes and edges and an attempt is made to construct a depth first tree in which the crossings between back edges is minimum. A modification of the depth first search is used so that an order exists in choosing the next node. The nodes are ordered such that the span of the back edges is minimum. Also, heuristics are used to draw the back edges on either side of the tree. The resulting depth first tree with curved back edges is the required logical mapping of the graph. Logical to physical translation is done by converting the arcs of the back edges to straight line edges appropriately. The algorithm to obtain the logical mapping has the same complexity as the depth first search algorithm. The nodes of the graph are scanned twice for the logical-to-physical translation.In effect, with a simple heuristic a neat display of the graph is obtained. To take care of the limitations of the approach, a facility is provided to *dynamically* move the nodes. Since the logical mapping of the graph has a tree structure, the approach can be used to display the Call graph of the environment also. In addition, a filter facility is provided to display sub-graphs of the dependence graph. Filters are provided so that the user can select, dependences of a particular type, the dependence graph above a certain depth, dependence relation within a given range of statements, dependence caused by a set of variables, a subgraph resulting from any combination of the above four filters.

6 Interprocedural Symbolic Data-Flow Analysis

The first phase of the dependence analysis consists in computing the effects of the statements of a program. In presence of procedures, the side-effect analysis is more complicated. In other terms, the dependence analyzer should know if a parameter is used or modified after the associated subroutine is called. A first approximation of side-effects is the following : Any actual parameter is both modified and used. [4] mentionned that this pessimistic strategy over-estimates up to 90% the size of the Dependence Graph. A second strategy analyzes the called subroutine without taking into account its internal control flow graph. This is called *the flow-insensitive side-effect determination*. In [7], the authors propose a fast algorithm for computing the flow-insensitive side-effect. A third way of performing interprocedural side-effect analysis is flow-sensitive. The analysis takes into account the internal flow graph of each subroutine. The only method proposed for that is due to [4]. On the other hand, the parallelizer can compute more precisely the dependencies if it knows which variables will have constant values, and what those values will be, when a procedure is invoked. This is the interprocedural constant propagation. In [6], a bunch of methods for determining these constants are described.

In PMACS, two interprocedural methods are provided: The flow-insensitive method due to [7] and a new value-oriented interprocedural data-flow method [9] that has been carried out. The latter unifies two interprocedural frameworks : The interprocedural constant propagation and the side-effect determination. The main idea is to perform a symbolic evaluation over the program. The result is then interpreted in order to either determine constants or to calculate side-effects. The symbolic evaluation of the commands is straightforward except for conditionals , procedure calls and loops. The first case is studied in [12]. The second is a natural extension of the standard semnatics of procedure calls. Finally, [9] proposes to approximate the computation of the fixpoint corresponding to the result of evaluation of loops by the detection of *Generalized Reductions* [12]. These programming paradigms are frequently used in scientific programs: Loop invariants, induction variables, reduction operations and re-

currences. Consequently, whenever a loop is encountered, first the generalized reduction detection is performed. Then, those variables that are modified inside the loop are analyzed. If a variable is a generalized reduction, its symbolic value will be inserted in the resulting store. Otherwise, a pessimistic alternative is chosen: The variable takes the value **unknown**. The correctness proof of this method based on the theory of abstract interpretation [5] is found in [9]. Once the symbolic evaluation of a procedure is performed, the resulting symbolic store is analyzed. [9] discussed several strategies for interpreting this store. Here, we discuss two interprocedural frameworks : constant propagation and flow-sensitive *modify* problem. The *modify* problem consists in determining if an actual parameter is modified by the invocation of a procedure. The *use* problem that determines if a variable is used in an invocation of a procedure, is fully discussed in [9]. Since many sophisticated tools are involved in the process of symbolic evaluation (symbolic arithmetic expression simplifier, symbolic boolean expression simplifier, symbolic store manipulator,..), a special attention has been paid to the internal structures used in the implementation. Subexpressions are shared such as in [16], decision graphs are used, stores are hash-coded.

We plan to extend all the techniques used in PMACS in order to analyze and parallelize languages that manipulate pointer variables.

References

[1] Allen, F., Burke, M., Charles, P., Cytron, R., and Ferrante, J. *An Overview of the PTRAN Analysis System for Multiprocessing.* Journal of Parallel and Distributed Computer, Vol. 5, 1988.

[2] Banerjee, U., *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Norwell, Massachusetts 1988.

[3] Balasundaram, V., Kennedy, K., Kremer, U., McKinley, K., and Subhlok, J. *The Parascope Editor : An Interactive Parallel Programming Tool.* Rice COMP. TR89-92, May 1989.

[4] Callahan, D., *The Program Summary Graph and Flow-sensitive Interprocedural Data Flow Analysis.* In the Proceedings of Sigplan PLDI'88. Atlanta, Juin 88.

[5] Cousot, p., and Cousot, R. *Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximations of Fixpoints.* In Proceedings of the 4th ACM POPL77, January 1977.

[6] Callahan, D., Cooper, K., Kennedy, K., and Torczon, L., *Interprocedural Constant Propagation.* In Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction, June 1986, pp. 152-161.

[7] Cooper , K., Kennedy, K., *Interprocedural Side-effect Analysis in linear Time.* In Proceedings of the ACM SIGPLAN PLDI'88, June 1988

[8] Duffin, R., *On Fourrier's Analysis of linear Inequality Systems.* Mathematical Programming Study 1, pp. 75-95, North Holland, 1974.

[9] Dehbonei, B., *Etude de la génération de code et de l'analyse interprocédurale au sein d'un environnement de programmation parallèle.* Ph.D. Thesis, University Paris 6, MASI Lab. December 1990.

[10] Feautrier, P. *Parametric Integer Programming* R.A.I.R.O. Recherche opérationnelle/operations Research vol 22, $N^{\underline{o}}$ 3 1988

[11] Guarna, V., Gannon, D., Gaur, Y., and Jablonowski, D. *Faust : An Environment for Programming Parallel Scientific Applications.* ACM-IEEE Supercomputing'88, Orlando, November 1988.

[12] Jouvelot, P., and Dehbonei, B., *A Unified Semantic Approach for the Vectorization and Parallelization of Generalized Reductions.* In the Proceedings of the ACM Sigarch ICS'89 Crete, Greece June 1988.

[13] Shei, B., and Gannon, D., *Sigmacs : A Programmable Programming Environment.* In the Proceedings of the International Workshop on compilers for Parallel Machines Irvine, 1990.

[14] Shen, Z., Li, Z., Yew, P.-C., *An empirical Study On Array Subscripts And Data Dependencies.* In the Proceedings of the International Conference on Parallel Processing, August, 1989.

[15] Rowe, L., Davis, M., and al *A Browser for Directed Graphs*, Software, Practice and Experience, pp. 61-76, vol. 17, Jan. 1987.

[16] Reif, J., Tarjan, R. *Symbolic Program Analysis in Almost Linear Time*, SIAM journal of Computing, pp. 81-93, vol. 11, Feb. 1981.