ParaScope:
A Parallel Programming Environment[†]

David Callahan
Keith D. Cooper
Robert T. Hood
Ken Kennedy
Linda M. Torczon

Department of Computer Science
Rice University
Houston, Texas 77251-1892

1. Introduction

Because parallel processing offers the promise of higher execution speeds than can be achieved by conventional single-processor systems, almost every new supercomputer will employ it in some form. Unfortunately, parallelism is not without its cost, because much of the burden of achieving high performance is shifted to the programmer. Unless a computation is properly organized to fully exploit the available parallelism, it is likely to achieve only a fraction of the potential performance of a parallel computer system.

Experience has taught us that parallel programming is significantly more difficult than sequential programming because it requires that the user manage and synchronize many concurrent tasks that interact in ways that are hard to predict. The most vexing problems arise because the schedule of execution of concurrent tasks in not deterministic. This leads to *schedule-dependent* errors, which depend not only on the program and its inputs but also on the schedule of execution chosen by the system at run time. To make matters worse, these errors usually disappear whenever any type of debugging is attempted because the presence of the debugger changes the execution schedule.

Of course, most of these difficulties would be avoided if the compiler could automatically translate sequential programs to a form suitable for parallel execution. Then the user would need only to write and debug a sequential program and the compiler would assume the responsibility for achieving high performance through the use of parallelism[1]. In a sense, this would reduce parallel programming to an optimization.

For the past six years, a group of researchers at Rice University has been developing a system called PFC that translates ordinary FORTRAN programs to an equivalent parallel program written in an extended dialect of FORTRAN [AlCK 87]. Similar efforts have been underway at the University of Illinois [KKLW 80, KKLP 81] and IBM Research [ABCC 87]. Section 2 of this paper reviews the work at Rice and reports the reasons why we have concluded that automatic techniques, by themselves, will not be sufficient to achieve high performance on asynchronous parallel systems. In other words, our research has convinced us that the programmer will need to be involved in the specification of parallelism at some level.

The logical question to ask next is whether we can use some of the analysis techniques from automatic translation systems to help the programmer produce correct parallel programs. Section 3 discusses one such effort undertaken at Rice. The result was a program understanding tool called PTOOL that is now being used at Los Alamos National Laboratory and Cornell National Supercomputer Facility. Section 3

[†] This work was supported in part by the National Science Foundation under grants ASC-8518578, CCR-8619893 and CCR-8706229, by the IBM Corporation and by Cray Research, Inc.

[1] There might still be the possibility that ordinary sequential programming errors might go undetected when executed on a sequential machine and then appear under parallel execution. We ignore this problem for the moment because it is fairly easy to design a debugger to help locate errors of this sort.

also contains a critique of the weaknesses in the design PTOOL that have been discovered through extensive interactions with its users.

Because asynchronous parallel machines need fairly large regions to run in parallel to compensate for the high cost of process startup and synchronization, it is important to analyze the whole program rather than one subroutine at a time. Both PFC and PTOOL analyze the flow of information on a program-wide basis. This is usually referred to as *interprocedural analysis* in the computer science literature. The problem with interprocedural analysis is that it requires an examination of the whole program. PFC and PTOOL accomplish this by reading and analyzing every module in the program, a time-consuming process. This approach is not practical in a production compilation system, because it would require recompiling the whole program after any change, no matter how small. Section 4 describes a FORTRAN programming environment, called , that was designed to support interprocedural analysis without sacrificing the benefits of separate compilation [CCHK 87]. The environment is currently being tested at the IBM Scientific Center in Palo Alto, California and is scheduled for installation at the Cornell National Supercomputer Facility in early 1989.

The deficiencies of PFC and PTOOL have led us to design and build ParaScope, an extension to the programming environment that assists the programmer in developing and debugging parallel programs, which is described in Section 5. When complete, ParaScope will consist of a collection of tools for elaborating a parallel program in an extended dialect of FORTRAN. The tools include an intelligent editor for FORTRAN that helps the user restructure programs for parallelism, a whole-program compilation system that hides the low-level details of parallel programming from the user, and an execution monitor that incorporates advanced techniques for locating schedule-dependent errors.

The paper concludes in Section 6 with a status report on the implementation and a discussion of issues that are yet to be resolved.

2. Automatic Decomposition

Since 1978, Rice has been conducting an active program of research in software for vector and parallel supercomputers. PFC (Parallel FORTRAN Converter) is a system that automatically vectorizes FORTRAN programs by performing a sophisticated analysis of *dependences*. A dependence exists between two statements if one statement can store into a location that is later accessed by the second statement. Although most optimizing compilers analyze dependences in a program, they use a particularly naive treatment of arrays. Vectorization systems employ a much more powerful analysis that is fairly effective in dealing with subscripted references in loops [Kuck 78, KKLW 80, AlKe 86]. Research on PFC concentrated on finding vectorization algorithms efficient enough for use in a compiler. An indication of its success is that PFC served as the prototype for the IBM VS FORTRAN Version 2 vectorizing compiler for the 3090 Vector Feature [ScKo 86], which achieves excellent results while remaining reasonably efficient.[2]

Overall, vectorization has been an enormous success. Every manufacturer of vector computers offers a vectorizing compiler that permits the user to program in FORTRAN 77 and achieve high efficiency, so long as the program is properly structured. Most of these compilers are based on principles developed at the University of Illinois by Kuck and his colleagues and by our group at Rice.

Since completion of the original PFC in 1982, research has concentrated on the problem of automatic parallelization. Our approach was to look for DO-loops whose iterations could be run in parallel on different processors. This strategy was motivated by the observation that most successful efforts to convert scientific programs for parallel execution employ *data decomposition*, a technique by which different processors compute different components of the result array in parallel. In FORTRAN, the DO-loop is the usual way to express a computation that is to be applied to corresponding elements of an array; hence, it seemed natural to try to run such loops concurrently.

[2] Although the IBM compiler is based on an early version of the PFC system, it is quite effective at vectorization and runs only 25% slower than the scalar VS FORTRAN compiler with optimization.

Dependence plays a key role in finding parallel loops, as the following example will illustrate.

The second statement depends upon the first one because it uses a value in the array A that is stored by the first statement on the same iteration of the loop. We call this dependence *loop-independent* because the value flow depends in no way on the order of iteration of the loop. However, there is also a dependence of the first statement upon the second because the second statement stores into a location in the array B that is used by the first statement on the next iteration of the loop. This dependence is called *loop-carried* because a variation in the order of the loop iterations can change the flow of values and lead to incorrect results. In fact, we have the following principle [AlKe 85]. **Fundamental Principle of Loop Parallelism.** *The different iterations of a given loop may be executed in parallel without synchronization if and only if the loop carries no dependences. This principle is illustrated in Figure 1. If we allocate the first and second iterations of the loop to processors 1 and 2 respectively, processor 2 may attempt to fetch B(2)* **before the desired value is stored by processor 1. Hence, synchronization is required to guarantee that the computation produces correct, and deterministic, results.**

Whereas vectorization attempts to parallelize inner loops, the overhead process startup and synchronization requires that we find larger regions to run in parallel on a multiprocessor. Hence, we must look for outer loops containing as many statements as possible. To be effective, a parallelization system must seek to fuse short parallel loops into longer ones and interchange parallel loops outside of sequential ones so that synchronization occurs less frequently. These techniques have all been implemented in PFC. In addition, it is important to parallelize loops that contain subroutine calls. This requires interprocedural analysis if it is to be done safely; hence, PFC incorporates a systematic analysis of interprocedural side effects.

So how well have we done? Our experiments are not yet complete, but we are in a position to draw a few preliminary conclusions. PFC and similar systems are capable of some dramatic accomplishments. For example, a version of PFC has generated a parallel matrix multiplication program that achieves a speedup of 8 on a 16-processor hypercube for 100 by 100 matrices. However, on ordinary programs it typically finds parallelism in loops close the the innermost one—usually the next innermost. In practice this will not be good enough. Our conclusion is that parallelization technology will never achieve the level of success achieved by vectorization.

There are several reasons why automatic decomposition fails. Foremost among these is the inaccuracy of dependence analysis. Because compilers are required to preserve the meaning of the original program, they must be conservative in testing for dependence. Whenever the compiler cannot disprove the existence of some dependence, it must assume the dependence exists. In large loops, subscripts can be very complicated and often contain symbols whose values cannot be known at compile time. Unfortunately, it only takes one dependence to exclude a loop from parallel execution.

In one experiment, we analyzed a program from Los Alamos National Laboratory in which the programmer had discovered that a specific loop was parallel. The loop was a large one that contained numerous subroutine calls, so it looked like a good test for PFC. On the first trial, PFC generated approximately 2000 dependences for the loop. Stunned by this result, we analyzed the reasons why it failed and identified some sophisticated strategies for overcoming those. In fact, we simulated by hand the most advanced strategies we could conceive of using in a compiler system. The result was a reduction in the number of

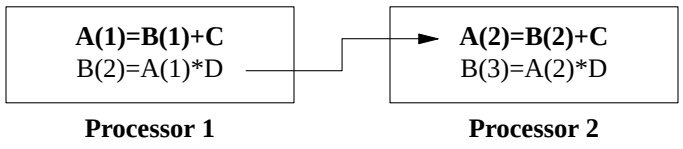


Figure 1. Effect of carried dependence.

dependences to 40. In other words, the best compiler we could envision would not identify that parallel loop!

As discouraging as they might seem, these results have *not* led us to abandon this line of research. Quite the contrary, we intend to continue investigating automatic translation because it can be a valuable tool in preparing a program for parallel execution. However, the results seem to indicate that we cannot rely on automatic techniques alone.

PFC now contains approximately 155,000 lines of PL/I and runs on IBM mainframes under VM/CMS. It currently generates PF (Parallel Fortran) a version of Fortran extended to contain parallel programming primitives[3]. It remains a valuable research resource because we can easily embed new techniques within it and evaluate their effectiveness. In addition, a modified PFC is used to support our research on program understanding tools described in the next section.

3. A Program Understanding Tool

Once it has been recognized that explicit parallel programming is necessary, it becomes important to consider debugging. The human programmer is more than likely to make an error in converting a program to parallel form and these errors are hard to locate because they are typically schedule-dependent. On the other hand, PFC is overly cautious and would never parallelize a loop that might carry a dependence. Thus, if an error occurs, we can assume that it is caused by some loop on which PFC and the programmer disagree. In converting the loop to execute in parallel, the programmer assumed there were no loop-carried dependences. The error was caused by some dependence that the programmer overlooked. A careful review of the dependences in the loop, aided by a powerful dependence analyzer like the one in PFC, might have prevented the error. This is the observation that led us to build PTOOL.

PTOOL is an interactive program understanding tool based on the analysis of dependence as implemented in PFC [ABKP 86]. In essence it is simply a sophisticated browser for a database of interstatement dependences created by PFC. As in PFC, the fundamental paradigm is loop parallelism. PTOOL permits the user to select a loop in a sequential FORTRAN program and ask whether or not the iterations may be run in parallel. If the answer is ‘‘no’’, PTOOL will display, in turn, all the dependences that prevent parallelization (in other words, all the dependences carried by the loop). In processing a loop, PTOOL determines the set of variables that may be declared private to each iteration. This is a a key feature in reducing the number of dependences displayed to the user, because no private variable can carry a dependence. In addition, since PFC employs interprocedural data-flow analysis to sharpen its approximation to the dependence graph, PTOOL can deal with calls inside loops.

As soon as the first version of PTOOL was complete, we invited a group of researchers from various laboratories, including Los Alamos, to a workshop on the system. Our goal was to foster the use PTOOL on real codes. After overcame the initial problems of getting the system to run on these codes, we asked the users for a critique of PTOOL. Their complaints fell into three categories.

1) PTOOL *is not truly interactive.* A naive programmer using the PTOOL browser often thinks he is in an editor. As soon as he sees how to eliminate a critical dependence he tries to make the change in the browser. Of course, PTOOL does not permit this because it would invalidate the map between the original source and the dependence data base. Hence, to see the effect of a change, the user must send the entire FORTRAN program back to the PTOOL analyzer to rebuild the dependence graph. To make PTOOL interactive, we would need to incorporate incremental dependence analysis.

2) PTOOL *cannot be used as a debugger.* We thought programmers would use PTOOL to understand their program before they began converting it to parallel form. This has not turned out to be the case. Instead, the programmer, filled with enthusiasm, converts it to run on a parallel machine and begins testing. As soon as an error occurs, the user wants help from PTOOL, but it is too late because the current system cannot read and analyze parallel programming extensions to FORTRAN. Even if you get the

[3] The PF compiler was developed by IBM for use on multiple-processor 3090 systems.

system to accept these extensions, it is not easy to analyze their effect. For example, PTOOL should not display any dependence that cannot be realized because the programmer has inserted appropriate synchronization. If a ''post'' and a ''wait'' on the same event make it impossible to take the endpoints of a dependence out of order, we should *not* display the dependence in PTOOL. The system needs to be expanded to handle these constructs.

3) PTOOL *displays too many false dependences.* On the example cited in the section on PFC, the system reported about 2000 dependences that did not exist in the program. This problem was, in large part, due to imprecise interprocedural information. We set out to improve this situation and devised two methods for sharpening the precision of the dependence graph. These methods are described in Section 5. We hand simulated their effect and discovered that the number of remaining dependences for the given loop was approximately 40. In addition, we added a feature that permits the programmer to eliminate whole classes of dependences from consideration. This feature permits a style of interaction in which the programmer looks at a specific dependence and then eliminates a whole class of similar dependences and then returns to consider the next dependence.

In Section 5, we will describe how these deficiencies are to be addressed in ParaScope.

PTOOL is installed and in use at Los Alamos National Laboratory [Hend 87] and the Cornell National Supercomputer Facility. There are browsers for IBM 3279 terminals, IBM personal computers and SUN workstations. In addition to its application to program understanding, it has proven to be an effective teaching tool—at the Cornell National Supercomputer Facility it is used in workshops on parallel programming to help students learn about data dependences.

4. An Environment for Interprocedural Analysis

To improve the precision of its dependence testing, PFC performs interprocedural analysis of the side effects of procedure calls. In addition, it propagates constants interprocedurally and analyzes aliases due to reference parameter passing. The problem with this is the organization of the compiling process. The current organization is depicted in Figure 2. PFC reads every routine in the program and gathers local information needed for interprocedural analysis, including the call sites in each procedure and the variables that are used and modified in the text. It then performs interprocedural analysis, propagating this information around the call graph until the global solution is determined. Finally, it rereads and analyzes every routine in the presence of the interprocedural information.

The problem with this scheme is that it is too costly to employ in a real compilation system. In response to any change, the system must read and reanalyze every subroutine in the program. The resulting time to rebuild a program will be intolerable, even to the most charitable of users. Our solution, represented by the system, is to embed the analysis process in a programming environment.

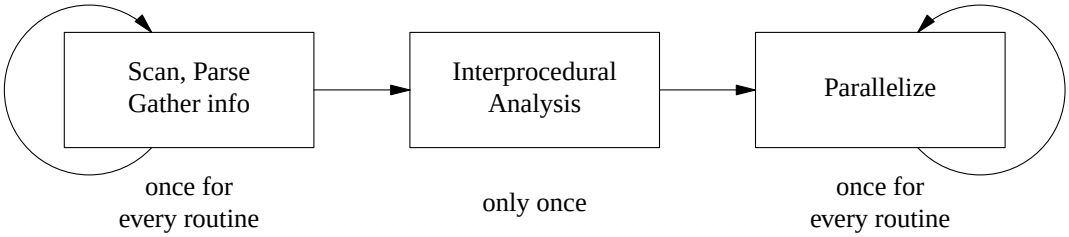


Figure 2. Interprocedural analysis in PFC.

On the surface, the programming environment is an integrated collection of tools designed to assist programmers building numerical software systems in FORTRAN [CCHK 87]. The environment provides balanced support for the programming process, incorporating tools to aid in the construction of whole programs as well as individual modules. There are five principal components in the environment.

- The *module editor* is an intelligent editor for program source.
- The *composition editor* is a tool by which collections of source modules can be defined to be a program.
- The *module compiler* is an optimizing FORTRAN compiler that uses interprocedural information.
- The *program compiler* is responsible for computing interprocedural data-flow information, performing interprocedural optimizations and rebuilding a correct executable after a change.
- The *execution monitor* runs programs and provides the debugging environment.

All of the tools interact with the user under the control of a window-based *monitor*. Information is passed among the various tools by saving it in a central database.

The compilation strategy is depicted in Figure 3. The local information gathered in the PFC scanner and parser is gathered by the source editor in and recorded in the database. The composition editor is used to define the precise set of modules that the program comprises. From the information provided by these two editors, the program compiler builds a call graph and solves for the interprocedural data-flow information. It then invokes the module compiler on each routine that needs to be recompiled. Since the interprocedural information is computed before any compilations are begun, there are no tricky compilation-order dependences. This is possible because the function of scanner and parser have been moved out of the compiler and into the editor.

The management of recompilation after a change presents some difficult problems. If a module is compiled in the presence of interprocedural information, its object code is a function not only of its source, but also of the source of the other routines in the program. This means that a change to one routine may force the recompilation of many routines in the same program. The program compiler pays special attention to the problem of limiting the number of recompilations by performing a detailed analysis of the effects of a change [CoKT 86].

The programming environment has grown to approximately 250,000 lines of C and has itself become a significant experimental resource. It is designed in a modular fashion to permit rapid prototyping

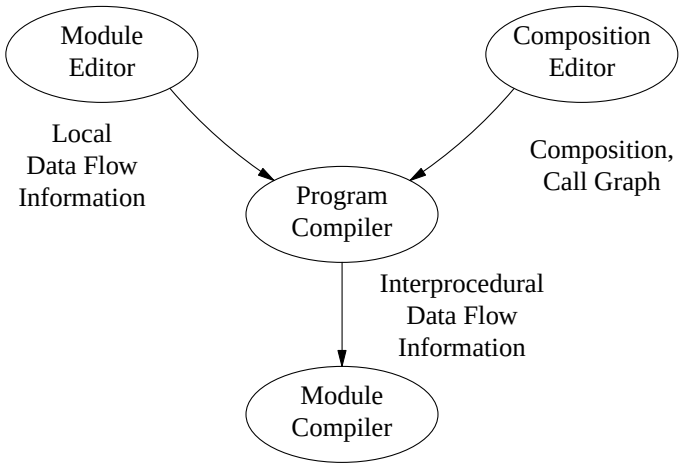


Figure 3. Interprocedural analysis in .

of experimental tools by providing them with a layer of environment services such as screen and database management.
runs under Berkeley UNIX 4.3 and X-Windows on SUN 3 and IBM RT PC workstations. A preliminary version of the system is scheduled for installation at the Cornell National Supercomputer Facility in early 1989.

5. ParaScope

Although the programming environment includes advanced tools for FORTRAN programming and performs interprocedural analysis, it lacks any support for explicit parallel programming. On the other hand, it provides a solid foundation on which to build new functions. By 1987, our work on PTOOL was at an impasse. Except for improvements in dependence testing, there was little we could do to overcome the weaknesses cited in Section 3 without embarking on a major effort. It seemed preferable to undertake such an effort in the context of a programming environment on workstations rather than continuing to develop system that ran only on simple terminals. Thus, the ParaScope project was begun.

The goal of the ParaScope project is to produce a programming environment that will assist in the formulation, implementation, and debugging of parallel FORTRAN programs. To accomplish this goal, we are enhancing in three important areas.

1) *Source Editor*. ParaScope includes a new version of the source editor that incorporates all the features of PTOOL while overcoming many of its weaknesses. In particular, it will incrementally reconstruct dependences after arbitrary editing changes and will be able to analyze common parallel constructs.
2) *Compiling System*. The program compiler will be expanded to employ more powerful forms of interprocedural analysis and optimization, including various forms of linkage tailoring. At the same time it will optimize the use of the memory hierarchy through a cooperative effort between the program and module compiler. Finally, the module compiler will incorporate features of PFC for automatic decomposition.
3) *Debugging System*. The execution monitor will be extended to incorporate remote debugging and parallel debugging. Parallel debugging will be based on a hybrid strategy of static analysis and dynamic testing. In addition, new features for *performance visualization* will be included to make it easier to find and eliminate performance bottlenecks.

The following sections present details on each of these subprojects.

5.1. The ParaScope Editor

The key piece of analytical information used by systems that automatically detect parallelism is the *dependence graph* [KKLW 80, AlKe 87]. Our experience with PTOOL has convinced us of the value of presenting dependence-based information to programmers as an aid to understanding why particular loops will not run in parallel. However, as we pointed out in Section 3, PTOOL has its weaknesses.

The ParaScope Editor, PED, is PTOOL's logical successor. In its final form, PED will combine the editing features of capabilities of PTOOL. The resulting tool will allow the user to edit program source, to examine the dependence structure of the resulting code, to apply pre-packaged transformations to that code (or make normal editing changes), and to view the impact of those changes on the dependence structure. PED will provide the user with advice about why a compiler could not run particular parts of a procedure in parallel. In addition, it will handle all of the common constructs of parallel programming—parallel loops, parallel cases, critical regions, event synchronization—and display only those dependences that might cause an error. In other words, it will not show a dependence that must be satisfied because of explicit synchronization.

Incremental Dependence Analysis

To achieve its goals in the context of an editor, PED must be able to quickly reconstruct the dependence graph in response to both editing changes and the application of pre-packaged transformations. Experience with PFC suggests that a complete recalculation of the graph in response to each change would

be unacceptably slow. Thus, we are designing and building an incremental dependence analyzer for PED. There are three principal components in this analyzer.

- an incremental data-flow analyzer that determines which dependences must be re-tested because of program changes;
- efficient implementations of one or more of the standard array independence tests [AlKe 87, Wolf 82] and
- an efficient notification mechanism that allows each of the components involved in the analysis system to inform the others of relevant changes in the abstractions that they export.

To understand this organization, consider dependence testing as a two-phase process. The first phase determines which definition points in the program can affect each use or definition point in the program. This can be formulated as a classical data-flow analysis problem called *reaching definitions* [Kenn 81, Zade 84]. The second phase applied a dependence test between a definition and every use or definition it can reach[4].

For the flow analyzer, we will use one of the incremental algorithms found in the literature [Ryde 83, Zade 84, CaRy 86]. The principal challenge in the flow analyzer will be engineering the interfaces carefully so the algorithm fits conveniently into a real editor. The implementation of the array independence tests should be orthogonal to the rest of the design. The individual tests should function in either a batch or an incremental analyzer.

Change notification is perhaps the most subtle of the three problems. Before the incremental analyzers can obtain any sort of reasonable performance, they must have a method for informing one another of the location and scope of individual changes. This implies a common underlying topology for viewing the program; it may also require a substantial amount of analysis to understand the size of the affected area for a change.

In designing each of these components, we are attempting to cleanly separate the abstractions. We hope that this will produce an analyzer that is relatively easy to retarget to other intermediate program representations.

Transformations

There are numerous transformations that have proven useful in converting a sequential program to parallel form. We plan to support a rich collection of these transformations in PED, including loop interchange, loop distribution, loop fusion, scalar expansion, loop blocking, sequential-to-parallel loop conversion and numerous others.

Each transformation will be implemented as a menu option. When a particular transformation is selected, PED will tell the user whether it can establish that the transformation is correct and will provide an estimate of its profitability. If the user chooses to go ahead, PED will apply the transformation and rebuild the dependence graph. Whenever the user makes a transformation that the system cannot prove correct, PED will record information about the transformation for later use in debugging. For example, if the user converts a loop with carried dependences to parallel form, those carried dependences may be clues to errors that arise later. By making it easy to install new transformations, we expect that PED will become a vehicle for experimenting with such transformations.

Eventually, we intend to experiment with advice to the user. For example, there are a number of transformations known to that might be applied to make a particular loop more parallel. Instead of making the user guess the right one, PED might suggest strategies that it finds promising in the light of the dependence pattern arising in the loop.

Parallel Language Features

The challenge in properly handling parallel language features is to show only the dependences that are not protected by some synchronization mechanism. Parallel loops and cases present no problem for

[4] For technical reasons having to do with the variety of dependence classes, a similar analysis is performed on reaching uses.

dependence analysis. They can both be treated as if they were sequential constructs, since any dependence between two code segments to be run in parallel indicates the possibility of a schedule-dependent result. Hence, the strategy is to report as errors any dependences carried by a parallel loop or between different parallel cases.

Synchronization primitives are much more difficult to deal with. There are two types of synchronizations that are common in parallel programming. *Critical regions* are used to insure atomic access to some shared resource. In most parallel programs, the user would want to ignore any dependence that has endpoints in critical regions guarded by the same lock. Interprocedural analysis of this property presents some interesting challenges, but we have devised a simple solution that tags each dependence with the locks at its endpoints. Although it is a bit naive, the solution works well in practice [CaKe 87].

A harder problem is how to deal with *event* synchronization. Suppose we have two parallel cases. In one, a variable is assigned and then an event is posted; in the other, a wait for the same event is followed by a use of the same variable. A good debugging system should not report the dependence between the assignment and use because the event synchronization insures that the assignment and use always occur in the same order.

It can be shown that the problem of determining whether dependences are protected by event synchronization is NP-hard, even in a simple model where no loops are permitted [CaSu 88]. Fortunately, an approximate problem can be formulated as a system of data-flow equations which can be solved in polynomial time [CaSu 88]. We expect this solution to be very accurate in practice.

5.2. The Parallel Program Compiler

The program compiler was originally designed to support optimization of scalar operations. The problem of compiling for parallel machines is a much deeper one and hence requires more complex tools. We are planning enhancements to the program compiler in three basic areas. First, we will improve the precision of interprocedural analysis to provide sharper analysis in PED. Second, we will incorporate much more ambitious whole-program optimizations, especially those designed to help in the management of the memory hierarchy. Third, we will enhance the module compiler to generate parallel code for at least one target machine.

Interprocedural Analysis

Both and PFC incorporate interprocedural analysis of side effects. Our experience with using interprocedural summary information in a working system for detecting parallelism has shown that the granularity of conventional summary information is too coarse to allow effective detection of parallelism in loops that contain call sites. The problem is that the current analysis treats whole arrays as single units. Thus, it is able to determine whether an array is modified somewhere, but not whether those modifications are restricted to some subarray. This limitation is disastrous for parallel decomposition because the most effective way to parallelize a loop is through *data decomposition*, in which each parallel iteration works on a different subsection of a given array.

Hence, some mechanism for determining the subsections actually affected by interprocedural side effects is needed. Triolet has proposed a method that finds the convex hull of the set of array locations affected as a side effect of a procedure call [TrIF 86]. Unfortunately, this method is too expensive to use in a compiler. Therefore, we seek to achieve a more limited goal: to recognize some important special cases of array side effects. For example, it would be extremely useful if we were simply able to recognize when the modification of an array by a procedure call is limited to a single column or row of the array.

Fortunately, a generalization of the approach currently used to solve interprocedural data-flow analysis problems can be used to develop more precise information about side effects. We have shown that the fast interprocedural analysis techniques used in can be extended to work on lattice elements [CoKe 88]. These lattice elements, each representing a subsection of the underlying array, can be used to achieve a much more precise analysis of side effects to array variables. This idea is illustrated by the sample lattice for array A shown in Figure 4. The top element of the lattice represents ‘‘no access’’ to the array, while nodes at the second level represent accesses to a single element of the array. Note that I, J, and K are

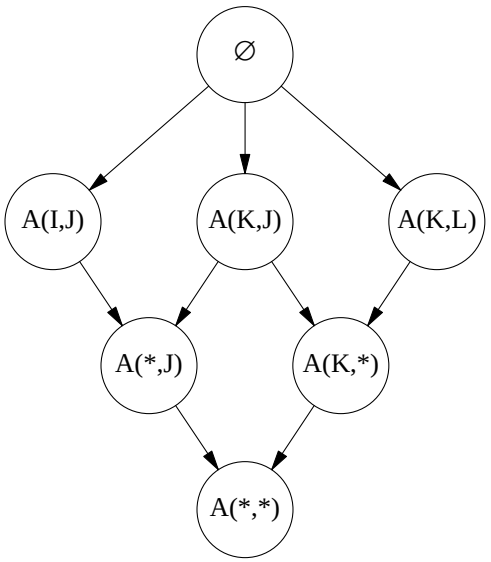


Figure 4. Simple side-effect lattice.

arbitrary symbolic input parameters to the call. The third level of the lattice represents accesses to a single column or row and the bottom element represents access to the whole array. Whenever two paths with access patterns represented by different lattice elements meet, the composite access pattern is approximated by the highest element of the lattice equal to or below each of the original elements. For example, if A(I,J) is accessed on one path and A(K,J) on the other, then the composite is A(*,J), the Jth column.

In his dissertation, Callahan proposes several regular section lattices [Call 87]. For ParaScope, we will enhance the interprocedural analysis to deal with lattice elements of this sort. We believe that this type of analysis will significantly enhance our ability to detect parallelism.

Another problem with the interprocedural analysis in PTOOL arose in the analysis of scalars. An important step in limiting the number of carried dependences is the determination of private variables. Any variable that is private to each iteration of the loop cannot carry a dependence. Consider the following example, which is an abstraction of something we have seen several times in practice.

```
DO I = 1, N
     CALL INIT(X)
     ...
     Y = X + 2
ENDDO
```

If we can determine that every path through INIT contains an assignment to X, then it can never be involved in a dependence carried by the loop. However, our original system only did ‘‘flow-insensitive’’ analysis; in other words, it was only able to determine that X *might* be assigned to in INIT. *In the absence of better information, the system must assume that it is defined on some iterations and not on others—hence, it can carry a value across iterations.*

To solve this problem we developed a new algorithm for the flow-sensitive problem of determining whether a variable is always assigned as a side effect of a procedure call [Call 88]. This method has been implemented in PTOOL and is being implemented in ParaScope.

Whole Program Planning

The program compiler is the vehicle responsible for planning optimization of a program on a global basis. In addition it must explicitly manage all interprocedural optimizations. As an example, consider *inline substitution*. When judiciously applied, this transformation can expose parallelism that is extremely hard to detect when subroutines are examined separately.

Unfortunately, inline substitution has several drawbacks: notably creation of much longer procedures and introduction of new recompilation dependences. Longer procedures are a concern because the analysis and transformation techniques used in compilers for parallel machines can require time that is quadratic in the length of the subroutine being compiled. Hence, inline substitution should be used sparingly lest we make recompilation times unacceptably long.

Another important optimization for parallel systems is efficient management of the memory hierarchy. If compiling systems are to free programmers from machine-dependent programming styles, they must do a good job of optimizing the flow of information through the memories of the target architecture. This problem cannot be solved by looking at one procedure at a time, it must be addressed on a whole program basis.

To address these problems, the ParaScope compiling system must plan the optimizing transformations, including parallelization, for the whole program. When complete, the ParaScope program compiler will perform the following sequence of operations:

1) construct the call graph;
2) perform interprocedural analysis;
3) gather information to support whole program optimizations such as inline substitution;
4) perform interprocedural optimizations and
5) invoke compilations of each module, passing information about the calling environment and side effects to the module compiler.

The implications of whole program planning in a programming environment are extremely complex. In ParaScope, planning will be the responsibility of the program compiler. To make the system work efficiently, we will need to develop incremental methods for revising plans in response to change, similar to the methods for limiting recompilation [CoKT 86] that we implemented in

The program compiler is also the natural place to identify large-grain task parallelism. By identifying individual routines or groups of routines whose executions can be run in parallel, the program compiler should be able to assign these tasks to separate processors. Very few compiling systems are able to deal with this type of parallelism.

Parallel Code Generation

To complete the task of preparing a program for execution in ParaScope, there must be a compiler that handles all the details of generating code. To this end, the optimizing module compiler will be extended to support generation of code for parallel supercomputers. Since the necessary dependence analysis will be performed in the editor and results stored in the database, the compiler can use this information in transforming and optimizing the program without directly paying the price for the analysis. This project will build upon our work on optimization and transformation of programs for parallel execution in PFC.

The optimizing compiler will benefit in a number of ways from the use of dependences, because they are more precise in their handling of subscripted variables than traditional data-flow information. Cytron has discussed a variety of interesting applications for such dependences [Cytr 86]. Allen and Kennedy [AlKe 86] have already shown that memory traffic for scalar computations can be reduced significantly by use of register allocation techniques based upon dependence. We also expect several other new scalar optimizations to present themselves.

The current plan is to support two different parallel machines: the IBM 3090 multiprocessor system and the Sequent Symmetry. This part of the project will be postponed until it becomes clear whether a standardized set of FORTRAN extensions will emerge from the activities of the Parallel Computer Forum [Leas 88].

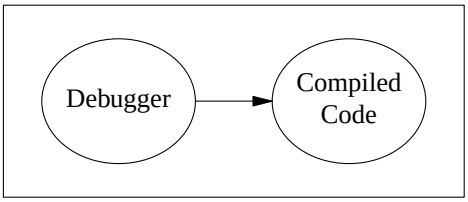
5.3. Parallel Debugging

Currently, the execution monitor (ExMon) supports debugging a sequential program on the local machine. In ParaScope, we will enhance it to support debugging programs executing on remote machines and to support debugging parallel programs. These extensions divide into two fundamentally different tasks: providing the mechanisms to control a process on a remote machine and examine or change its internal state, and developing paradigms to allow the programmer to relate source code constructs to the execution state of the parallel program. Both of these are necessary before a practical -style execution monitor can be constructed for parallel programs.

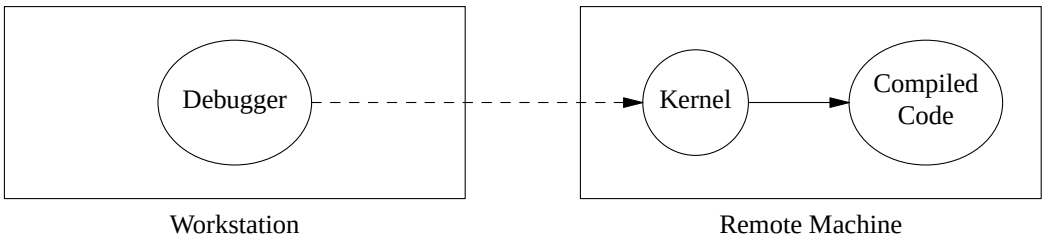
Remote Debugging

Fortunately, ExMon, the execution monitor, was designed to facilitate extensions to deal with processes on remote machines. As illustrated in Figure 5(a), it is implemented as two UNIX processes. The debugger itself, as well as all interpretive execution, runs in a foreground process. Whenever, compiled code is running, it runs in a separate process, with its own address space, in the background. Consistency of data values is insured by always keeping the correct values in the address space of the compiled process, using the UNIX system call *ptrace*.

Since *ptrace* is not implemented for processes running on different machines, a more complex arrangement is required to debug code compiled and running on a remote machine. As shown in Figure 5(b), this is handled by having the debugger communicate, using UNIX interprocess communication facilities, with a remote debugging kernel, which runs on the same machine as the compiled process and, hence, can use *ptrace*.



(a) ExMon



(b) Remote Debugging

Figure 5. Debugger organization.

Parallel programs involving multiple processes can be handled by allowing the debugger to manage more than one background process. Thus, with a remote debugging kernel supporting these operations via a remote procedure call mechanism, we expect to use ExMon running on a workstation to debug programs running on a remote supercomputer.

Relating State to Source

Most manufacturers are extending their sequential source debuggers to parallel systems by permitting the user to halt processes and single-step each process. Unfortunately, this approach doesn't provide a practical tool for debugging complex parallel programs. We intend to attack two of the problems that arise in dealing with the execution state of a parallel program: relating that state back to the user's source code in a meaningful way and providing tools to help the programmer cope with the nondeterministic nature of execution.

The first problem arises regularly in attempts to understand optimized code. The problem is made more acute by the radical transformations that a compiler attempts to discover additional parallelism. ParaScope will help the programmer understand the relationship between source and machine code by producing compiler generated annotations that can be examined in the source editor. One possible form for the annotations would be a transformed version of the source, in a pseudo-language that exposes constructs hidden in the original FORTRAN source. Other techniques will surely suggest themselves as the work on the display of annotations matures.

The second problem may well be the most difficult problem in debugging code for shared memory multiprocessors. Simply put, it is very hard to recreate with a debugger the sequence of events that leads to an error of unintentional data sharing. Although techniques based upon non-intrusive tracing show promise [GeHo 83], the hardware is not yet efficient enough to routinely support this approach. Semi-intrusive techniques generate trace information in software until an error occurs then back up to a point before the error and generate exhaustive traces [MiCh 88]. These approaches also show promise, but have serious performance drawbacks.

In ParaScope, we will use the information discovered during analysis and compilation to provide clues to the location of errors at run time. For example, if an incorrect value is detected in a parallel program at a point where the sequentially scheduled version produced the correct value, the debugging system would be invoked. It could trace back along dependence edges to locations inside parallel regions that are potential sources of the problem. Then, it could use *adversary scheduling*, a technique that employs dependence analysis to pick schedules likely to lead to errors, to locate the error. If the computation inside the parallel region was involved in a dependence that the compiler ignored under the programmer's direction, the system could generate a processor schedule that caused the dependence to be violated. In other words, the debugger would single-step the processors in a sequence likely to effect the value-passing pattern represented by the dependence.

Performance Visualization

Once the program produces correct answers, the programmer's attention turns to achieving maximum performance. Here the challenge is to provide tools that show the performance at a sufficient level of abstraction that the programmer can discern the patterns without being overwhelmed by details. Typically this is best done graphically.

The strategy that we will employ in ParaScope is illustrated in Figure 6. The idea is that a preprocessor analyzes the program to determine its structure, particularly the structure of parallelism, and inserts code to generate trace records when the program is run. Then the code is run using normal data to produce normal results. This will be slower than with no tracing, but hopefully the performance data will not be too far off. Finally, the trace file is analyzed by a post-viewer that graphically relates the timing information to the program structure, so that parallel execution can be displayed and bottlenecks discovered.

Several systems have already employed this strategy successfully, including *Schedule*, a system developed at Argonne National Laboratory [DoSo 86]. We are currently employing the same idea to analyze performance of the memory hierarchy on parallel machines.

6. Implementation Status and Future Plans

After approximately a year of effort, we have a solid prototype of the ParaScope editor that performs all the functions of PTOOL and implements most of the pre-packaged transformations. The completion of

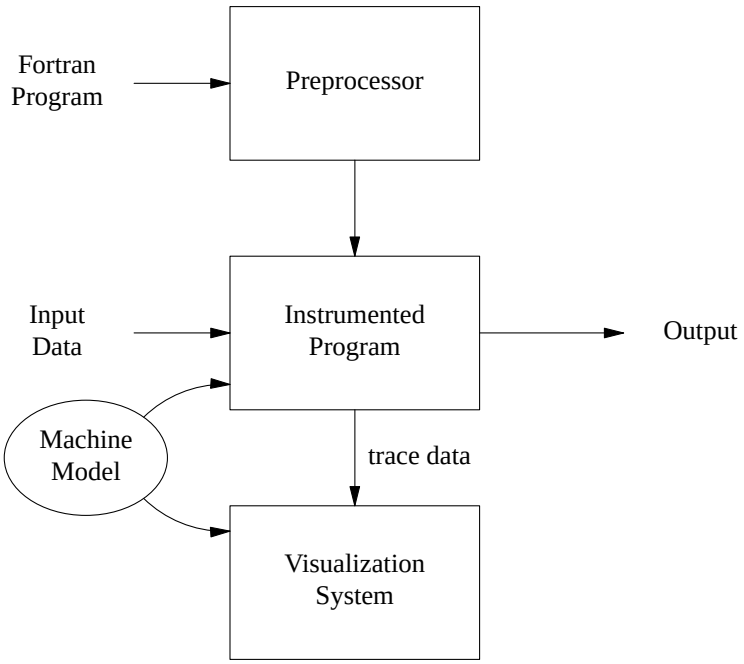


Figure 6. Performance visualization.

the project is waiting for the implementation of incremental dependence testing, which is in progress. Currently all dependence testing for PED is done by PFC.

We have implemented both the enhanced side-effect analysis and the flow-sensitive analysis in PFC, but not yet in ParaScope. The improvements in dependence analysis in both PTOOL and PED are dramatic. The program compiler has been equipped to do inline substitution, but none of the advanced features to support multiprocessor memory are implemented because we are waiting for our simulations to be finished before we select the transformations.

The execution monitor supports remote debugging and a simple form of adversary scheduling to find errors. There is still a significant amount of work to be done here. We also have a prototype visualization system for memory hierarchy. It is currently limited to two levels—memory and cache, but will be expanded in the near future.

Although we believe that cooperation among the tools of an integrated programming environment is the right concept upon which to base a programming environment, we will ultimately need to validate the concepts in ParaScope by putting the system into use by scientific researchers. Our plan is to target the first system for use on workstations in concert with the IBM 3090-600 at the Cornell National Supercomputer facility where PTOOL is installed. An internal system will support parallel programming for the 20-processor Sequent Symmetry at Rice.

An important question to be addressed is which dialect of parallel FORTRAN should we support. Currently we are using a subset of IBM PF (Parallel FORTRAN) which is implemented on the Cornell 3090. We hope that the current effort by the Parallel Computing Forum to standardize parallel extensions to FORTRAN for shared-memory machines will be successful [Leas 88] because that would make it possible to support a single language in ParaScope. If that happens, we will need to update the ParaScope module compiler and associated libraries to deal with that language.

Clearly, there is much work to be done before we achieve a satisfactory level of programming support for parallel computer systems. This should come as no surprise, since two previous generations of

supercomputers—the highly pipelined machines like the CDC 6600 and the vector machines like the Cray 1— each required a decade for compiler technology to catch up. Since parallel processing is arguably more complex than either of its predecessors, we can expect to do no better. It is our contention that a completely new breed of compilation system, based on programming environments, will be required to do an adequate job. In ParaScope, we are building what we hope will be the prototype for such compiling systems.

Bibliography

[ABCC 87] Allen, F., Burke, M., Charles, P., Cytron, R., and Ferrante, J. 1987. An overview of the PTRAN analysis system for multiprocessing. In *Supercomputing*, E.N. Houstis, T.S. Paptheodorou, and C.D. Polychronopoulos, eds. Lecture Notes in Computer Science 297. Berlin: Springer-Verlag, pp. 194-211.

[ABKP 86] Allen, J.R., Baumgartner, D., Kennedy, K., and Porterfield, A. 1986. PTOOL: a semi-automatic parallel programming assistant. *Proceedings of the 1986 International Conference on Parallel Processing*. Washington: IEEE Computer Society Press, pp. 164-170.

[AlCK 87] Allen, J.R., Callahan, D., and Kennedy, K. 1987. Automatic decomposition of scientific programs for parallel execution. *Conference Record, 14th ACM Symposium on Principles of Programming Languages*. pp. 63-76.

[AlKe 85] Allen, J.R. and Kennedy, K. 1985. A parallel programming environment. *IEEE Software* 2(4), pp. 21-29.

[AlKe 86] Allen, J.R. and Kennedy, K. 1986. Vector register allocation. TR-86-45. Houston: Rice University, Department of Computer Science.

[AlKe 87] Allen, J.R. and Kennedy, K. 1987. Automatic translation of FORTRAN programs to vector form. ACM TOPLAS 9(4), pp. 491-542.

[Call 87] Callahan, D. 1987. A global approach to the detection of parallelism. Ph.D. Thesis. Houston: Rice University, Department of Computer Science.

[Call 88] Callahan, D. 1988. The program summary graph and flow-sensitive interprocedural data flow analysis. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementations, SIGPLAN Notices* 23(7), pp. 47-56.

[CaKe 87] Callahan, D. and Kennedy, K. 1987. Analysis of interprocedural side effects in a parallel programming environment. In *Supercomputing*, E.N. Houstis, T.S. Paptheodorou, and C.D. Polychronopoulos, eds. Lecture Notes in Computer Science 297. Berlin: Springer-Verlag, pp. 138-171.

[CaSu 88] Callahan, D. and Subhlok, J. 1988. Static analysis of low-level synchronization. TR 88-70. Houston: Rice University, Department of Computer Science.

[CaRy 86] Carroll, M. and Ryder, B.G. 1986. An incremental algorithm for software analysis. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices* 22(1), pp. 171-179.

[CCHK 87] Carle, A., Cooper, K.D., Hood, R.T., Kennedy, K., Torczon, L. and Warren, S.K. 1987. A practical environment for scientific programming. IEEE Computer 20(11), pp. 75-89.

[CoKe 88] Cooper, K.D. and Kennedy, K. 1988. Interprocedural side-effect analysis in linear time. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation SIGPLAN Notices* 23(7), pp. 57-66.

[CoKT 86] Cooper, K.D., Kennedy, K., and Torczon, L. 1986. Interprocedural optimization: eliminating unnecessary recompilation. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices* 21(7), pp. 58-67.

[Cytr 86] Cytron, R. 1986. On the implications of parallel languages for compilers. Report RC 11723. Yorktown Heights, New York: IBM T.J. Watson Research Center.

[DoSo 86] Dongarra, J.J., and Sorensen, D.C. 1986. SCHEDULE: tools for developing and analyzing parallel FORTRAN programs. TR 86. Chicago: Argonne National Laboratory, Math and Computer Science Division.

[GeHo 83] Gentleman, W.M. and Hoeksma, H. 1983. Hardware assisted high level debugging. *Proceedings SIGSOFT/SIGPLAN Symposium on High Level Debugging, SIGPLAN Notices* 18(8), pp. 140-145.

[Hend 87] Henderson, L. 1987. The usefulness of dependency-analysis tools in parallel programming: experiences with PTOOL. TR Preprint LA-UR-87-3135. Los Alamos National Laboratory.

[KKLP 81] Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M. 1981. Compiler transformation of dependence graphs. *Conference Record, 8th ACM Symposium on Principles of Programming Languages*, pp. 207-218.

[KKLW 80] Kuck, D.J., Kuhn, R.H., Leasure, B., and Wolfe, M. 1980. The structure of an advanced vectorizer for pipelined processors. *Proceedings Fourth International Computer Software and Applications Conference*. Washington: IEEE Computer Society Press, pp. 709-715.

[Kenn 81] Kennedy, K. 1981. A survey of data flow analysis techniques. In *Program flow analysis*, Jones, N.D. and Muchnick, S.S., eds. Englewood Cliffs, NJ: Prentice-Hall, pp. 5-54.

[Kuck 78] Kuck, D.J. 1978. *The structure of computers and computations, vol. 1*. New York: Wiley and Sons.

[Leas 88] Leasure, B., et. al. 1988. PCF FORTRAN: language definition, version 1. Parallel Computing Forum draft document. Champaign, Il: Kuck and Associates, Inc.

[MiCh 88] Miller, B.P., and Choi, J.D. 1988. A mechanism for efficient debugging of parallel programs. *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 23(7), pp. 135-144.

[Ryde 83] Ryder, B.G. 1983. Incremental Data Flow Analysis. *Conference Record, 10th ACM Symposium on Principles of Programming Languages*, pp. 167-176.

[ScKo 86] Scarborough, R.G., and Kolsky, H.G. 1986. A vectorizing FORTRAN compiler. *IBM J. Research and Development* 30(2), pp. 163-172.

[TrIF 86] Triolet, R., Irigoin, F. and Feautrier, P. 1986 Direct parallelization of call statements. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices* 21(7), pp. 176-185.

[Wolf 82] Wolfe, M.J. 1982. Optimizing supercompilers for supercomputers. Ph.D. Dissertation. Champaign, Il: University of Illinois at Champaign-Urbana, Department of Computer Science.

[Zade 84] Zadeck, F.K. 1984. Incremental data flow analysis in a structured program editor. *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices* 19(6), pp. 132-143.