Parallel Programming Support in ParaScope

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

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

1. Introduction

It now seems clear that almost every new supercomputer design will employ some form of parallelism, because it offers the promise of higher execution speeds at reasonable costs. However, the burden of achieving that promise has been shifted to the programmer, who must decompose his problem into pieces that can be executed in parallel and ensure that the parallel processors assigned to these pieces are properly synchronized to produce a correct answer deterministically. This is not an easy task, because it requires not only an understanding of the problem being solved, but also of the underlying computer architecture and of the data-flow patterns in the program used to solve the problem. Clearly, sophisticated programming support tools are needed to assist in this process.

Typically, the process of parallel programming maps into four distinct tasks.

- *Decomposition*. The programmer must decompose the application problem into parallel tasks. This is most effectively done when the programmer can deal with the problem at a high level of abstraction.
- *Implementation*. The programmer must convert the abstract decomposition into a parallel program in some computer language. This task tends to be highly error-prone because it is fairly difficult to produce a program that precisely implements the vision of the parallel decomposition without making clerical errors.
- *Compilation*. The program must be translated into a machine-language version that can be executed on a parallel processor. This is typically the responsibility of the compiling system, although the sophistication of the compiler determines the extent to which the computer language can hide the details of parallel programming from the user.
- *Debugging*. The programmer, with the help of any available tools, must eliminate any logic errors that are discovered while testing the program. Parallel programming has given rise to a new class of bug: the *schedule-dependent* error. Schedule-dependent errors are difficult to locate because they are manifested only when certain schedules are employed in the parallel region. Frequently, the error disappears during debugging only to reappear when the program is put into production.

If we are to make parallel programming tractable, we must develop better tools in each of these areas.

ParaScope is a parallel programming environment under development at Rice University. It is designed to support the implementation, compilation and debugging phases in an integrated support system. In addition, it will also provide limited support for decomposition. When complete, it will consist of a collection of tools for elaborating a parallel program in an extended dialect of FORTRAN. The tools will include a sophisticated editor for FORTRAN source, an editor for defining whole programs, a powerful

compilation system that attempts to enhance the parallelism in the FORTRAN program through sophisticated transformation techniques, and an execution monitor that supports powerful techniques for debugging parallel programs. In addition, the source editor will have facilities to assist the programmer in decomposing a program for parallelism and to find clerical errors in a proposed decomposition.

ParaScope is the logical outgrowth of our research on scientific programming environments and our investigation of techniques for automatic detection of parallelism. Over the past four years, the $\mathbb{R}^n$ project at Rice University has been constructing a sophisticated programming environment for FORTRAN [HoKe 85, CoKT 86c, ChHo 87]. The current system consists of an integrated collection of tools to aid programming teams in entering and testing FORTRAN programs. It provides sophisticated facilities for managing all the code in the various components of the program. These tools provide the environment with enough control over the source of a program to attempt significant program optimizations that are not possible within a conventional compiler [CoKe 84, Coop 85, CCKT 86, CoKT 86b].

At the same time, the PFC project has been exploring methods for vectorization and automatic detection of parallelism [AlKe 84, AlKe 85, AlCK 87]. A significant subproject has developed PTOOL, a sophisticated browser that helps the user find parallelism inherent in an existing program [ABKP 86].

By combining technology developed in these projects, we expect to synthesize a useful tool for parallel programming. The central theme of this work is to develop ParaScope into an environment that runs on a high-performance workstation and assists in the preparation of programs intended for execution on a parallel-vector supercomputer like the IBM 3090 with vector feature.

The paper begins in Section 2 with a description of our past research on programming environments and automatic detection of parallelism. Section 3 describes our plans for ParaScope, including the six main research areas:

- incremental dependence analysis in a structure editor,
- improvements to the existing interprocedural analysis,
- techniques for debugging parallel programs,
- whole program planning for parallelization,
- code generation for parallel machines, and
- the display of complex program annotations.

Finally, Section 4 contains some concluding remarks.

2. Background

2.1. The $\mathbb{R}^n$ FORTRAN Programming Environment

The $\mathbb{R}^n$ programming environment is an integrated collection of tools designed to assist programmers building numerical software systems in FORTRAN [HoKe 85, CoKT 86c]. The environment provides balanced support for the programming process, incorporating tools to aid in the construction of whole programs as well as individual modules. Along these lines, it provides editors for the source code of a single routine, called a *module*, and for the structural description of a program, called a *composition*. It has two compilers, a *module compiler* that deals with intraprocedural optimization and code generation and a *program compiler* that deals with interprocedural issues. The $\mathbb{R}^n$ *execution monitor* supports execution of hybrid programs in which some modules are interpreted and others are executed from compiled code. 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. A brief description of the principal components follows:

Module Editor

The module editor combines a knowledge of FORTRAN with access to the database to help the user construct syntactically correct programs. It provides a convenient blend of structure editing and text editing, allowing the user to shift freely between these two paradigms. A module is defined by a set of entry point specifications; the different versions of a module all implement the same entry points. The editor directly constructs an *abstract syntax tree* representation of the module; this internal form for the program is used throughout the environment.

Composition Editor

In $\mathbb{R}^n$, both programs and composite modules are represented by their structural descriptions, called compositions [CKTW 86]. Compositions have hierarchical structure; they contain individual modules and other compositions. Import and export lists associated with each level of the composition provide a scoping and renaming facility. In essence, the $\mathbb{R}^n$ notion of a composition is itself a module interconnection language, albeit one with a screen-oriented concrete syntax. The user creates and modifies these descriptions with the composition editor, a structure editor for this interconnection language. The editor checks the program for consistency and completeness. It also ensures that entry point specifications match the call sites to which they are bound.

Program Compiler

The program compiler's task is to construct an executable image of a program that is both consistent with the current state of the source code for its components and fully optimized. When invoked on a specific program, it checks the records updated by the module editor and the composition editor to determine what changes have occurred since the last compilation. It uses this information to update the interprocedural summary, aliasing, and constant propagation information available for the program. Next, it performs a recompilation analysis to construct a complete list of procedures that must be recompiled. Finally, it looks at the set of procedures being recompiled to determine where interprocedural optimizations like linkage tailoring are likely to be profitable. Given all this information, it then invokes the module compiler to perform the needed recompilations [CoKT 86a, CKTW 86].

Module Compiler

The module compiler's task is to translate one or more modules into optimized code for the target machine, using interprocedural information and optimization directives provided by the module compiler. Essentially, the module compiler is an optimizing code generator that makes use of the information collected in the program compiler. It has been carefully structured to allow experimentation with different optimization and recompilation strategies.

Execution Monitor

The execution monitor allows the programmer to execute a program constructed by the environment. It provides the user with a range of instrumentation levels, including execution of hybrid programs in which some modules are run from compiled, optimized code and other modules are run interpretively [ChHo 87]. Instrumentation levels are dynamic; the programmer can change them whenever execution is paused at a breakpoint. Breakpoints can be specified in terms of either the program's composition or the source code for a specific module. We plan to support reversible execution; efficient implementation of this feature will rely on using the flow-insensitive interprocedural summary information computed by the rest of the environment.[1]

The $\mathbb{R}^n$ programming environment has grown to approximately 105,000 lines of C and has itself become a significant experimental resource. It is designed in a modular fashion to permit rapid prototyping of experimental tools by providing them with a layer of environment services such as screen and database

[1] This information is used when an interpreted subroutine makes a call to compiled code. Without the side-effect information, the interpreter would need to checkpoint the entire memory state accessible outside the calling routine both before and after the call and then compare the two images to determine what variables were changed by the call. With the information, it need only examine those variables contained in the interprocedural MOD set for the call site, a much less formidable task.

management. We expect to distribute an experimental version of the environment in late 1987.

2.2. PFC and PTOOL

Since 1978, Rice has been conducting an active program of research in software for vector and parallel supercomputers. PFC (the 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 has 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]

Since completion of the original PFC in 1982, research has continued in three areas. First, algorithms to perform interprocedural analysis on whole programs have been added. The approach used is based upon results from the batch version of the algorithm developed for $\mathbb{R}^n$. The aim of this work is to determine the impact of global program knowledge on vectorization and parallelization. Currently, PFC analyzes side effects of procedure calls, aliasing patterns, and constants propagated across procedure boundaries.

Second, we have been examining the use of PFC's analysis phase in parallel programming tools. Our first effort, called PTOOL, is an interactive adviser designed to assist in the prevention of errors arising from unintentional data sharing or unforeseen load-store orders for shared data in parallel programs [ABKP 86]. PTOOL is really a sophisticated browser for a database of interstatement dependences created by PFC. It 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 all the dependences that prevent parallelization.

While PTOOL is extremely helpful for identifying loops that can be parallelized, it provides no assistance in the generation of parallel code. A third project is investigating automatic generation of code for multiprocessors. This has led to another derivative of PFC, called PFC Plus, that not only recognizes loops whose iterations can be run in parallel but also performs sophisticated transformations, such as loop interchange, loop alignment, and code replication to enhance the parallelism available [AlKe 85].

PFC now contains approximately 95,000 lines of PL/I code and runs on an IBM 4341. PTOOL is already in use at Los Alamos National Laboratory and is scheduled to be installed at Livermore and Argonne. Both PTOOL and PFC Plus will be installed on the Cornell Theory Center's IBM 3090 supercomputer system. These systems represent a significant research resource because they are relatively easy to modify. This makes it possible to add and evaluate new transformation methods rapidly. In addition, PTOOL has value as an educational tool. It is now used to teach compiler students about dependences in programs.

3. ParaScope

[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.

It is the goal of the ParaScope project to produce a programming environment that will assist in the formulation, implementation, and debugging of parallel FORTRAN programs. In the environment we envision, a programmer will prepare a FORTRAN program (containing only standard FORTRAN constructs) using the environment's tools. The program might consist entirely of new code or it might incorporate packaged source modules. When the programmer is ready to consider the implications of parallelism, the system will bring to bear the analytical tools from the compiling system, reporting back on its successes and failures.

For those regions of the program that the compiler cannot run in parallel, the programmer can then examine the dependence information to see if a simple revision might permit more parallelism. At each step, the system will report success or failure and permit further investigation of dependences. When the programmer is satisfied, he can instruct the environment to generate code for the target machine and run the program.

Under this approach, program development is a cooperative effort by both the programmer and the system, exploiting the strengths of each. The system performs the tedious analysis of dependences, freeing the programmer to develop more highly parallel algorithms. Given feedback from the system, the programmer should be able to experiment with innovative approaches.

If successful, this project will produce a sophisticated programming system that not only helps the programmer produce correct, well organized code, but also assists him in tailoring that code to achieve high performance on parallel architectures. The following sections present details on the six subprojects we envision.

3.1. Dependence Analysis in a Structure Editor

In the automatic detection of parallelism, the key analytical information used by a compiler is a *dependence graph* [KKLW 80, Kenn 80, KKLP 81, Alle 83] for the procedure. To create a responsive tool for parallel programming, we will need to perform dependence analysis on the procedure as it is modified in the environment's structure editor for FORTRAN. While structure editing systems that incrementally update semantic information are reasonably common, we are aware of no system that maintains a consistent dependence graph of sufficient complexity to support the transformations required for parallel code generation.

Our experience with the PTOOL system has convinced us of the value of presenting dependence based information to programmers trying to understand why particular loops will not run in parallel. Unfortunately, because PTOOL relies on a batch analysis of the program, users have found the delay between typing a proposed change into the source and getting feedback about its impact to be frustrating. Constructing a useful, on-line parallel programming advisor will require a method for rapidly updating the dependence information for a procedure in response to one or more editing changes.

Given the need for incremental dependence analysis, a natural starting point for the research is to consider application of a general technique like an attribute grammar framework [Reps 82] or one of the existing incremental data-flow analysis frameworks [Zade 84, Ryde 83, CaRy 86]. Unfortunately, the computation of dependence analysis appears to be sufficiently complex to make re-casting it in an attribute grammar framework unbearably inefficient. Similarly, there appears to be no natural way to cast the problem in a data-flow analysis framework. Certainly, the resulting framework would be so complex that the standard algorithms from global data-flow analysis would not achieve their fast time bounds.

We believe that the speed of incremental updates to this information is critical to the responsiveness of the overall parallelism advisor. To achieve the goals of this project, we intend to develop efficient incremental updating techniques to deal with both factual and structural changes.

3.2. Improvements to Interprocedural Analysis

Our experience with interprocedural data-flow analysis in both $\mathbb{R}^n$ and PFC has convinced us that improvements are needed in both the theory and the implementation. Two particular problems present themselves: the need for more precise treatment of arrays and the need for incremental updating techniques. The next two subsections provide more detail on these problems.

3.2.1. Regular Section Analysis

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 it is modified in only a single column or row. This limitation is disastrous for parallelization 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. Although it may not be immediately clear from perusing the published papers [CoKe 84, CoKT 86c], the Cooper-Kennedy algorithm for summary problems can be extended to work on lattices. Consider the example lattice of reference patterns to the array A shown in Figure 1. Note that I, J, and K are arbitrary symbolic input parameters to the call. To incorporate a more accurate treatment of arrays, we must extend the side effect analysis to use

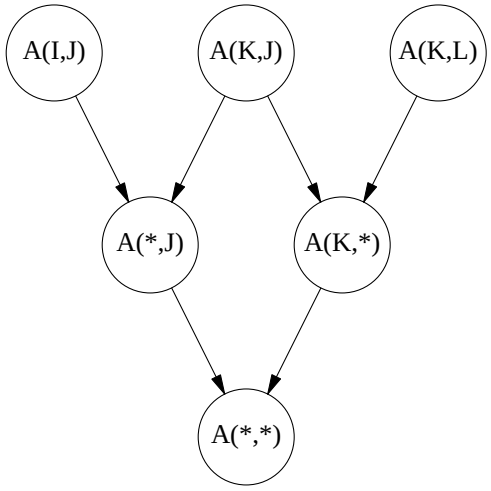

Figure 1. Simple side-effect lattice.

more general vectors of lattice elements rather than simple bit vectors.

Our technique for computing summary information is particularly well suited to this type of analysis because it divides the computation of interprocedural information between a step that traces the impact of chains of parameter binding and a step that uses this binding map to transform local information into the desired interprocedural information [CoKe 84]. Applying the method to a more complex domain of interprocedural facts doesn't change the first part of the analysis; only the second step requires modification. Since the first step requires time that is nearly linear in the size of the program, the cost of performing the meet operations in the second step may dominate the cost of the entire analysis. This makes choosing an appropriate lattice important, not only from the perspective of modeling facts that aid in the parallelization, but also because the lattice will directly impact the cost of performing the analysis.

In his dissertation, Callahan proposes several regular section lattices [Call 87]. As part of this research project, we will implement a version of the interprocedural summary algorithm that can operate on such lattices and use this to determine the efficacy of his various models. We believe that this type of analysis will significantly enhance our ability to detect parallelism.

3.2.2. Incremental Update Techniques

The current version of the $\mathbb{R}^n$ environment completely recomputes interprocedural information each time a program is compiled. The analysis is sufficiently fast that this has not been a major problem in the past. However, we expect that regular section analysis will require substantially more computing resources than the current techniques. This will increase the desirability of incremental methods for updating interprocedural information in response to a program change. We have identified several promising approaches for these updates [Coop 83, CoKe 84]. In implementing an update technique, we must examine not only our own work, but also the ongoing research into incremental updating techniques both at Rice and elsewhere [Reps 82, Zade 84, CaRy 86]. We plan to pick the most promising technique emerging from ongoing research on incremental analysis, implement it in ParaScope, and evaluate its efficacy. In performing this research, it will be important to evaluate the tradeoff between incremental and parallel evaluation methods. It seems likely that batch style algorithms may be more amenable than incremental techniques to parallel execution

3.3. Parallel Debugging

Currently, the $\mathbb{R}^n$ 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 $\mathbb{R}^n$-style execution monitor can be constructed for parallel programs.

3.3.1. Implementing Remote Debugging

Fortunately, the implementation of ExMon was designed in a manner that should facilitate extending it to deal with processes on remote machines. 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*.

To implement a version of ExMon that supports remote debugging will require implementation of a *remote debugging kernel* that supports the following operations:

- a combination of *fork*(2) and *execve*(2)
  The kernel must be able to start execution of a given file on the remote machine.
- *ptrace*(2)
  In order to read and write in the address space of the target program and to set breakpoints there, the remote kernel must provide a function equivalent to the UNIX *ptrace* system call.
- pause process
  The kernel needs some mechanism that will allow the debugger to pause a running target program.
- *wait*(2), *wait3*(2)
  In order to determine when the target program has stopped at a breakpoint, the remote kernel must support something similar to the UNIX *wait* and *wait3* calls. To avoid polling, the remote machine should asynchronously advise the debugger of a change of state in the child.
- get symbol table from executable
  In order to determine the location of various identifiers in the target program, the remote kernel must support reading the target program's symbol table.
- stack abstraction
  The kernel must provide operations that permit the run-time stack to be manipulated abstractly. This includes routines for addressing the various storage classes of identifiers as well as routines for pushing and popping stack frames.
- redirection of *stdin*, *stdout*, *stderr*
  In order to enter input data from the debugger's machine and to get the target program's output on it, the remote kernel must support some form of redirection of standard I/O.

An implementation of NFS on both the local and remote machine would simplify several problems, notably reading files on the remote machine and examining the target program's symbol table. 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.

3.3.2. 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 using the facilities discussed in Section 3.6. 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 great promise [GeHo 83], the hardware is not yet efficient enough to routinely support this approach. 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 step the processors in an order that is most likely to give rise to the value-passing pattern that caused static analysis to suggest that the sequential and parallel semantics differed.

3.4. Whole Program Planning

In a system for automatically decomposing a large program for parallel execution, optimizing transformations should be planned from a global perspective. For example, when choosing the loop to run in parallel from among several possibilities, it is desirable to select the outermost loop in order to generate less frequent synchronizations. However, if the program consists of many separately-compiled subroutines, the compiler will have trouble determining if a given loop is actually the outermost loop. Suppose that it confronts a loop within a subroutine and that the version of Parallel FORTRAN does not allow nested parallel DO's. In this case, the compiler should only select the loop if the subroutine is not called from within a parallel loop in some other routine. Determining when this is the case requires information about the whole program.

As a second example, consider inline substitution. When properly used, it can uncover parallelism that is very hard to see when subroutines are examined separately. Furthermore, it permits code from the substituted procedure to be tailored to the situation at the point of call, making more optimizations possible. This combination of effects can be extremely beneficial. However, unrestricted use of inline substitution can lead to an explosion in code size.

To make appropriate use of inline substitution, the compiling system must plan the optimizing transformations, including parallelization, for the whole program. The approach we envision constructs the call graph, gathers information about each procedure (including information for the inline substitution phase), computes interprocedural data-flow information and performs inline substitution. It then determines the important environmental information, like whether or not a call to the routine is contained in a parallel loop, to pass to the compiler for each individual compilation. Finally, it invokes compilations of the individual modules, passing the environmental data to the 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 we developed for limiting recompilation [CoKT 86b, Torc 85]. An important preliminary step to such algorithms is to implement recompilation limitation and gain experience with it.

The program planning scheme also appears to provide a natural place to identify larger granularity parallelism. It seems likely that looking for large grain parallelism on a whole program basis can lead to more effective use of a multiple processor machine by a single FORTRAN program. 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.

An important application of this research is to develop techniques for planning the optimizations of whole programs that are being prepared for execution on background supercomputers. ParaScope will include a powerful new program compiler that will manage programs on a background mainframe. The enhanced techniques should permit program compilation and module compilation to take place on the mainframe itself. Implementing this scheme will require cooperation with the mainframe FORTRAN compiler to achieve highly optimized whole programs.

3.5. 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 for the supercomputer. To this end, the $\mathbb{R}^n$ optimizing module compiler, currently in prototype form, will be completed and 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 would build upon our work on optimization and transformation of programs for parallel execution in PFC [AlKe 85, AlCK 87].

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 main problem to be attacked is the generation of parallel code for a supercomputer. Unlike PFC, this work would be in the context of a compiler. Hence, a number of machine-dependent issues can be explored. For example, what is the correct tradeoff between vector and parallel execution? Also, how can memory traffic be minimized for highest performance on a machine with a complex memory hierarchy?

If we attempt to isolate the transformations in one section of the compiler, we should also be able to produce a version of the compiler that generates FORTRAN annotated with parallel execution primitives. This would permit the resulting program to be compiled by the VS FORTRAN compiler for execution on the IBM 3090. If done properly, this goal would require that IBM make extensions to VS FORTRAN to accept the parallel primitives along with the interprocedural information produced by the program compiler.

The ParaScope optimizing module compiler should provide an ideal laboratory for investigating these and other issues related to compilation for parallel supercomputers.

3.6. Display of Complex Program Annotations

Designing a good user interface for ParaScope's sophisticated program analysis and transformation capabilities is a major challenge. $\mathbb{R}^n$ produces and consumes many kinds of information about FORTRAN programs in addition to source code; that information is voluminous, diverse in structure, and highly interrelated. Parallel programming features will add to this information considerably. The current version of $\mathbb{R}^n$ uses a graphical window system to display each kind of information in a separate window, with few

provisions for displaying inter-relationships. Up to now, the study of user interface issues has not been an explicit goal of the IR^n project.

If ParaScope is to become a useful tool for parallel programming, we need to develop mechanisms to help the ParaScope user deal with the complex web of information surrounding his FORTRAN program. Our approach is to view the information as a *document*, an ''electronic book'' written in the dynamic medium of computer graphics instead of on paper [KayG 77, YMvD 85, WeyB 85]. We think of all the extra information as *annotations* of the FORTRAN source text, analogous to a book's footnotes, figures, and appendices. Unlike a paper book, portions of the annotated FORTRAN can be selectively revealed or concealed, viewed in a choice of different formats, or edited. There has been much research on such dynamic media, but little of it has dealt with computer programs and none that we know of has focused on the complex, inter-related information managed by a programming environment.

3.6.1. Annotations

Some of the information managed by ParaScope comes from the programmer, and some is produced by ParaScope as a product of program analysis or transformation. The programmer provides four kinds of information to ParaScope. *Project management information* includes ownership of versions and revision histories of modules. *Programming information* includes formal specifications, design documentation, source code modules, and compositions of modules into programs. *Compiling information* includes such choices as where to perform in-line substitutions, what target machine to assume, and which modules to run interpretively. *Testing information* includes input test data, expected output, and debugging session logs.

ParaScope provides information to the programmer as a result of analyzing, transforming, or executing FORTRAN source. Conventional analysis finds syntactic and semantic errors in FORTRAN source code, connects declarations and uses of identifiers, and determines variables aliased by common or equivalence statements. Data-flow analysis finds constant valued expressions, connects stores with loads, and determines what dependences exist between statements or expressions. Transformation exposes parallelism at the source code level. In addition to producing normal program output, execution can produce other important information: suspended execution contexts, traces of adversary schedules, and performance measurements.

3.6.2. Source Code Enhancement

To display these annotations, we will explore techniques from four areas of related research: structured documents [DKLM 84, YMvD 85], hypertext [Carm 69, Nels 74], Knuth's WEB system of structured documentation [Knut 84], and graphical programming [Raed 85, Reis 84, MorH 85]. We will begin by applying some of these techniques in isolation, using the current IR^n system as a test bed. Our eventual aim is to find ways of unifying these techniques into a simple but powerful model for annotated FORTRAN.

In the first phase of research, we will use the existing IR^n user interface as a test bed to explore ways of presenting annotations. Later, we will build an entirely new user interface based on the unifying notion of annotated documents. One benefit of this research would be to ensure that ParaScope is easy to use when it is finished.

4. Conclusions

The first vector supercomputers appeared on the market in the early to mid seventies. Yet, because of the lag in developing supporting software, it is only recently that vectorizing compilers powerful enough to effectively utilize vector hardware have been developed.

Since parallel programming is a much more complex task than vectorization, we expect the challenge of producing adequate programming support to be much greater. In the ParaScope project, we will be exploring the leverage to be gained through an integrated collection of tools in which each tool depends on the others for important information. For example, the editor will depend on the interprocedural analyzer, which itself depends on the results of editing other modules. The debugger uses dependence information to assist in the location of potential problems. The user interface permits abstract displays of the data-flow information within a program.

We believe that it is essential to have this sort of cooperation to provide adequate support for programming on the evolving class of highly parallel machines.

Bibliography

[Alle 83] J.R. Allen, ''Dependence analysis for subscripted variables and its application to program transformations,'' Ph.D. dissertation, Department of Mathematical Sciences, Rice University, Houston, Texas, April, 1983.

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

[AlCK 87] J.R. Allen, D. Callahan and K. Kennedy, ''Automatic decomposition of scientific programs for parallel execution,'' *Conf. Record, 14th POPL*, January, 1987.

[AlKe 84] J.R. Allen and K. Kennedy, ''PFC: a program to convert FORTRAN to parallel form,'' *Supercomputers: Design and Applications*, K. Hwang, ed., IEEE Computer Society Press, Silver Spring, MD, 1984, 186-203.

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

[AlKe 86] J.R. Allen and K. Kennedy, ''Vector register allocation'', Technical Report 86-45, Rice University, Department of Computer Science, December, 1986.

[CCKT 86] D. Callahan, K.D. Cooper, K. Kennedy, and L. Torczon, ''Interprocedural constant propagation'', *Proc. SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), July, 1986.

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

[Carm 69] S. Carmody et al., ''A hypertext editing system for the /360'', *Pertinent Concepts in Computer Graphics*, M. Faiman and J. Nievergelt (eds.), Univ. of Illinois Press, 1969, 291-330..

[CaRy 86] M. Carroll and B.G. Ryder, ''An incremental algorithm for software analysis'', *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices* 22(1), January, 1987.

[ChHo 87] B.B. Chase and R.T. Hood, ''Selective interpretation as a technique for debugging computationally intensive programs'', *Proc. SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, June, 1987.

[Coop 83] K.D. Cooper, ''Interprocedural Data Flow Analysis in a Programming Environment,'' Ph.D. Dissertation, Department of Mathematical Sciences, Rice University, May 1983.

[Coop 85] K.D. Cooper, ''Analyzing aliases of reference formal parameters'' *Conf. Record, 12th POPL*, January, 1985.

[CoKe 84] K.D. Cooper and K. Kennedy, ''Efficient Computation of Flow Insensitive Interprocedural Summary Information,'' *Proc. SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices* 19(6), June 1984.

[CoKT 86a] K.D. Cooper, K. Kennedy, and L. Torczon, ''Optimization of compiled code in the $\mathbb{R}^n$ programming environment'', *Proc. of the 19th Annual Hawaii International Conference on System Sciences*, 1986.

[CoKT 86b] K.D. Cooper, K. Kennedy, and L. Torczon, ''Interprocedural Optimization: Eliminating Unnecessary Recompilation'', *Proc. SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), July 1986.

[CoKT 86c] K.D. Cooper, K. Kennedy, and L. Torczon, ''The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment'', *ACM TOPLAS* 8(4), October, 1986.

[CKTW 86] K.D. Cooper, K. Kennedy, L. Torczon, A. Weingarten, and M. Wolcott, ‘‘Editing and compiling whole programs’’, *Proc. of the Second SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments*, *SIGPLAN Notices* 22(1), January, 1987.

[Cytr 86] R. Cytron, ‘‘On the implications of parallel languages for compilers’’, Technical Report RC 11723, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., 1986.

[DKLM 84] V. Donzeau-Gouge, G. Kahn, B. Lang, and B. Melese, ‘‘Document structure and modularity in Mentor’’, *Proc. SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, 1984, 141-148.

[GeHo 83] W.M. Gentleman and H. Hoeksma, ‘‘Hardware assisted high level debugging’’, *Proc. SIGSOFT/SIGPLAN Symposium on High Level Debugging*, *SIGPLAN Notices* 18(8), August, 1983.

[HoKe 85] R.T. Hood and K. Kennedy, ‘‘A Programming Environment for FORTRAN’’, *Proc. of the 18th Annual Hawaii International Conference on System Sciences*, 1985.

[KKLP 81] D.J. Kuck, R.H. Kuhn, B. Leasure, D.A. Padua, and M. Wolfe, ‘‘Compiler transformation of dependence graphs,’’ *Conf. Record, 8th POPL*, January, 1981.

[KKLW 80] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe, ‘‘The structure of an advanced vectorizer for pipelined processors,’’ *Proc. IEEE Computer Society Fourth International Computer Software and Applications Conf.*, IEEE, October 1980.

[KayG 77] A. Kay and A. Goldberg, ‘‘Personal dynamic media’’, *IEEE Computer*, 10(3), 1977.

[Kenn 80] K. Kennedy, ‘‘Automatic translation of FORTRAN programs to vector form,’’ Rice Technical Report 476-029-4, Rice University, October 1980.

[Knut 84] D.E. Knuth, ‘‘Literate programming’’, *Computer Journal*, 10(3), 1977, 31-41.

[Kuck 78] D.J. Kuck, *The Structure of Computers and Computations, Vol. 1*, Wiley, New York, NY, 1978.

[MorH 85] M. Moriconi and D.F. Hare, ‘‘PegaSys: a system for graphical explanation of program designs’’, *Proc. SIGPLAN ‘85 Symposium on Language Issues in Programming Environments*, June 1985, 148-160.

[Nels 74] T.H. Nelson, *Dream Machines*. Published by the author, 1974.

[Raed 85] G. Raeder, ‘‘A survey of current graphical programming techniques’’, *IEEE Computer*, 18(8), 1985, 11-25.

[Reis 84] S.P. Reiss, ‘‘PECAN: program development systems that support multiple views’’, *Proc. Seventh Int’l Conf. Software Engineering*, March 1984, 324-333.

[Reps 82] T. Reps, ‘‘Optimal-time incremental semantic analysis for syntax-directed editors’’ *Conf. Record, 9th POPL*, January, 1982.

[Ryde 83] B.G. Ryder, ‘‘Incremental Data Flow Analysis’’, *Conf. Record, 10th POPL*, January, 1983.

[ScKo 86] R.G. Scarborough and H.G. Kolsky, ‘‘A vectorizing FORTRAN compiler,’’ *IBM J. Research and Development* 30(2), March, 1986.

[Torc 85] L. Torczon. ‘‘Compilation dependences in an ambitious optimizing compiler,’’ Ph.D. Dissertation, Department of Computer Science, Rice University, Houston, Texas, May, 1985.

[TrIF 86] R. Triolet, F. Irigoin, and P. Feautrier, ‘‘Direct parallelization of call statements’’, *Proc. SIGPLAN ‘86 Symposium on Compiler Construction*, *SIGPLAN Notices* 21(7), July, 1986.

[WeyB 85] S. Weyer and A. Borning, ‘‘A prototype electronic encyclopedia’’, *ACM Trans. Office Information Systems*, 3(1), 1985, 63-88.

[YMvD 85] N. Yankelovich, N. Meyrowitz, and A. van Dam, ‘‘Reading and writing the electronic book’’, *IEEE Computer*, 18(10), 1985, 15-29.

[Zade 84] F.K. Zadeck, ‘‘Incremental data flow analysis in a structured program editor’’, *Proc. SIGPLAN ‘84 Symposium on Compiler Construction, SIGPLAN Notices* 19(6), June, 1984.