Programming Language Support for Supercomputers¹

Robert T. Hood
Ken Kennedy

Rice MASC TR83-23

Department of Mathematical Sciences
Rice University
P.O. Box 1982
Houston, TX 77251

October, 1983

¹Support for this work was provided by the National Science Foundation under grants MCS-8104006 and MCS-8121884 and by IBM Corporation.

Programming Language Support for Supercomputers¹

Robert T. Hood
Ken Kennedy

Department of Mathematical Sciences
Rice University
Houston, Texas 77251

Abstract

To date, programming language research has made several major contributions to the support of high speed computing. First, highly optimizing compilers have permitted the programmer to write in a high level language without sacrificing much of the efficiency that could be gained by programming in assembly language for the particular target machine. Beginning with the remarkable Fortran I system, the quality of a Fortran compiler has been measured in terms of its ability to produce excellent code for its target machine. This has led to compilers that analyze a program globally and produce suprisingly accurate knowledge about the expected behavior of the program at run time.

Second, new features have been introduced into high level languages to facilitate the coding of scientific problem solutions while exploiting the special features of the underlying hardware. Examples are the array notations in APL and Fortran 8x.

Third, specialized analysis techniques to precisely determine the dependences inherent in a program have led to vectorizing translators that discover most of the vector operations inherent in a sequential program.

This paper surveys these contributions and discusses the challenge presented by parallel processor supercomputers and the emergence of program analysis tools based on a new paradigm -- the interactive programming environment. It concludes that techniques from program flow analysis and optimization can be adapted for use in an interactive problem-solving environment that provides a powerful collection of tools by which to exploit the capabilities of a supercomputer.

1. Introduction

From the very first automatic compiler project, efficiency of the executed code has been an extremely important consideration. In his reflections on the project at the History of Programming Languages

¹Support for this work was provided by the National Science Foundation under grants MCS-8104006 and MCS-8121884 and by IBM Corporation.

Conference in June 1978, John Backus reflected on the motivation for much of the Fortran I compiler design:

It was our belief that if FORTRAN, during its first months, were to translate any reasonable "scientific" source program into an object program only half as fast as its hand coded counterpart, then acceptance of our system would be in serious danger. ... To this day I believe that our emphasis on object program efficiency rather than on language design was basically correct. I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed. In fact, I believe that we are in a similar, but unrecognized, situation today: in spite of all the fuss that has been made over myriad language details, current conventional languages are still very weak programming aids, and far more powerful languages would be in use today if anyone had found a way to make them run with adequate efficiency [Back 78].

In this passage, Backus clearly reflects the concern that it may not be possible to compile a high level language into code which is comparable in efficiency to code that can be produced by a competent assembly language programmer.

Since we cannot expect that modern scientific applications will be programmed in assembly language, the concern for the machine designer is that all the advantages of a clever machine architecture may be voided by a poor compiler that is unable to take advantage of them. From a different viewpoint, the potential performance upper bound of a machine means nothing if that upper bound cannot be achieved or even approached in code generated from the standard language compilers. This is a sobering thought to most machine architects, which probably explains why so few of them seem to think about the problems of compiling to their architectures.

The fact of the matter is, that optimizing compilers are hard to write and hard to adapt to radical new machine organizations. In the words of Neil Lincoln:

There was, in 1964, no great body of theoretical work to guide compiler developers in dealing with the multiplicity of CDC 6600 functional units and its many asymmetrical registers. After nearly ten years, a truly optimum match between language features, compiler, object code reliability and machine organization was finally achieved. To expect less of an effort for similar results with new supercomputer schemes is fatuous [Linc 77].

Extending this observation to tomorrow's supercomputers, it is clear that a software initiative will be needed if such machines are really to achieve their potential.

In this paper, we speculate on how some of the current trends in the programming systems and programming language compilers may evolve into programming systems for supercomputers -- systems that will be well suited to help the programmer take advantage of the performance of parallel machine architectures while not sacrificing the convenience of high level language programming. Section 2 contains a survey of the standard techniques used in optimizing compilers, and includes an example of how they are used to support machine-dependent optimization. Section 3 surveys the existing techniques for automatic vectorization while Section 4 discusses how these techniques might be extended to detecting parallelism for multiprocessor-based supercomputers. Section 4 also discusses the the problems which must be solved if we are to be successful. Section 5 introduces the programming environment, a new model for programming language systems in which all the tools for program preparation and testing -- the editor, the debugger, the compiler, and the file system -- are cognizant of the specific programming language being used and work together to assist the programmer in his or her task. It is shown that such an environment offers clean and effective solutions to some of the problems from Section 4. Finally, Section 6 speculates on extensions to current programming environments which may be particularly useful for programming supercomputers.

The conclusion is not surprising: current work in programming support software can be naturally extended to effectively support supercomputers but there is much to be done if we are to make that come about.

2. Optimizing Compiler Technology

The problem with any machine architecture is that there are special features which the assembly language programmer will naturally take advantage of. However, the programming language compiler must be adapted so that it makes use of these features in the code for every program it generates. The compiler writer is faced with the task of reasoning about the code that will be generated when various input

programs are presented. The optimizing compiler must be able to analyze the whole program and tailor the generated code to the special features of the machine.

In order to do this, the compiler must thoroughly analyze the program to discover enough about it to transform it into efficient code. The analysis is usually partitioned into two phases. First, the control flow analysis phase subdivides the program into basic blocks of straight-line code and builds a directed graph representing program flow, called the control flow graph, in which vertices represent the basic blocks and edges represent the possible control flow transfers. The control flow graph is then used by the data flow analysis phase as an auxiliary data structure to help determine the data relationships in the program.

As an example, consider live analysis, the determination at each point in the program of which variables may be used again. The standard approach is to determine which variables are live at the beginning of each basic block, and propagate this information forward to the instructions contained in that block.

A variable A is said to be live on entry to block x if there exists a control flow path, free of any redefinition of A, from the beginning of block x to some block y that contains a use of A before any redefinition of A. Let $LIVE(x)$ be the set of all variables that are live on entry to block x. We can compute $LIVE(x)$ as follows. Suppose we have, for each block y, the sets:

$IN(y)$ -- the set of all variables for which there is a use in y prior to any definition in y.

$THRU(y)$ -- the set of all variables for which there is no assignment in block y.

These two quantities can be determined for any given block by simply scanning the block. There are two ways that a variable A can be live on entry to block x.

- (1) Block x can have a use of A that is not preceded by a definition, i.e.

$$A \in \text{IN}(x)$$

- (2) Block x has no definition of A, and A is live at some block y to which control may transfer after x is executed, i.e.

$$A \in \text{THRU}(x) \cap \text{LIVE}(y)$$

These observations lead to a system of equations describing the LIVE sets.

$$\text{LIVE}(x) = \text{IN}(x) \cup \bigcup_{y \in S(x)} (\text{THRU}(x) \cap \text{LIVE}(y))$$

Systems like these can be solved using any of a number of efficient techniques [Kenn 81].

A particularly useful data flow relationship is captured by use-definition chains, which link instructions that use a variable with the instructions that might compute the value used. Use-definition chains can be applied to the problem of dead code elimination by marking all instructions that are known to be useful (e.g., output instructions) and then tracing back on use-definition chains to mark new instructions that compute values used by instructions that have already been marked. When there are no more instructions that can be marked, the unmarked instructions are useless.

Use-definition chains can be computed by solving a set of data flow equations to determine the set of defining instructions whose result may reach a given block x without being redefined. Once this information is available for block entries, the construction of use-definition chains is straightforward.

The usefulness of global data flow analysis is illustrated by considering an example. The register allocation problem is to determine the best assignment of variables to registers, where "best" means the one that produces the most efficient program. Because the cost of memory references is high relative to the time required to get an operand from a cpu register, register allocation is an extremely

important machine dependent optimization. Unfortunately, the problem of finding an optimal register allocation scheme is NP-complete in all but the simplest of cases [Seth 75, BruS 76]. Thus, heuristic methods must be used.

A particularly good register allocation scheme is employed in the PL/8 optimizing compiler at IBM Research in Yorktown Heights, New York. The approach is as follows:

- (1) Generate low-level machine code for the target machine (a reduced instruction set computer) using an infinite number of symbolic registers. In other words, do everything normally done in code generation except register assignment.
- (2) Perform live analysis on the symbolic registers.
- (3) Build an interference graph reflecting which symbolic registers cannot be assigned to the same real register. Two variables cannot occupy the same register if one is live at a definition point for the other. In the interference graph, each vertex represents a symbolic register and each edge connects two symbolic registers that cannot share a real register.
- (4) Using a heuristic algorithm, attempt to n-color the interference graph, where n is the number of available registers. If this succeeds, we are done -- all variables have been assigned to registers. Otherwise, we must find the regions with high register requirements and generate instructions to write certain symbolic registers to memory to reduce those requirements. This process is known as spilling.

This method is incorporated into an experimental compiler and is extremely effective in allocating scalar variables to registers [CACC 81, Chai 82].

The main problem with the data flow analysis techniques used in most optimizing compilers is that they do not attempt a systematic analysis of array subscripts. Instead any use or definition of a subscripted variable is treated as a use or definition of the whole array. This oversimplification makes it extremely difficult to adapt these

techniques to support vector machines, because on such systems it is important to determine when loads and stores from arrays access the same elements. Thus new techniques were needed.

3. Software Support for Vector Computers

A more precise treatment of array dependences can be obtained by observing that most subscripted code occurs within DO-loops. The indexes in DO-loops follow a regular pattern. We can use that regularity to analyze dependence within arrays.

Many investigators have examined the problem of array dependence in loops [Coha 73, Mysz 78, Highb 79], but perhaps the most complete job has been done by Kuck and his colleagues at the University of Illinois [Kuck 77, Kuck 78, KKLW 80, KKLP 81].

Consider two statements in a program, such as the ones shown below:

```
DO 100 I = 1, 50
S1:  A(I) = X(I) * 10
      .
      .
      .
S2:  B(I) = A(I-1) + 3
100 CONTINUE
```

We say that statement S₂ depends upon statement S₁ if:

- (1) S₂ can be executed after S₁, and
- (2) On some execution, S₂ uses as input a value that has been computed by a previous execution of S₁.

There exist reasonably precise tests for this condition, such as Banerjee's test [Bane 76].

How can this form of dependence help us to vectorize programs? Consider the following loop.

```
DO 100 I = 1, 100
  X(I) = X(I) + 10
100 CONTINUE
```

This loop can be directly converted to a vector statement in Fortran 8x or some other language supporting vector notation.

$X(1:100) = X(1:100) + 10$

However, if we consider a slight variation on this loop:

```
DO 100 I = 1, 100
  X(I+1) = X(I) + 10
100 CONTINUE
```

we cannot directly transliterate to the corresponding vector notation:

$X(2:101) = X(1:100) + 10$

This is because the semantics of Fortran 8x require that the vector statement behave as if all quantities on the right hand side of the statement are extracted from memory before any stores occur on the left hand side. These semantics reflect the reality of vector machine instructions which attempt to gain speed by streaming data from memory -- hence the loads cannot wait on the stores.

The reason that the transliteration cannot work is because the instance of the statement on the second iteration of the loop uses a value computed on the first iteration. Hence the statement depends upon itself. No statement that depends upon itself either directly or indirectly can be transliterated to vector form.

This idea can be extended to a general procedure for vectorization [Kuck 78]:

- (1) Build a dependence graph, using a precise test for dependence such as Banerjee's, in which each use-def link is treated as an edge from definition to use.
- (2) Find all strongly-connected regions in this graph.
- (3) Any statement that is not in a strongly-connected region may be vectorized directly.

An extension to these ideas can greatly increase the amount of vectorization possible in a given program. It is possible to connect each dependence edge with the DO loop that gives rise to it. To understand this, consider the following example.

```

        DO 200 I = 1, 100
          DO 100 J = 1, 100
            S1:    A(I,J)  = B(I,J) + C(I,J)
            S2:    B(I+1,J) = A(I,J) * D(J)
          100 CONTINUE
        200 CONTINUE

```

The two statements in the inner loop form a recurrence. S_2 depends on S_1 in a way that is independent of any loop because it uses $A(I,J)$, which is textually identical to the output of S_1 . On the other hand, S_1 depends on S_2 because it uses a value in the array B that was created on the previous iteration of the loop on I. Hence this dependence is tied to the variation of the outer loop. If we hold that loop constant, the recurrence does not exist.

Thus, we can vectorize both statements in the example if we run the outer loop sequentially:

```

        DO 200 I = 1, 100
          A(I,1:100) = B(I,1:100) + C(I,1:100)
          B(I+1,1:100) = A(I,1:100) * D(1:100)
        200 CONTINUE

```

The recurrence is broken because we do not attempt to run in parallel the loop that gives rise to one of the essential dependences.

This concept, known as layered dependence is at the heart of the successful vectorizer we have produced at Rice University [AllK 80]. The highly effective Parafrase Compiler at the University of Illinois [KKLW 80] incorporates a similar idea.

Another important transformation from the point of view of vectorization is loop interchange. This is perhaps best illustrated by an example.

```

        DO 200 I = 1, 100
          DO 100 J = 1, 100
            A(I,J+1) = A(I,J)*B(K,J)
          100 CONTINUE
        200 CONTINUE

```

This code cannot be vectorized, because the statement depends upon itself by a dependence edge associated with the inner loop. However, if we interchange loops to make the loop on I the inner loop, the recurrence will be associated with the outer loop and we can vectorize as follows:

```

      DO 200 J = 1, 100
        A(1:100,J+1) = A(1:100,J)*B(K,J)
200  CONTINUE

```

Both vectorizers mentioned above incorporate very aggressive loop interchange transformations.

In the next section we will discuss how this technology can be applied to detecting parallelism for multiple-processor supercomputers.

4. Extension to Parallel Computers

Let us consider the problem of automatically detecting sections of code that can run on multiple processors without complex synchronization. As a base, we will assume that we are dealing with a parallel machine with a shared memory and local caches. In such an organization, it is extremely efficient to run different iterations of the same loop on different processors so long as one processor does not store into any location used by another processor. In other words, if there is no dependence between different iterations of the same loop, we can achieve maximum parallelism by loading each of the processor caches with the data needed by a particular iteration and running each iteration in parallel. If we know that no dependence exists, we need not worry about synchronization.

To illustrate this, consider the following example which applies a rotation to a matrix A.

```

      NM1 = N - 1
      DO 200 J = 1, NM1
        DO 100 I = 1, M
          TEMP = COS*A(I,J) - SIN*A(I,N)
          A(I,N) = SIN*A(I,J) + COS*A(I,N)
          A(I,J) = TEMP
100    CONTINUE
200  CONTINUE

```

If each processor has its own copy of TEMP, the iterations of the inner loop can be run in parallel because the array accesses in each iteration are to different rows. However, the outer loop on J cannot be run in parallel because the element A(I,N) is computed on one iteration of the outer loop and used on another.

Clearly, the sort of analysis done as a part of vectorization can be useful for this. We can use the ideas of layered dependence and loop interchange introduced in the previous section. If we can identify a loop that gives rise to no dependence cycles, the iterations of that loop can be run on different processors. For example, consider the following loop:

```

      DO 100 I = 1, 100
        DO 50 J = 1, 100
          A(J+1,I) = A(J,I) + B(J)
        50  CONTINUE
      100 CONTINUE

```

Here there is a recurrence carried by the loop on index J, but none carried by the loop on I. Hence, we can run the different iterations of the loop on I in parallel, because they have no store conflicts. In parallel loop notation, this would be written:

```

      DOALL 100 I = 1, 100
        DO 50 J = 1, 100
          A(J+1,I) = A(J,I) + B(J)
        50  CONTINUE
      100 CONTINUE

```

where the DOALL indicates that the outer loop may be run in parallel.

It should be noted that there is a difference between the kinds of transformations one does to support vectorization and the kind used to support multi-processor parallelism. In the example above, the vectorizer would move the recurrence to the outer loop so that the inner loop could be vectorized. However, in a multi-processor system, one assumes that the overhead for scheduling is non-trivial, so that it pays to get more code into the body of the loop being run in parallel. Hence, for the multi-processor, the parallel loop is moved to the outside. Applying this idea to the example at the beginning of this section, we would interchange the two loops and run the outer one in parallel to yield the following.

```

      NML = N - 1
      DOALL 200 I = 1, M
        DO 100 J = 1, NML
          TEMP(I) = COS*A(I,J) - SIN*A(I,N)
          A(I,N) = SIN*A(I,J) + COS*A(I,N)
          A(I,J) = TEMP(I)
100    CONTINUE
200 CONTINUE

```

Now instead of having each processor running three Fortran statements, we have each processor running a whole loop.

In general, it is desirable to move as much code as possible into the body of a parallel loop to insure that the multi-processor system functions efficiently. However, this leads to another difficulty. Suppose there are externally compiled functions or procedures in the body of a loop we wish to run in parallel, as in the example below.

```

      DO 100 I = 1, 100
        A(I) = USERFN (A(I), C)
100 CONTINUE

```

In the absence of better information, we must assume that any location in A might be changed by the call to USERFN, especially if A is in COMMON storage. Without some information about what happens inside of USERFN, we must assume that we cannot safely run this loop in parallel. This difficulty is not as serious for the vectorizer as the following example will illustrate:

```

      DO 100 I = 1, 100
        A(I) = B(I) * 10
        A(I) = USERFN (A(I), C)
100 CONTINUE

```

If B is not in COMMON, the first statement can be vectorized to yield the following code:

```

      A(1:100) = B(1:100) * 10
      DO 100 I = 1, 100
        A(I) = USERFN (A(I), C)
100 CONTINUE

```

Here at least some vectorization has been achieved, but for the multi-processor case, single-statement parallelism may not be sufficient to compensate for the scheduling overhead.

These problems and others have convinced us that, if language systems for parallel computers are to be truly effective, they must

incorporate some sort of interprocedural analysis. But one major reason why few compilers incorporating such analysis have appeared is the reluctance of the typical Fortran programmer to turn over all the modules of a program to the compiler at once. Usually, such programs are developed incrementally, a module at a time, and the cost of recompiling the entire program at once would be intolerable.

In the next section we will discuss a new type of programming support system in which interprocedural analysis can be done naturally and efficiently.

5. The Programming Environment

How can we do an effective job of interprocedural analysis without requiring that the whole program be compiled together? The answer is to provide a system which will automatically keep track of all modules in the program being developed and remember optimizing information from one compilation for use in another. Such a system, which would oversee all activities related to development of the program, would permit incremental development and compilation while making interprocedural optimization possible.

Over the past several years, programming language research has turned to a new paradigm in programming systems -- the programming environment. A programming environment is a collection of programming support tools that understand enough about the task at hand to give rather specific assistance to the programmer. Typically, these tools include a language-oriented editor, facilities for management of the program source, debugging tools tailored to the programming language being used, and facilities for making consistent executable modules. Examples of environments abound in the literature. The Cornell Program Synthesizer is an environment to support development of small PL/I programs by introductory programming students [TeiR 81]. Mentor, one of the earliest environment projects, supports programming in Pascal [Donz 75]. The Interlisp system contains many Lisp-specific tools to assist the programmer [Teit 77]. The Gandalf project at CMU is building an environment-generating system that can be used for any of a number of languages [HabN 82]. Finally, a group at GTE is working on a

programming environment for the Chill language [RudM 81].

A programming environment provides the ideal vehicle to support interprocedural optimization because the various tools can record information in the data base for later reference by other tools. If this approach is to be successful, the system must be so convenient to use that it is easier for the programmer to turn over all of his program for the system to manage than it is to manage them himself. In other words, by providing a rich system of programming assistance tools, we encourage the programmer let the system manage his program source. We are then free to record in a program data base whatever information we find useful for interprocedural optimization or other purposes.

As a part of the $\mathbb{R}^n$ project at Rice University, we have been developing a programming environment for Fortran which will support interprocedural analysis while providing a powerful collection of tools to the Fortran programmer. It is intended for use with Fortran systems for fairly conventional machines, but we will indicate how it can be very effectively used to support multiprocessor supercomputers.

5.1. Overview of the Environment

Our conceptual view of the environment is depicted in Figure 1. outset. First, it is major component of the $\mathbb{R}^n$ project at Rice University, which is building a network of high performance workstations designed to provide the modern scientist, engineer, or numerical analyst with a computational resource tailored to his needs. The specifications for such a workstation are summarized in Table 1.

- 1-2 mip CPU
- high speed floating point
- large virtual address space
- 1-2 megabytes real memory
- ≥ 20 megabytes local disk
- 800×1000 pixel bit-mapped display
- graphic input device (mouse or tablet)
- network interface
 - file servers
 - print servers
 - compute servers
 - gateways to other networks
- Unix software base
- Reasonable cost (\$5K-15K)

Table 1. Workstation Specifications.

Thus the environment is designed from the outset to run on such a workstation and to take advantage of the high resolution graphics, the graphical input device, and the local computational power. The environment may also run on simpler graphics devices, such as dumb terminals, but we have not compromised the design to accommodate such devices. We envision that the network will have other resources connected to it such as a computation server, a long-haul network gateway, and a file server.

The programming environment will be partitioned between the file server and the workstation as depicted in Figure 1. At the heart of the environment is the project data base which resides on the file server. It records all information about the programs and modules in a project, including source, specifications, test data, documentation, interprocedural information, and much more. All access to it is made through the project manager software, which stores and retrieves the source files in the project, answers questions about the programs and modules, insures that project rules are obeyed, and makes consistent versions of the programs for execution. The project manager stores information from and provides information to three main tools in the environment:

- (1) An intelligent Fortran editor which not only helps the programmer build syntactically correct programs, but also warns of possible run-time anomalies that can be detected at compile time.
- (2) A debugging interpreter which can step through parts of the program allowing the programmer to interrupt and monitor execution. With

Figure 1. The R^n Programming Environment for Fortran.

the help of the project manager, the interpreter would be able to handle a hybrid program consisting of some compiled and some interpreted modules. Thus control could be made to pass quickly through most of the program to the module under development which could then be interpreted.

- (3) An optimizing compiler that converts the partially compiled version of the program maintained by the editor to an optimized form suitable for integration with the rest of the system. With the help of the project manager, it uses the system data base to do a thorough job of interprocedural analysis and optimization.

These tools work together to assist the programmer in preparing, documenting, and testing the program. They also cooperate to make the final programs as efficient as possible. We will discuss each of them in more

detail.

5.2. Project Management

A project manager program will control access to all programs and modules within the project and maintain a data base of semantic information about them that can be used by other tools in the project. The basic framework is depicted in Figure 2. In this scheme, the largest entity is the project, which may be thought of as a collection of programs that are being worked on by a common pool of programmers and which may make use of a common group of subprogram modules. A project might contain one program that is the central focus of the work, along with a collection of test versions, or it might include several central programs.

A program is simply a collection of modules that, when integrated into a whole, may be executed. Several programs in a project may share the same modules, so the system must support some mechanism for sharing. A program may have several named versions, each using a different set of modules.

A program may be viewed as nothing more than a recipe indicating how to compose the modules (with versions specified) that it incorporates. Indeed, this is how programs are implemented in our preliminary system.

A module is a collection of entry points that is always edited and compiled as a whole. Modules may also have named versions. For example, there will usually be both an "official" and a "test" version of any module that is being modified. Presumably, the programmer will be working on the test version.

Associated with modules are various kinds of information, including source and compiled intermediate language. Each module also contains some number of entry points, which are the names by which the module is accessed externally. This will include all the callable entry point names. Associated with each entry point will be two kinds of information. Specifications are properties of the entry point which are entered by the programmer. In the first system, the specifications will

Figure 2. A conceptual framework for project management.

consist only of the number and types of parameters. Later this may include other information about the intended behavior of the entry. Annotations are facts about the behavior of the entry which are gleaned by any of the tools in the environment. For example, the editor might

add an annotation that indicates which other entries might be called as a result of a call to an entry point.

To conveniently provide the facilities of standard groups of modules, such as LINPACK or IMSL, the system will need to understand the concept of a library, which is an external project in which some modules have been declared to be "public." Single modules may be incorporated into a project from any of a number of specified libraries for the project.

Finally, any system that supports versions of its basic components must also support defaults. The environment we are developing would always have a standard version of every object for which versions are supported.

It is the role of the project manager to maintain the project data base, to maintain the consistency of programs within the project, and to provide information about the project or any of the programs and modules within it. For example, the manager must keep track of which programmers are working on which versions of a given module. Also, the manager will provide tools by which new programs can be constructed from modules in the data base. Finally, the project manager will provide the interface through which all queries about the project must pass.

To understand the role of the project manager, it is helpful to consider a selection of the functions we envision it performing. There are essentially three main functions performed by project management.

- (1) Query Answering -- in this category, we include any operation which provides information about the project in a non-destructive fashion. In other words any operation which does not cause a change in the current project state. Examples are requests to browse source modules, questions about specifications or annotations for a given entry point, and questions about the structure of a given program, such as a request to display the call graph.
- (2) Module Creation and Modification -- in this category are all operations on modules in the project that lead to new or changed modules being stored in the data base. Examples are requests to edit a

given module or to create a new one.

- (3) Program Creation -- in this category we find the function of program composition. In the system, programs are created by specifying a collection of modules to be incorporated in the composition. The project manager then adds enough modules to make a complete program or until it must report that the program is incomplete.

An issue related to program composition is the current context program. In the process of working on a project, we envision that the programmer will establish a program as the one in which he or she is currently working. The current context program establishes the default for many operations of the project manager. For example, when a query asks for information about a given entry point name, the project manager will assume that the query refers to the version of that entry point in the current context program. Similarly, in performing a composition, the program is completed by adding modules from the current context program.

The project manager will also be responsible for maintaining the project privilege rules. We use a very simple mechanism for deciding the authority of project programmers to perform certain functions. Each module and program has a creator, a status (public or private) and a reference count. A private program or module belongs to its creator and may be modified or released by that programmer. A program may be made public by its creator. By doing so, the creator relinquishes his or her authority over the program and every module contained in it. The creator may not modify a public module or program. He or she must create a new version of the module and build a whole new program composition in order to make such a change. Presumably, this composition will be private.

This mechanism insures the stability of public programs. Only the chief project programmer may release a public program; it then reverts to private status. Modules released by the chief project programmer revert to private status only when all references to it are by programs owned by the creator of the module.

5.3. Intelligent Editor

The intelligent Fortran editor will serve as the programmers home environment from which all activities are invoked. It will permit the programmer to browse through projects and programs; it will acquire modules and other information from the data base; and it will make heavy use of a sophisticated graphics interface featuring multiple windows, highlighting, structured region hiding, and the use of a pointing device such as a mouse.

The editor will be language-oriented and will assist the programmer in entering Fortran by providing commands that generate templates for the major language constructs. For example, to insert a DO-loop, the programmer need only invoke the DO-loop command and the cursor will be replaced by a DO-loop template with place markers in the positions where further text should be entered.

```
DO <iterator>
  <body>
REPEAT
```

The syntax displayed above is taken from Fortran 8x.

Not only does the editor help a programmer enter syntactically correct programs, it also obviates the need for a parser by directly constructing the abstract syntax tree for the program. All components of the environment can then use the abstract syntax tree as the standard program representation. The display is constructed by unparsing the abstract syntax tree. The abstract syntax permits the editor to provide both textual and treewalk cursor movement. Thus, there will be separate keys that allow the programmer to move quickly to subparts of a larger language construct. For example, if the cursor is positioned at an IF statement, there will be a cursor move function which will quickly move to the condition field.

The high-resolution display on the workstation permits a particularly convenient view of the program to be presented. Typically, the display will have three windows as in Figure 3. The main window will display the current region on the screen, with region hiding as appropriate. A parallel window will always display the current

Module: band-solve	Program: reservoir pde's	variables
<pre> subroutine bands1 (matrix, neq, width) { iwidth = width / 2 nqm = neq - 1 off = neq + 1 do i = 1 , nqm { ... do j = iplus1 , irange { ... if (temp ≠ 0.0) { noff = i do k = iplus1 , irange { noff = noff + off loff = noff + icount if (matrix[noff] ≠ 0.0) { ... } } } } ... } return } </pre>		<pre> i : integer icount : integer iplus1 : integer irange : integer iwidth : integer j : integer k : integer loff : integer matrix[neq] : double neq : integer noff : integer nqm : integer off : integer temp : double width : integer </pre>
insert delete info mark paste exit		

Figure 3. A typical editor display.

declarations for each variable used in the region of program on the screen. Finally, an option window will show the functions which are available at the current cursor position. For example, in Figure 3, the cursor is positioned at a statement location, so any statement may be inserted. An option may be selected using the mouse or by explicitly entering the command.

Note that the program is displayed in a notation that is somewhat different from that of standard Fortran. In an environment such as this one, the user will have the opportunity to tailor the display format to his taste. The program is shown in a format similar to the one used for

Ratfor or C on Unix² systems.

In our system the editor will also be able to detect and report many subtle semantic errors such as uninitialized variables. It will make use of information stored in the project data base to help construct subprograms that are consistent with the program being developed. For example, when a programmer wishes to insert a call to an external subroutine, the editor will query the data base to provide a template for the parameters that are required.

```
CALL S ( <integer ncases>, <real array x>, <real array y> )
```

In this statement, the programmer inserted CALL S and the system provided the parameter template.

There are several documentation functions that the editor will perform, including prompting the programmer for certain kinds of specifications, and maintaining a modification history. The editor will also compute and record summary data flow information for each module that it creates or modifies; such information can be used in both optimization and error detection.

In advanced versions of the editor, we will experiment with incremental data flow analysis. New results by Reps [Reps 82], Wegman [Wegm 82], and Zadeck [Zade 83], lead us to believe that use-definition chains (pointers from statements which use variables to the statements that might create the value used) can be efficiently created by the editor in an incremental fashion. If this is true it will be possible to provide some powerful diagnostic features.

For example, it will be possible to have a function which scrolls back from a usage point to successive points of definition for the value used (see Figure 4). This facility would be extremely useful in debugging because most errors are detected when a bad value causes some fault to occur. The point of fault is easily located. However, the real error probably occurred where the bad value was created. Use definition chains can help us quickly find all possible creation points.

²Unix is a trade mark of Western Electric Company.

	Def	Variable: iwidth	Use	Variable: iwidth
Def	<pre>read (5, 1000) nobs read (5, 1001) istart, iend read (5, 1002) a <u>iwidth</u> = <u>iend</u> - <u>istart</u> sum = 0.0 do i = 1, nobs { sum = sum + a[i] }</pre>			<pre>do i = 1, nobs { sum = 0.0 do j = 1, nobs { sum = sum + x[i]*a[j] } b[i] = <u>sum</u> / <u>iwidth</u> }</pre>
Use				

Figure 4. Scrolling back to definition points.

5.4. Debugging Interpreter

The debugging interpreter will enable the programmer to step through parts of a given program allowing him to interrupt and monitor execution.

The debugger, like the editor, will also make effective use of the high resolution graphics. As we envision it, the programmer will be able to monitor execution using a display similar to the one depicted in Figure 5. While highlighting the statement being executed in one window, the debugger will simultaneously display the changed values of variables in a second and the program output in a third window.

An important design goal is to support hybrid execution, in which compiled and interpreted modules are intermixed. This will permit interpretive testing of a module that may not be executed until many minutes of execution of the whole program. To support this feature, it is absolutely critical that compiled and interpreted programs maintain a consistent layout of data in the program. This strategy will also make it more likely that the compiled and interpreted versions of the same module will behave identically.

Module: fib	Program: fib	variables	output
<pre> program fibonacci { read n i = 0 current = 0 next = 1 print 'i', 'fib(i)' while (i ≠ n) { i = i + 1 print i, current <u>next</u> = <u>next</u> + <u>current</u> current = next - current } } </pre>		<div> <div>8</div>current </div> <div> <div>7</div>i </div> <div> <div>10</div>n </div> <div> <div>13</div>next </div>	<div> <div>i</div><div>fib(i)</div> </div> <div> <div>1</div><div>0</div> </div> <div> <div>2</div><div>1</div> </div> <div> <div>3</div><div>1</div> </div> <div> <div>4</div><div>2</div> </div> <div> <div>5</div><div>3</div> </div> <div> <div>6</div><div>5</div> </div> <div> <div>7</div><div>8</div> </div>
insert delete info mark paste exit			

Figure 5. Execution monitoring in the debugger.

Another important debugging feature we intend to support is reversible execution. There is no special difficulty to this. Since the abstract tree is doubly linked, we can easily move backward in it. A problem arises at three main points of ambiguity: assignments, gotos, and calls to compiled code. Traditionally, these are handled by saving on some file the value of the changed variable or the location from which control came. The big problem is caused by calls to compiled code. At these, the interpreter must save the values of every variable that might be changed before control returns. In the absence of better information, this means every variable in common and every parameter.

In the $\mathbb{R}^n$ environment, the burden will be much smaller because the interprocedural analysis will provide the interpreter which a much more precise estimate of what might be changed by a call. Thus many fewer variable values will need to be saved.

Debugging is an extremely important programming activity that has received too little attention from Fortran implementors. Here is an area where the graphic capabilities of the personal workstation will be

especially valuable. One aspect of numerical debugging is common to non-numeric debugging: the elimination of semantic errors in the program, errors that cause the program to behave in an incorrect manner. There is another type of debugging common in numerical programs -- elimination of the errors of precision and accuracy that make the answers incorrect or the algorithm fail to converge rapidly enough. It is here that debugging truly takes on the flavor of experimentation and the ability to interactively follow execution while monitoring the output may permit enormous savings of research time.

5.5. **Optimizing Compiler**

The optimizing compiler, which is really an optimizing code generator, will convert the partially compiled version of a module maintained by the editor to an optimized form suitable for integration into the program of which it is a part. The main advance in this tool over previous optimizing compilers for Fortran is its use of interprocedural analysis and optimization.

Compiler optimization researchers have long believed that the interprocedural effects are the last remaining major source of inefficiency in languages with optimizing compilers. Why then are there so few compilers with any interprocedural analysis and optimization? The answer is that the compiler would need access to all the code in a program in order to do a good job. It is unreasonable to expect to compile whole programs at once -- the cost in computation time would be too great. It would be almost as impractical to perform data flow analysis on the whole program at each module compilation.

The solution is to save the interprocedural information needed for optimization between compilations in the project data base. This requires that the interprocedural information be updated each time a module is edited.

We intend to use the environment to attack two problems. First, we will investigate the use of interprocedural information to do linkage tailoring -- the construction of efficient subroutine linkages tailored to the actual caller and callee. An example of linkage tailoring is

inline substitution, but there are many less dramatic forms.

A second area is to compute the patterns of data usage and definitions as a result of procedure invocations. An example is the computation of `mod(s)`, the set of variables that might be changed as a result of the procedure invocation at call site `s`. There are two components to this information.

- (1) First there are the immediate effects of the procedure being invoked. These can be recorded in the data base by the editor -- on putting a module away, the editor need only store the list of variables that are changed in some statement in the program.
- (2) To this list must be added the secondary effects due to calls to other routines from within the called routine. These must be handled by solving a data flow problem on the call graph [Alle 74, AllS 72, Bann 79, Bart 78, Myer 81, Rose 79, Spill 72, Weih 80]. A recent dissertation by Cooper [Coop 83] describes fast algorithms to solve this problem in an incremental fashion. The basic idea is that whenever the editor puts away a module that is incorporated in a program, a demon is invoked to update the interprocedural information. This demon makes use of comparisons of old information with new information to keep from doing redundant work.

As a result of the actions of the demon described above, several modules may need to be recompiled in the light of new interprocedural information.

Thus the environment will permit us to at last mount a concerted attack on interprocedural optimization and analysis.

6. Implications for Multiprocessor Supercomputers

As Section 2 indicated, interprocedural analysis will make it possible to do a much more aggressive job of analyzing vector dependences. This will in turn lead to compilers which will be able to construct very precise dependence information about arrays of the sort described in Section 3. Such a compiler might well attempt to automatically assign different iterations of a loop to different processors. It is conceivable that this might lead to parallel programs that are acceptably

7. Summary and Conclusions

It has become obvious to all those who program parallel computers that the task is a hard one. If the programmer is to do an adequate job, he must be able to factor out all but the most essential details. This implies that we must have strong programming support systems -- languages, compilers and environments -- that support high level constructs for parallel programming which are free of artificial constructs arising from the specifics of the underlying machine.

The other side of the coin is that such languages will need powerful optimizing compilers to tailor individual programs to the available hardware. We hope that we have convinced you that the technology of compiler optimization and vectorization is sufficiently advanced to expect that, in time, we will be able to develop those compilers. However, the need for concern with larger granularity of parallelism makes the need for interprocedural analysis and optimization even more pressing.

Programming environments, which have been the subject of much recent study, promise to provide a vehicle for increasing programmer productivity while permitting cross-procedural information passing among the tools of the environment. In particular, the tools of an environment can easily be adapted to aid in the process of interprocedural information gathering. The $\mathbb{R}^n$ programming environment project at Rice University is attempting to develop the techniques for interprocedural analysis in the context of an environment.

In addition to their other advantages, programming environments can be adapted to provide interactive programming support for parallel programming. It is easy to envision these environments as powerful programming laboratories for crafting supercomputer programs.

However, the outlook is not completely rosy. Work has only just begun on programming and much software remains to be developed. Only if we start to actively pursue the goal of a truly interactive parallel programming environment now will we be ready to support the multiprocessor supercomputers when they reach the market in the second half of this decade.

References

- [Albe 81] C.N. Alberga, A.L. Brown, G.B. Leeman, M. Mikelsons, and M. Wegman, "A program development tool," Conf. Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, 1981, 92-104.
- [Alle 74] F.E. Allen, "Interprocedural data flow analysis," Information Processing 74, North Holland Publishing, Amsterdam, 1974, 398-402.
- [AllK 82] J.R. Allen and K. Kennedy, "PFC: a program to convert Fortran to parallel form," Report MASC TR 82-6, Department of Mathematical Sciences, Rice University, Houston, Texas, March, 1982.
- [AllS 72] F.E. Allen and J.T. Schwartz, "Determining data relationships in a collection of programs," RC 4989, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., August 1974.
- [ANSI 81] American National Standards Institute, Inc., "Proposals approved for Fortran 8x," X3J3/S6.80 (preliminary document), November 30, 1981.
- [Back 78] J. Backus, "The history of FORTRAN I, II, and III," ACM Sigplan Notices 13, 8, August 1978, 165-180.
- [Bane 76] U. Banerjee, "Data dependence in ordinary programs," Report 76-837, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, November 1976.
- [Bann 79] J.P. Banning, "An efficient way to find the side effects of procedure calls and the aliases of variables," Conf. Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, January 1979, 29-41.
- [Bart 78] J.M. Barth, "A practical interprocedural data flow algorithm," Comm. ACM 21, 9, September 1978, 724-736.
- [BruS 76] J.L. Bruno and R. Sethi, "Code generation for a one register machine," J. ACM 23, 3, 1976, 502-510.
- [CDKJ 82] R.S. Cartwright, J.E. Dennis, J.R. Jump, and K. Kennedy, " R^n : an experimental computer network to support numerical computation," TR 82-5, Department of Mathematical Sciences, Rice University, Houston, Texas, 1982.
- [CACC 81] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, P.W. Markstein, "Register allocation via coloring," Computer Languages 6, 1981, 47-57.
- [Chai 82] G.J. Chaitin, "Register allocation and spilling via graph coloring," SIGPLAN Notices 17, 6, June 1982, 98-105.
- [Coha 73] W.L. Cohagen, "Vector optimization for the ASC," Proc. Seventh Annual Princeton Conference on Information Sciences and Systems, Dept. of Electrical Engineering, Princeton, New Jersey, 1973, 169-174.

- [Coop 83] K.D. Cooper, "Interprocedural Data Flow Analysis in a Programming Environment," Ph.D. Dissertation, Department of Mathematical Sciences, Rice University, May 1983.
- [DeRT 81] A. Demers, T. Reps, and T. Teitelbaum, "Incremental evaluation for attribute grammars with application to syntax-directed editors," Conf. Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, 1981, 105-116.
- [Donz 75] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J.J. Levy, "A structure oriented program editor: a first step toward computer assisted programming," International Computing Symposium 1975, North-Holland Publishing Company, 1975, 113-120.
- [Feld 77] S.I. Feldman, "Make -- a program for maintaining computer programs," Software -- Practice and Experience 9, 1977, 255-265.
- [Fos0 76] L.D. Fosdick and L.J. Osterweil, "Data flow analysis in software reliability," Computing Surveys 8, 3, September 1976, 305-330.
- [Habe 82] A.N. Habermann, computer science colloquium, Rice University, Houston, Tx., April 1982.
- [HabN 82] A.N. Habermann and D.S. Notkin, "The Gandalf software development environment," research report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., January 1982.
- [Higb 79] L. Higbee, "Vectorization and conversion of FORTRAN programs for the Cray-1 CFT compiler," Publication 2240207, Cray Research, Inc., Mendota Heights, Minn, June 1979.
- [Kenn 80] K. Kennedy, "Automatic translation of Fortran programs to vector form," Rice Technical Report 476-029-4, Rice University, October 1980.
- [Kenn 81] K. Kennedy, "A Survey of Data Flow Analysis Techniques," Program Flow Analysis: Theory and Applications (S.S. Muchnick and N.D. Jones, editors), Prentice-Hall, New Jersey, 1981, 1-54.
- [Kuck 77] D.J. Kuck, "A survey of parallel machine organization and programming," Computing Surveys 9, 1, March 1977, 29-59.
- [Kuck 78] D.J. Kuck, The Structure of Computers and Computations Volume 1, John Wiley and Sons, New York, 1978.
- [KKLP 81] D.J. Kuck, R.H. Kuhn, B. Leasure, D.A. Padua, and M. Wolfe, "Compiler transformation of dependence graphs," Conf. Record of the Eighth ACM Symposium on Principles of Programming Languages, Williamsburg, Va., January 1981.
- [KKLM 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.

- [Linc 77] N.R. Lincoln, "Its really not as much fun building a super-computer as it is simply inventing one," High Speed Computer and Algorithm Organization, (D.J. Kuck, D. Lawrie, and A. Sameh, editors), Academic Press, New York, 1977.
- [Lome 77] D.B. Lomet, "Data flow analysis in the presence of procedure calls," IBM J. Research and Development 21, 6, November 1977, 559-571.
- [Masi 80] L.M. Masinter, "Global program analysis in an interactive environment," SSL-80-1, Xerox PARC, Palo Alto, Calif., Jan. 1980.
- [MikW 80] M. Mikelsons and M.N.Wegman, "PDELL: the PL/1 program development environment principles of operation," RC 8513, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., Nov. 1980.
- [Myer 81] E.W. Myers, "A precise inter-procedural data flow algorithm," Conf. Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, 1981, 219-230.
- [Mysz 78] M. Myszewski, "The Vectorizer System: current and proposed capabilities," Report CA-17809-1511 Massachusetts Computer Associates, Inc., Wakefield, Massachusetts, September 1978.
- [Oste 81a] L. Osterweil, "Using data flow tools in software engineering," Program Flow Analysis: Theory and Applications (S.S. Muchnick and N.D. Jones, editors), Prentice-Hall, New Jersey, 1981, 237-263.
- [Oste 81b] L. Osterweil, "TOOLPACK architectural design," Toolpack Ref. LO-10304, University of Colorado at Boulder, March 1981.
- [PinS 73] J.H. Pinc and E.J. Schweppe, "A Fortran language anticipation and prompting system," Proc. ACM Nat. Conf., Atlanta, Georgia, 1973.
- [Reps 82] T. Reps, "Optimal-time incremental semantic analysis for syntax-directed editors" Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, 1982, 169 - 176.
- [Rose 79] B.K. Rosen, "Data flow analysis for procedural languages," J. ACM 26, 2, April 1979, 322-344.
- [Rose 81] B.K. Rosen, "Linear cost is sometimes quadratic," Conf. Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, 1981, 117-124.
- [RudM 81] A. Rudmic, B. Moore, "The Chill compiling system: towards a Chill programming environment", Proceedings of the Fourth International Conference of Software Engineering for Telecommunications Software, 1981, 187 - 190.
- [Russ 78] R.M. Russell, "The CRAY-1 computer system," Comm. ACM 21, 1, January 1978, 63-72.

- [Seth 75] R. Sethi, "Complete register allocation problems," SIAM J. Computing 4, 3, 1975, 226-248.
- [Site 78] R.L. Sites, "An analysis of the Cray-1 computer," Proc. Fifth Annual Symposium on Computer Architecture, April 1978, 101-106.
- [ShaP 81] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," Program Flow Analysis: Theory and Applications (S.S. Muchnick and N.D. Jones, editors), Prentice-Hall, New Jersey, 1981, 189-233.
- [Spil 72] T.C. Spillman, "Exposing side effects in a PL/I optimizing compiler," Information Processing 71, North Holland Publishing, Amsterdam, 1972, 376-381.
- [TeiR 81] R.T. Teitelbaum, and T. Reps, "The Cornell program synthesizer: a syntax-directed programming environment," Comm. ACM 24, 9, September 1981, 563-573.
- [Teit 77] W. Teitelman, "A display-oriented programmer's assistant," Proc. Fifth Int. Joint Conf. Artificial Intelligence, 1977, 905-915.
- [Wegm 80] M.N. Wegman, "Parsing for a structural editor," Proc. Twenty-first Annual Symposium on Foundations of Computer Science, IEEE Computer Society, 1980.
- [Wegm 82] M.N. Wegman, "Summarizing Graphs by Regular Expressions", RC 9364 (41252), IBM Research, Yorktown Heights, N.Y., April 1982.
- [Weih 80] W.E. Weihl, "Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables," Conf. Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, 1980, 83-94.
- [Zade 83] F.K. Zadeck, "Incremental data flow analysis in a structured program editor", dissertation, Department of Mathematical Sciences, Rice University, August 1983.