ON PROGRAMMING PARALLEL COMPUTERS

Leslie Lamport
Massachusetts Computer Associates, Inc.

INTRODUCTION

In this paper, I will make some general observations about how computers should be programmed, and how programs should be compiled. I will restrict my attention to programming computers to solve numerical analysis problems, although most of my remarks can be applied to other problem areas as well. My primary concern is for large parallel computers. However, I will show that parallelism is not a major issue, and I will not restrict the discussion to any single type of computer architecture.

The paper is a very general one, and I will make few specific proposals for language or compiler design. Instead, I will examine the fundamental nature of the programming/compiling process, how this process is done now, and how it should be done in the future. Few of my observations will be new or original. However, a careful analysis will lead to a somewhat surprising conclusion: a higher level programming language is needed both to reduce programming costs and also to obtain more efficient programs. The analysis will also provide some useful ideas for designing a higher level language and compiling efficient code from it.

ON PROGRAMMING

I will begin with a general discussion of what programming is all about. I will not say anything very profound, but I hope to clarify some of the problems, and provide a basis for the more specific material I will present later.

We are given a domain of problems to be solved. We may think of these problems as specified in some very abstract form -- perhaps as "pure thoughts". I will remain vague about the nature of a problem specification, since it is very hard to define it precisely. For example, the specification of a problem involving the simulation of a physical system includes some sort of statement about the likely range of certain physical parameters.

Given one of these problems, a necessary task is to generate a suitable machine code for it, where a machine code is a collection of operating instructions for some specific computer. For example, a machine language (binary) program for a PDP-10 and an input tape for some particular Turing machine are both machine codes.

This task is usually accomplished by writing a program, which is an intermediate stage between the problem and the machine code. The task of generating machine code for a problem is then broken into two steps: (1) programming -- the essentially human task of constructing a program which represents a solution to the problem, and (2) compiling -- the essentially computer-performed task of constructing a machine code which correctly implements the program. I will represent this process with the following picture:

$$\text{PROBLEM} \xrightarrow{\text{programming}} \text{PROGRAM} \xrightarrow{\text{compiling}} \text{MACHINE CODE}\ .$$

We can think of the problem, the program, and the machine code as variables assuming values in their appropriate domains, so programming and compiling are mappings between these domains.

The usefulness of this picture lies in the fact that it clearly shows the duality or complementarity of the programming and compiling tasks. Suppose we consider the problem and the machine code to be fixed and the program to be variable. Choosing a program which makes the programming task easier will tend to make the compiling task more difficult, and vice-versa -- assuming that all else remains the same. (Requiring the programmer to be blindfolded will make programming more difficult, but it will not make compiling any easier.) It is important to remember that this duality exists only when the program is varied, but the machine code -- or the class of acceptable machine codes -- is not.

In going from the problem to the machine code, the basic goal is to minimize the cost of solving the problem. This cost has three distinct components:

I Programming Cost -- the cost of generating the program. Its major element is the human effort required, including the debugging effort. One seldom solves a single problem on a single computer. Therefore, evaluating the effect of some policy on this cost requires considering the cost of modifying the program for a different problem (program maintenance) or for a different computer.

II Compiling Cost -- the cost of generating machine code from the program. Its major element is the computer resource used to perform all the compilations needed to write and debug the program. It also includes part of the cost of developing the

compiler. Since this latter cost is shared by all uses of the compiler, it is usually negligible.

III Execution Cost -- the cost of executing the machine code on a computer in order to solve the problem. It consists mainly of the cost of the computer resource used.

Observe that the first two costs are attached to the arrows in our picture. Hence, they are dual to each other. Decreasing the programming cost will tend to increase the compiling cost, and vice-versa -- assuming that the quality of the machine code and all other factors remain the same. The third cost is just attached to the machine code. It measures the quality of the end product of the programming and compiling processes.

Let us now examine the compiling task more closely. It can be separated conceptually into two parts: (i) translation -- the process of transforming a program into some correct machine code, and (ii) compiler optimization -- the process of choosing the best possible machine code. Although it is impossible to completely separate these two aspects of compiling, many compilers do have a separate optimization phase which is devoted solely to performing this choosing. Of course, other phases of these compilers also perform some compiler optimization, since they make choices among different possible machine codes.

There is a dual separation of the programming task into two parts: (i) the process of transforming the problem into some program, and (ii) programmer optimization -- the process of choosing the best possible program. It is impossible to actually separate them, but it is useful to remember that there are these two different aspects to the programming task.

Let us return to our picture of the programming/compiling process. Moving from left to right in the picture represents a loss of information. Choosing a particular program destroys some information about the problem. A machine code implementation of that program contains even less information about the problem than the program does. Information is the same as freedom of choice. There are many possible machine codes to solve a problem. When we write a program for the problem, we lose the possibility of obtaining some of those machine codes. For example, writing a PDP-10 assembly language program for a matrix calculation problem undoubtedly destroys the information about the program which is needed to generate an efficient machine code for the ILLIAC IV. We could certainly write a compiler to produce ILLIAC IV machine code from a PDP-10 assembly language program. However, it could not generate the type of efficient ILLIAC machine code that we could obtain by programming in an ILLIAC array language such as IVTRAN. By writing our program in PDP-10 assembly language, we have greatly restricted the compiler's freedom in choosing a machine code to solve the original problem.

It is very difficult to precisely define what "loss of information" means. In a certain sense, the PDP-10 assembly language program is equivalent to the original problem. It is theoretically possible to "decompile" it into an IVTRAN program which can then be compiled into very efficient ILLIAC IV code. However, we know that this is very difficult because something has been lost by transforming the problem into the PDP-10 program and must be put back by the decompiler. It is this vague something which I am calling "information".

This concept of information is closely related to the concept of entropy in physics. A liter of blue ink and a liter of yellow ink are equivalent to the two liters of green fluid obtained by mixing them, since the individual "ink molecules" are unchanged by the mixing process. However, ergodic theory tells us that it is more difficult to separate the two colors than it is to mix them. Without an ergodic theory for programming, we cannot precisely define the concept of information. However, it seems to be a valid concept, and we can place some trust in our intuitive understanding of it.

Consider a specific program for a certain problem. We say that it is a high level program if it contains much information about the problem, and that it is a low level program if it contains little information about the problem. In other words, we have lost less information about the problem in writing a high level program than in writing a low level one.

Equivalently, a high level program allows a great deal of freedom of choice in selecting a machine code to implement it, and a low level program allows little choice of machine code implementation. For example, an assembly language program allows the compiler little choice in the machine language it produces. It is only free to choose such things as the absolute machine addresses of certain program segments. Hence, the assembly language program is a low level one. Conversely, a Fortran program can be compiled into many different machine codes. For example, there may be many different instruction sequences for evaluating a single Fortran arithmetic expression. Hence, the Fortran program is a high level one.

Another way of viewing this is to think of a problem as specifying what is to be done, and a machine code as specifying how it is done. A program lies somewhere in between. The closer it comes to describing what is to be done rather than how to do it, the higher level a program it is.

A high level programming language is a language in which one usually writes high level programs, and a low level programming language is one in which low level programs are usually written. However, it is important to remember that even with a single programming language, one can write higher and lower level programs for the same problem.

Now let us consider the programming/compiling duality in terms of the level of a program. Suppose we want to obtain a certain quality of machine code for a given problem. The more freedom of choice the compiler has, the harder it is to make an optimal choice. Therefore, compiler optimization is harder for a high level language then for a low level one. Dually, the programmer has to make a more specific choice when he writes a low level program than when he writes a high level one. Therefore, programmer optimization is harder for a low level program than a high level one. Choosing the level of the program provides a means of making a trade-off between programming cost and compiling cost.

Our programming/compiling picture leads us to expect compiler optimization to be costlier for a high level program than a low level one. However, it gives us no reason to expect this to be true of

translation. Thus, although it is costlier to compile a single Fortran statement than a single assembly language statement, a Fortran program will have fewer statements than an assembly language one which solves the same problem. There is no reason to suppose that translating a Fortran program is costlier than translating an equivalent assembly language program.

There is, however, one reason why it may be costlier to translate a higher level program than a lower level one. It seems to cost more to develop a compiler for a higher level language, even one which does no optimizing. Hence, the program's share of the compiler development cost is greater for a program written in a high level language than for one written in a low level language. For a popular language like Fortran, this cost is negligible. However, it becomes significant if one is thinking of writing a compiler for an extremely high level language, such as English.

FORTRAN

Before I propose a new method of programming numerical analysis problems, let us consider the current method. Most numerical analysis programs are written in Fortran, so I will restrict my attention to Fortran programming. By "Fortran", I will mean ANSI standard Fortran [1], which represents the "intersection" of the various dialects which have been implemented.

Why is Fortran so popular? There are historical reasons for its popularity -- it was the first widely available high level language, so it has the weight of tradition behind it. However, there is a more valid reason why it is still so widely used today: Fortran programming provides a reasonably low cost means of solving numerical analysis problems. Let us examine why this is so in terms of the three components of this cost.

(1) Programming Cost: Fortran is a significantly higher level language than assembly language, and we know that it should be easier to write programs in a high level language. Because it is fairly simple and easy to learn, Fortran actually does make it easier to write programs. There is a small number of different statement types, and it uses ordinary algebraic notation and a simple subscript notation. Fortran also requires no detailed knowledge of how computers actually work -- it can be used by a programmer who has never heard of accumulators or index registers. This all helps to reduce programming costs by making Fortran programs easier to write and to maintain.

Another important factor in reducing programming costs is Fortran's machine-independence. A Fortran compiler has been written for almost every computer, so one can transfer a Fortran program from one machine to another with minimal effort. Although there are some incompatibilities among the dialects of Fortran implemented on different computers, these are relatively minor and affect only a small fraction of the statements in most programs.

(2) Compiling Cost: It is not too difficult to compile a Fortran program into machine code. The syntax of Fortran does not lead to efficient parsing, probably because parsing was not well understood when the language was designed. However, most of the executable Fortran statements can be translated quite easily into machine code. (The formatted I/O statements are notable exceptions.)

(3) Running Cost: Fortran was designed to enable the programmer to produce efficient machine code, thus keeping the running cost low. The concern for efficient machine code was apparent from Fortran's inception. For example, the numerical IF statement was designed to allow efficient use of the IBM 704's compare instruction.* Efficiency has taken precedence over elegance throughout Fortran's evolution. Thus, recursive function definitions are still not allowed.

This emphasis on efficient machine code has made it possible for a Fortran programmer to produce reasonably efficient machine code for solving numerical analysis problems on almost any computer. The ability to produce efficient machine code for a wide range of computers has contributed enormously to Fortran's success. One area in which Fortran compilers have not produced efficient machine code is input/output. This has led to non-standard I/O statement types in several dialects of Fortran.

Although Fortran produces efficient machine code for problems in numerical analysis, this is not true for other, non-numeric problem domains. Fortran's success is largely the result of designing it specifically for the domain of numerical analysis problems. However, the ability of Fortran to reduce programming costs has led to its use even for problem areas in which it does not produce very good machine code. For example, the ILLIAC IV Fortran compiler is written in PDP-10 Fortran.

Having listed the reasons for using Fortran, I will now explain why I feel that Fortran is no longer adequate. Its basic problem is that it is too low level a language. Fortran has come to be used as a sort of "universal machine language". (It has even been proposed that other high level languages should be compiled by first translating them into Fortran [4].) A machine language is one in which the compiler can simply translate a program in a straight forward manner without doing much optimization. The programmer who uses Fortran as a machine language is therefore trying to write a low level program. He wants to do all that he can to choose the final machine code, and to remove as much choice as possible from the compiler. Let us examine the effect of this practice on the cost of problem solving.

(1) Programming Cost: Writing low-level programs obviously increases the programming cost. The programmer must work harder to write the program in the first place. Because he is optimizing for efficient machine code rather than for readability, his optimization will almost always produce a program which is harder to understand, and thus harder to maintain. In practice, this manifests itself in

*Ironically, the problem of detecting "minus zero" defeated this attempt at efficiency.

"unstructured" programs, replete with undisciplined GOTOs and other obscuring features. This increase in programming cost is well recognized, and I need not dwell upon it. As we all know, the programming cost is becoming a larger and larger part of the total cost of solving a problem.

(2) Compiling Cost: By optimizing his own program, the programmer can decrease the cost of compiling it. Since the compiler has less to do, it should cost less to compile the program. Although this is true, the saving is not that important. Computing costs are decreasing, and we would much prefer to let the compiler do the optimizing instead of the programmer. I will have more to say about compiling cost later.

(3) Running Cost: The costs of programmer optimization have been incurred in order to decrease running costs. It has been felt that the extra programming effort was necessary in order to obtain more efficient machine code. There seems to be a general feeling that low level programs are necessary in order to obtain good machine code. However, our picture of the programming/compiling process shows that this feeling is based upon the assumption that the programmer is better at optimizing than the compiler is. If this assumption is correct, then the programmer wants to do the optimizing, and leave little choice to the compiler, thus writing a low level program. However, if the compiler is better at optimizing than the programmer, then the opposite is true. We then want the program to be a high level one, and leave most of the optimizing to the compiler.

In practice, neither the programmer nor the compiler is strictly better at optimizing. Each performs certain types of optimizations better than the other. Programmers are usually better at such tasks as choosing a numerical algorithm for solving the problem. Compilers are often better at such other tasks as register allocation. In order to produce the most efficient machine code, it is necessary to have the programmer perform only those optimizations which he does best, and leave the rest to the compiler.

The success of any new high level language for numerical analysis problems will depend largely upon its ability to obtain efficient machine code. Therefore, let us consider how good Fortran is in this respect. I will show that Fortran is too low level a language to obtain efficient machine code for the large parallel computers of the present such as the ILLIAC IV and the CDC Star, or for the complex computers of the future.

The idea of needing a higher level language to produce more efficient machine code is a new one for most people. In order to justify it, let me first consider why it has been possible to compile efficient code for so many different computers from a single Fortran program. The basic reason is that up to now, computers have all been pretty much alike. They have contained a single arithmetic unit connected to a random access memory of comparable speed. Viewed in this way, an IBM 370/165 is basically the same as an IBM 704. When writing a Fortran program, one is essentially programming a "Fortran computer". This Fortran computer reflects the essential properties of a standard sequential computer, as used for numerical computations. (The fact that this is still true today is perhaps partly due to the influence of Fortran on computer design.) An efficient program for the Fortran computer will describe an efficient program for a real machine.

However, input/output on the 370/165 is quite different from on the 704. Thus, ANSI standard Fortran has not been successful at producing efficient machine code for I/O. For a similar reason, the non-I/O aspect of Fortran is unsatisfactory for the new large parallel computers. The ILLIAC IV, consisting of sixty-four identical processors controlled by a single instruction stream, is quite different from an IBM 704. One therefore does not expect Fortran to be a good language for the ILLIAC IV.

It is usually thought that Fortran is unsuitable for the ILLIAC IV simply because it lacks any way of specifying parallel execution. However, this is not the reason. ANSI Fortran will soon be extended to include parallel operations [3], but the following problems will still prevent it from generating efficient code for parallel computers:

(1) Parallel computers differ in what they can execute in parallel. The presence or absence of a particular feature on a standard sequential computer will have only a small effect on the efficiency of the compiled code. However, for the ILLIAC, it could mean the difference between sequential and parallel execution of a program -- a difference of up to a factor of sixty-four in execution time. Machine independence is one of Fortran's main features. If the programmer can only express parallel computation which can be implemented on all parallel computers, then he will be unable to write an efficient parallel program for any of them.

(2) There is evidence to indicate that people are not very good at finding the possible parallel execution in their programs. In [7], I described a case in which compiling techniques developed for the ILLIAC can transform a standard sequential algorithm into better parallel programs than ones written explicitly in parallel for two different types of parallel computers. I once gave several members of the IVTRAN compiler group a seven line Fortran program to be rewritten in parallel for the ILLIAC. (This rather contrived program was a simplified version of the example used in [8].) None of them obtained as much parallel execution as would a compiler using the methods of [8].

These observations about parallel programming can be abstracted to obtain the following general reasons why higher level programs are needed to produce more efficient machine code.

(1) For programs to be run on a wider variety of computers, more freedom of choice must be left to the compiler.

(2) As computers become more complex, and parallelism introduces a large degree of complexity, the programmer becomes

less competent at optimization. Hence, more optimization must be left to the compiler.

A further example of the difficulty of writing efficient programs in Fortran is given by a surprising result reported by Owens [12]. He described how taking a Fortran program coded for the CDC 7600, translating it into a vector program for the CDC-STAR, and running the new program with a STAR emulator, resulted in a program that ran twice as fast on the 7600 as did the original program. The immediate implication of this is that it is better to program the 7600 as a vector computer than as a "standard Fortran computer".

I have worked on one important aspect of compiler optimization for parallel computers: developing techniques for the parallel execution of sequential programs. Many such techniques have been found, by myself and others [8-11, 13, 14]. Very often, the basic algorithm used in writing a Fortran program allows parallel execution. If the program were written in a simple, direct style then these optimization techniques would permit the compiler to obtain this parallel execution. However, after the programmer has finished optimizing his Fortran program, it is almost impossible for a compiler to find this parallelism.

What the programmer has done by his optimization is to write a lower level Fortran program, thus destroying information about the problem. Even if the information has not actually been "destroyed", it has been hidden so as to make it much more difficult to discover. I believe that most of the programmer optimization techniques we have learned for writing more efficient programs will ultimately turn out to be bad programming practice. As an example, consider the elementary technique of removing invariant code from a loop. Suppose we want to set all elements of a 50 element array A equal to the product of the scalars B and C . A novice might write the following:

```
      DO 1 I = 1, 50
1     A(I) = B * C  .
```

However, he would soon learn to write this "better" version:

```
      TEMP = B * C
      DO 1 I = 1, 50
1     A(I) = TEMP  .
```

But is it really better? The first version is easier to understand, since we need only look at statement 1 to see what values the elements of A receive. Hence, the first version leads to lower programming cost. Furthermore, it is a simple job for the compiler to transform the first version into the second. It then knows that TEMP is a temporary variable which need not be saved after execution of the DO loop. It might be difficult for the compiler to determine this from the second version -- especially if the programmer were clever enough to save memory by using TEMP elsewhere, perhaps even putting it in COMMON storage. I.e., programmer optimization can destroy the information needed to generate optimal machine code -- code in which B*C is saved in a live register and not stored in memory. Finally, in the ILLIAC IV, the DO Loop would be executed in parallel just once for all values of I . The first version is actually more efficient than the second for the ILLIAC.

So far, I have indicated why the programmer should write a higher level program. He could try to do this simply by writing a better structured, higher level Fortran program. However, I will now describe how Fortran makes it impossible to write a sufficiently high level program. First of all, Fortran requires the programmer to be too specific -- to make too many choices. Below are two examples of this.

(1) The ANSI Fortran standard requires that the integrity of parenthesized subexpressions be maintained when evaluating an arithmetic expression. The compiler may not use distributivity relations to obtain an algebraically equivalent expression which can be executed more efficiently. Algebraically equivalent expressions may differ because of roundoff error, but often this is not the case. The user has no way of specifying which parentheses are significant. Although it does not affect the ILLIAC, this restriction prevents the use of the methods of [11] for other types of parallel computers.

(2) Many numerical algorithms involve a convergent iterative procedure for computing successive approximations to the desired result. The iteration is terminated when some convergence criterion is met. As explained in [10], the most efficient parallel execution of such an algorithm often requires executing extra iterations after convergence occurs. However, this is impossible because the Fortran programmer must specify that the calculations stop after a particular iteration, and the compiled code must produce the exact results specified by the program.

Secondly, Fortran requires the destruction of certain information about the problem because it gives the programmer no way of including it in his program. The following are some examples of the information which a compiler would like to have to help it optimize, but cannot find out from a Fortran program.

(1) The context in which a subroutine or function is executed. From where is the subroutine called? For the ILLIAC, it is sometimes best to compile two different versions of a single routine -- one to be called in parallel, and another to be called sequentially which can perform its own parallel computations.

(2) Information about a called function or subroutine. What data can be modified by calling the routine? Is the routine called from anywhere else? If not, perhaps it should be compiled in line.

(3) Frequency of execution information. How often is a particular branch of the program actually executed? If a DO loop has variable limits, what are the approximate values of those limits? This information was provided by the original Fortran FREQUENCY statement, which has unfortunately disappeared from the language.

(4) There are also many other facts about the program which might help the compiler generate better code. E.g., is a certain input variable always positive? This might be necessary to allow the parallel execution

of a loop on the ILLIAC. The compiler would like to ask the programmer such questions, but Fortran gives him no way of putting the answers in his program.

The above criticism of Fortran is in no way intended to belittle its achievements. It has been a very useful tool, with a remarkable longevity. The ILLIAC IV compiler will produce surprisingly good parallel code from sequential Fortran programs, largely because of the basic conceptual soundness of many aspects of the language. However, a new way of programming is needed for the ILLIAC and for other new complex computers.

A NEW WAY TO PROGRAM

The discussion of Fortran indicates that both the programming cost and the execution cost can be reduced by writing higher level programs. I will now try to show how this can actually be done.

Programs should be written in a simple, straightforward fashion. This will make them easy for people to understand, thereby reducing the cost of writing and maintaining them, and easy for compilers to understand, thereby allowing better compiler optimization. This implies the use of hierarchical, structured programming.

Achieving this style of programming requires a programming language which encourages simple, straightforward programming. The need to inhibit constructions such as GOTOs which lead to unstructured programs is well understood. What is perhaps less well understood is the need to inhibit unnecessary elegance. It is elegant to allow dynamically defined data types. However, their use is not likely to lead to simple, easily understood programs. Elegance leads to short programs, but not necessarily to straightforward ones.

Writing simple programs does not mean using the simplest possible programming language. A Turing machine is simple, but its programs are not. A good programming language is sufficiently general and elegant to permit clear, simple programs, but not so elegant as to defeat attempts to compile efficient code. An example of what I consider to be a good programming language feature is the DO loop. It is very useful -- especially for the array computations common in numerical analysis, it is easy to understand, it encourages good structured programming, and it makes compiler optimization easier. Of course, I am assuming a properly defined DO loop, with no "extended range" and for which a "DO I = 1, N" results in no execution of the loop body if $N < 1$. One might desire some syntactic improvement to make the loop body easier to identify, but semantically it is marvelous.

The fact that a language is to be compiled for a parallel computer does not mean that it must be a "parallel language". People tend to think in terms of sequential processes. Some parallel operations are simple and natural, such as writing "A = 0" rather than a nest of DO loops in order to set all elements of the array A to zero. However, the fact that people think sequentially implies that programs should be essentially sequential.

Encouraging simple programs means discouraging the programmer's clever attempt at programmer optimization. This will not be easy, since programmer optimization has become a reflex action. Leaving invariant code inside a loop requires a conscious effort for many programmers. I do not know how the programmer's urge to blindly optimize can best be discouraged. One possible answer is provided by the growing interest in program correctness [2]. The desire to prove the correctness of his program should force the programmer to concentrate on simplicity rather than self-defeating efficiency.

It is easier to see how Fortran's specific impediments to writing higher level programs can be overcome. For example, one can devise a way of specifying convergent iterative algorithms which allows the execution of extra loop iterations. It is also easy to devise ways of including frequency of execution information in the program. I will not discuss any of these problems. However, I will briefly discuss the problem of I/O specification. Higher level programming means specifying <u>what</u> rather than <u>how</u>. Specification of I/O should be done by describing the relation between the input/output and the program variables. Thus, instead of writing

```
READ from file F into A
PROCESS A
READ from file F into B
PROCESS B
```

the programmer would specify in some way that A and B are the first two elements on file F , and then just write

```
PROCESS A
PROCESS B .
```

The compiler would then be free to insert the actual READ operations any place it chose, so long as it produced a correct implementation.

So far, I have been talking only about a new programming language. In addition to a new language, the programmer needs a whole new way of interacting with the compiler. With Fortran, the programmer simply compiles each subroutine individually, completely independent of any other subroutine. This is no longer satisfactory. If the compiler is to assume greater responsibility for optimizing then it requires information about the whole program. It cannot produce optimal machine code if it sees only one subprogram at a time, knowing nothing about the rest of the program.

I envision the compiler -- that object which actually produces machine code -- being imbedded in a larger <u>programming system</u>. The programmer uses the system in all phases of program design, from the initial high level design down to the final programming stage. At each stage, the programmer supplies the system with preliminary information about the program, and the system can respond with preliminary estimates of what the machine code will be like. This will allow the system to identify the most important parts of the program, so the programmer and the compiler can concentrate their efforts on them.

To obtain an efficient machine-independent program, the programmer may want to write two (or more) versions of some subroutines. Both versions would become part of the program. The compiler can choose the version which will produce the best machine code for each individual computer.

The ideas I have given for the programming system are quite vague, and not all original. Some of them can be found in [16]. Actually designing such a system is a formidable task. I have tried to

indicate why it is necessary. I will next explain why I think it is feasible.

COMPILING

The most difficult part of the programming system to implement is the compiler. Translation -- the process of simply producing any correct machine code for a program -- is not difficult. The difficult part of compiling is optimizing -- producing good machine code. I will therefore consider only the task of compiler optimization.

Compiler optimization methods can be divided into two general classes: local and global. Local optimization is performed on a single semantic unit of the program, such as one program statement, and it maintains the integrity of that unit. It includes such optimizations as choosing the fastest instruction sequence to evaluate a single arithmetic expression. Local optimization produces a separate machine code object for each program unit. Although very important and often quite difficult, local optimization presents no conceptual problems.

Global optimization involves the interaction among separate program units. It produces machine code in which there is no single machine code object which implements an individual program unit. Common subexpression elimination, in which a subexpression appearing in two distinct program statements is evaluated only once, is an example of a global optimization. The resulting machine code does not contain two separate objects which implement the two program statements.

The distinction between local and global optimization depends upon the level of semantic unit chosen. If the unit is the program statement, then removing invariant code from a DO loop is a global optimization. However, it is a local optimization if the DO loop is the semantic unit being considered.

The programmer's attempts at global optimization are primarily responsible for the high (programming) cost of programmer optimization. The basic principle of structured programming is that one should be able to think about individual program units independently of one another. Global optimization destroys this independence, so it destroys the hierarchic program structure necessary for reducing programming costs.

Most global compiler optimization techniques that have been developed are global with respect to the individual program statements, but local with respect to larger semantic units such as the subroutine. Effective compiler optimization must be global at the highest possible level. Thus, it should include inter-subroutine optimization. Such optimization techniques are discussed in [17]. As a simple example, let INVERT be a general subroutine for inverting an $N \times N$ matrix, with the matrix and the value of N as arguments. Suppose that a particular program always calls INVERT with N equal to 100. The compiler can then generate more efficient code for INVERT by substituting 100 for N , and eliminating N as an argument.

Programs seldom remain unchanged for very long. They are usually subject to modification, which means recompiling. In order to prevent very large compilation costs, this implies the use of incremental compiling: only the program unit which is changed should have to be recompiled. However, if there is no single machine code unit which implements it, how can we recompile just that one program unit?

With global optimization, we cannot guarantee that changing a single program unit will require recompiling just that unit. However, we can minimize the number of units which must be recompiled. To do this, some machine code object must be associated with each program unit. Global optimization prevents us from finding any such object which implements the original program unit. However, we can find a machine code object which implements that unit under certain assumptions about its environment. For example, common subexpression elimination removes the calculation of a subexpression from a statement and replaces it with a fetch of the value from some register. The compiler generates a section of machine code which correctly implements the statement under the assumption that the register contains the value of the subexpression.

For each program unit, the compiler can generate a machine code object and a list of assumptions under which that object correctly implements the unit. The unit must be recompiled if changing another program unit invalidated those assumptions. For example, consider the subroutine INVERT. The compiler generates a machine code subroutine for INVERT which is correct under the assumption that its argument is a 100×100 array. When recompiling any subroutine that calls INVERT, the compiler checks if the call still satisfies this assumption. If so, then INVERT need not be recompiled. If not, then a new machine code must be compiled for INVERT.

With a sophisticated global optimizer, incremental compilation requires saving a great deal of information about the environment of each program unit. Hence, it is feasible only for large program units such as subroutines. A less sophisticated optimizer can save less information, and can allow incremental compiling of smaller program units. Thus, a compiler which can recompile an individual Fortran statement is feasible only if it does no common subexpression elimination.

Two major objections have been raised against sophisticated optimizing compilers. The first is that they often make compilation too expensive. This would be a valid objection if the optimizer were only trying to minimize execution cost. However, a good optimizer will try to minimize the sum of the compiling cost, the execution cost, and the debugging cost. To do this, the optimizer must choose among various options for the different techniques which it can apply. These will include the option of no optimization (perhaps executing the program interpretively), options for applying techniques to produce faster running machine code, and options for including debugging aids in the machine code.

In order to minimize the total cost, the compiler must be able to estimate both the cost of performing a certain type of optimization and the benefits it will yield. Both types of estimates require that the compiler be able to estimate the quality of the machine code which it will generate with its various options. For example, it may know that with no optimization, a program block containing N arithmetic operations will yield machine code requiring an average of 2N microseconds to execute on a particular computer. Eliminating common subexpressions will reduce this

average to 1.5N microseconds, but will add N^2 milliseconds to the compiling time.

Estimating the benefit of applying an optimization technique requires that the compiler have good frequency of execution information. Not only must the compiler know how often a program block will be executed during a single execution of a subroutine, but it must also know how often the subroutine is executed during a single execution of the program. Moreover, it must know how many times the programmer expects to execute the program before the subroutine must be recompiled.

Frequency of execution information and the ability to estimate the quality of machine code will allow the compiler to choose the optimization option which is most likely to minimize total compiling and execution cost. It can first use the frequency information to quickly identify which parts of the program produce most of the execution time. It can then choose the best option for each of those parts. For example, suppose the compiler discovers that most of the execution time of a program occurs in one program block containing 500 arithmetic operations, which is executed 100,000 times each time the program is run. It then knows that it should perform common subexpression elimination on that block if the program is expected to be executed more than 10 times before the block must be recompiled.

Since it is very hard for the compiler to estimate debugging costs, the programmer should decide what debugging aids are to be included in the machine code. The compiler can provide him with the information needed to make this decision by telling him the compilation and execution cost of the various options.

The second objection which has been raised against optimizing compilers is that they are unreliable. There are two kinds of unreliability: producing incorrect machine code, and pessimizing instead of optimizing. Producing incorrect code has certainly been a real problem with optimizing compilers. However, it is just a manifestation of the general problem of writing correct programs. Progress in structured programming and program correctness should enable it to be solved. It should become less likely for compiler optimization to produce an incorrect machine code than for programmer optimization to produce an incorrect program.

The problem of designing an optimizing compiler which actually optimizes instead of pessimizing is harder to solve. High level programs leave most of the optimization to the compiler. However, the programmer will be better than the compiler at some kinds of optimization, although which kinds may depend upon the particular programmer. A programmer who wants to produce very good machine code with a high level language could find himself fighting the optimizer.

The solution to this problem is to let the programmer and the compiler cooperate, instead of having the compiler always act independently. This can be accomplished with a generalization of an idea described in [6]. We can let the programmer see the results of those optimizations which he might do better than the compiler. More specifically, there can be a compiler optimization phase in which the program is transformed into an equivalent lower level program in a machine-specific dialect of the original programming language. The programmer and the compiler can cooperate in this phase of optimization. The programmer can direct the compiler to perform certain transformations to selected portions of the program. He might even write some parts of the low level version entirely by himself. The compiler can provide him with information about the machine code which it will generate from the transformed version of the program.

Transforming the program into a low level one produced especially for a particular computer offers two related advantages: (1) the programmer can be confident that the compiler will produce good machine code from the transformed program, and (2) the compiler can make accurate estimates of the quality of machine code it will generate from the transformed program. Using a dialect of the original programming language makes it easier for the programmer to read the transformed program.

The programmer can choose how much help he wants to give the compiler. This allows him to make a trade-off between programmer optimization cost and program execution cost. Retaining the higher level, machine-independent version of the program helps lower the cost of maintaining the program and moving it to a different computer.

This idea has been used in the ILLIAC IV Fortran compiler. This compiler has a "Paralyzer" phase which transforms the Fortran program into a program written in IVTRAN -- a dialect of Fortran designed specifically for the ILLIAC. The same idea was also described in [15], except that the transformed version was a lower level program in the same programming language.

Optimizing by transforming into a lower level program has one additional advantage: the compiler can be written gradually. Initially, it can require a great deal of prompting from the programmer in order to produce an efficient low level program. The compiler can be continually improved to perform the transformation better, gradually relieving the programmer of the optimization task. Even the initial version would be very useful, especially since optimization is usually necessary for only a small part of the program [5].

CONCLUSION

I have tried to show that using a higher level programming language for numerical analysis problems can reduce programming costs while producing better machine code. This is a very encouraging idea, and it suggests that the cost of problem solving can be made very small by using a high enough level language. Unfortunately, there is one limiting factor: the cost of developing the compiler. This cost is prohibitive for an extremely high level language such as English. I have indicated that a good optimizing compiler can be written for a higher level language than Fortran. However, it would be a difficult and costly software project, and the programming profession has a poor record on such projects. Developing the type of high level programming system which I have described will require a level of programming quality that has rarely been achieved. I hope that progress in structured programming methods will make it possible.

REFERENCES

[1] ANSI Fortran Standard, American Standards

Association, American Standard Basic Fortran, X3.10, ANSI, New York, 1966.

[2] Elspas, B. et al. An Assessment of Techniques for Proving Program Correctness. Computing Surveys 4, 2 (June, 1972), pp. 97-147.

[3] Engel, F. Future FORTRAN Development. SIGPLAN Notices 8, 3 (March, 1973), pp. 4-5.

[4] Gear, C.W. What Do We Need in Programming Languages? Mathematical Software II, Informal Conference Proceedings, Purdue University (May, 1974), pp. 19-24.

[5] Knuth, D. An Empirical Study of Fortran Programs. Software: Practice and Experience, Vol. 1, Issue 2, (April-June, 1971), pp. 105-133.

[6] Knuth, D. Structured Programming with go to Statements. Stanford University, California, Department of Computer Science, Tech. Report STAN-CS-74-416, May, 1974.

[7] Lamport, L. Some Remarks on Parallel Programming. Massachusetts Computer Associates, Inc., Wakefield, Massachusetts, CA-7211-2011, November, 1972.

[8] Lamport, L. The Coordinate Method for the Parallel Execution of DO Loops. Proceedings of the 1973 Sagamore Conference on Parallel Processing (August, 1973), pp. 1-12.

[9] Lamport, L. The Parallel Execution of DO Loops. Comm. ACM 17, 2 (February, 1974), pp. 83-93.

[10] Lamport, L. The Hyperplane Method for an Array Computer. To appear in Proceedings of the 1974 Sagamore Conference on Parallel Processing.

[11] Muroaka, Y. Parallelism Exposure and Exploitation. Ph.D. Thesis, University of Illinois, Urbana, Illinois (1971).

[12] Owens, J.L. The Influence of Machine Organization on Algorithms. Proceedings of the Symposium on Complexity of Sequential and Parallel Algorithms held at Carnegie-Mellon University (May, 1973).

[13] Ramamoorthy, C.V. and Gonzalez, M.J. A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs. AFIPS Proceedings, Fall Joint Computer Conference, Vol. 35, (1969), pp. 1-15.

[14] Schneck, P.B. Automatic Recognition of Vector and Parallel Operations. Proceedings of the ACM 25th Anniversary Conference, (August, 1972), pp. 772-779.

[15] Schneck, P.B. and Angel, E. A Fortran to Fortran Optimizing Compiler. The Compiler Journal, Vol. 16, Number 4, (November, 1973), pp. 322-330.

[16] Schwartz, J.T. On Programming, An Interim Report on the SETL Project. Computer Science Department, Courant Institute of Mathematical Sciences, N.Y.U. (1972).

[17] Wegbreit, B. Procedure Closure in EL1. The Computer Journal, Vol. 17, Number 1 (February, 1974), pp. 38-43.