Examining the Usability of Parallel Language Constructs From the Programmer's Perspective
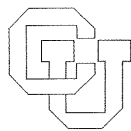
Robert P. Weaver and Clayton Lewis

CU-CS-492-90

University of Colorado at Boulder

DEPARTMENT OF COMPUTER SCIENCE

Examining the Usability of Parallel Language Constructs
from the Programmer's Perspective

Robert P. Weaver and Clayton Lewis

CU-CS-492-90   October 1990

Department of Computer Science
Campus Box 430
University of Colorado,
Boulder, Colorado, 80309

Examining the Usability of Parallel Language Constructs from the Programmer's Perspective[1]

Robert P. Weaver and Clayton Lewis

Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430

Abstract

We present a method for assessing the usability of language constructs which we have applied to the design of a language for parallel numerical computing. The language designer selects a suite of sample problems and analyzes the process users would have to go through to write programs for those problems. The decisions users must make are examined, and the facts and principles they must apply to make the right choices are noted. This inventory of knowledge and reasoning required to use a language can be used to compare alternative language constructs, identify weak points in a language design, and construct helpful user documentation. We believe this approach can be especially valuable in the design of languages for parallel computing, where the need to explore novel constructs coexists with the need to make languages readily usable by scientists and engineers who are not computer specialists.

1. Introduction.

Parallel computation is an extremely active area of research. In particular there have been a significant number of new languages proposed that are specifically designed to support parallel computation. Designers of these languages face the challenge of devising implementable constructs capable of expressing a wide range of computations. These constructs must also be understood and used by scientists and others who wish to exploit the power of parallel machines while investing as little effort as possible in mastering the tools.

Little attention has been paid to this latter aspect of the design problem, either in recent work on parallel languages or in earlier language design efforts. Work on software usability has concentrated on "end user" applications like word processing rather than on programming [2]. Textbook accounts touch lightly on the need to consider users' mental processes, which lie at the center of any consideration of usability, as an aspect of the language design problem [1, 3].

The challenge of parallel computing offers an opportunity to make progress in this area of design. Parallel languages must introduce new language constructs, and so offer wide scope for the development and application of new design methods. Equally, the intended role of parallel languages in supporting useful work by scientists, engineers, and others who are not computer specialists creates a need for design methods that can make these languages as easy to understand and use as possible. We describe here a method, the Programming Walkthrough Analysis, that may contribute to this design enterprise.

2. Background.

Evaluations of programming languages usually focus on the character of programs in the language. More rarely, and usually implicitly, operations on programs are considered. For example, can separately written programs be combined easily? From the user's perspective, these evaluations neglect key aspects of the use of a language which engage the user's cognitive faculties. Is it easy to write a program in the language, given a specification of a problem? Is it easy to understand a program once written?

The Programming Walkthrough Analysis is intended to allow language designers to come to grips with some of these issues. In particular, it focuses on the process of writing programs in a language rather than on the character of the resulting programs. For example, the analysis allows that programs in a language might be long and complex but yet simple to write or, more likely, that programs in a language might be short and elegant but yet very difficult to write.

This analysis can be helpful in design in a number of ways. First, analysis of a number of sample problems can provide an indication of where the trouble spots in a design lie. It is easy for a designer to misjudge the naturalness of a design because of the familiarity bred by the design process. The walkthrough analysis helps to see the design from the perspective of a user who is not familiar with it, without the overhead of getting real users to try the language. Second, the analysis can be used to compare design alternatives. Often there are alternative approaches to issues in language design, and differences in the decisions users must make can supplement or modify the intuition that often guides such choices. Third, the results of the analysis form an inventory of the knowledge users must have to use the language effectively. This inventory can be used to make sure that documentation includes what is needed.

In the remainder of this Section we first describe the Programming Walkthrough Analysis, then we describe the parallel language we will use in our example. In Section 3, we present a simple example to illustrate the analysis and discuss the results of using a more complex series of examples. In Section 4, we discuss the benefits and limitations of the analysis. Finally, in Section 5, we suggest areas for future research.

2.1   The Programming Walkthrough Analysis.

To use the Programming Walkthrough Analysis to evaluate a language the designer chooses a suite of one or more specific problems whose solutions will be evaluated. He or she then writes programs for these problems in the language, keeping track of the decisions that must be made to produce the program. All decisions made in the process of developing the solution are noted, including decisions about choice of data structure and algorithm that may be made before any code for the solution is written. The designer also notes what basis is available for making each decision, that is, what facts and principles should be used to make the right choice.

It is this basis that is the key to the Programming Walkthrough Analysis. A little thought about what a user must do to write a program reveals immediately that users of a language must know much more than the definitions of the various constructs of the language. They have to know, for example, how to combine low-level structures into complexes that correspond to meaningful parts of problems [4]. They may have to know in what order decisions about data structures and algorithms should be made. While some of this knowledge may be part of general programming knowledge that every user of the language can be expected to know, some is specific to a particular language, and must in some way be made available to its users. We call this latter knowledge "doctrine".

The sequence of user decisions, together with the indication of the doctrine behind each, becomes the focus for the evaluation. A language will look good for a given problem if few decisions are required, or if only a small

amount of doctrine is required for each decision. A language will look bad if many decisions are required, or if the doctrine is lengthy and/or complex. These decisions will require extensive problem-solving for users of the language, so that writing the program can be expected to be difficult and error-prone.

The usability of the language is influenced not only by the nature of the doctrine required by individual problems but also by the doctrine needed to solve a class of problems for which the language might be used. If the same compact and simple doctrine suffices to write a wide range of programs this is good; if different problems require very different doctrine this is bad. In this bad case users will have to know a lot of doctrine to get their work done. Additionally, some of this doctrine will probably be conditional in character, contributing to the difficulty for users because the doctrine contains ideas that are sometimes, but not generally, applicable.

Thus developing a body of doctrine adequate for general use of a language is an important part of assessing the language's usability. We do this by an iterative process, starting with draft doctrine which we hope will cover most of the ground. We then walk through a suite of sample problems, noting all decisions for which the draft doctrine is inadequate or incorrect. We then refine the doctrine and repeat the process, until the doctrine is adequate, or until it becomes clear that the language design permits no sufficiently simple body of doctrine to cover these problems. In the latter case we can accept the weakness of the design or we can try to change the design so as to avoid the issues that the problems raised.

We will illustrate the programming walkthrough analysis by presenting an analysis of a particular pair of alternative language constructs. We did eight walkthroughs for the entire analysis. We will present in some detail the simplest walkthrough and then our results for the complete set of walkthroughs pertaining to these alternatives. First, we present a brief summary of the parallel language that we draw our examples from.

2.2   DINO

DINO is a language for programming distributed memory parallel computers which is designed primarily for doing regular numerical problems in a data parallel fashion. DINO is based on the philosophy that the programmer must say something about the way in which the problem is parallelized. To this end, it attempts to provide the programmer with high-level constructs for distributing data to processors and specifying inter-processor communication.

Essentially, DINO allows the programmer to declare a structured virtual parallel machine on which the computations will be performed, to declare the way that each piece of global data is to be distributed among these processors, and to specify communication patterns among the processors. The programmer then writes code for the computation that occurs on a typical processor, using a simple syntax to specify when communications should take place. The compiler handles all the other details of process initialization and termination, distribution of global data, and communications.

There are four major concepts in DINO: environments, distributed data, composite procedures, and communications. Environments are virtual processors. DINO allows the programmer to declare a structure of environments, that is, a collection of virtual processors usually organized as an array which is suited to the specific problem. Then the programmer specifies how data structures are distributed among the environments. This specification allows for replication as well as partitioning and, additionally, tells DINO what the communication patterns among the environments will be. The programmer next writes the code for a typical environment as a composite procedure. When that procedure is called, each environment will execute the same code using its portion of the distributed data, resulting in

a single program multiple data form of parallelism. Finally, communications occur in two ways. The first is parameter distribution and return when a composite procedure is invoked. This is handled automatically by DINO. The second is when the programmer specifies communication between environments in a structure. The programmer need only specify the name of the data to be sent or received and where (lexically) the communication occurs and DINO handles the rest.

In addition to this general description, it will be helpful for the reader to be aware of the following points of DINO syntax: (1) It is a superset of C. (2) Sends and receives are specified by "decorating" a variable name with a "#" either in a write context (send) or a read context (receive). (3) DINO has a syntax for referencing sub-arrays. For more information on DINO, see [5].

3. An Illustration of the Programming Walkthrough Analysis using DINO.

We present an illustration of the use of the Programming Walkthrough Analysis in the context of DINO. For this illustration, we will explore the differences between a pair of alternative language constructs. We first present the doctrine we developed for DINO before doing these walkthroughs. Second, we look in some detail at a walkthrough generated by applying one of our alternative language constructs to one simple example from the set of examples we selected for this illustration. Third, we present the results of doing walkthroughs on the entire set of examples. Finally, we discuss the results of this process.

3.1 The Doctrine for DINO.

Here is our initial draft of the DINO doctrine, somewhat abbreviated:

I. You must know the doctrine for the language C.

II. Before proceeding to create a DINO program for the particular problem, make sure you know:

    A. The basic algorithm for the solution;

    B. A basic idea about how the solution will be parallelized; and

    C. A good idea of the data structures necessary for the solution.

III. There are four steps necessary to create the DINO program:

    A. Chose a structure of environments --- chose a structured virtual machine on which to do the problem.

        1. Chose constants that will be used to identify the name of a specific environment.

    B. Determine how the principal data structures should be distributed among the environments in the structure you selected:

        1. For each data structure, determine the basic partitioning; or

        2. Decide if it should be replicated. Then,

        3. If you chose partitioning, decide where any copies (used for communication) should go.

    C. Write a composite procedure that specifies the computation on each environment:

        1. Write the code just to do the computation.

    2.   Insert any necessary receives

        a.   A rule of thumb is to put them as late as possible.

        b.   Another rule is to put as much data as possible into a single receive.

    3.   Insert any corresponding sends:

        a.   A rule of thumb is to put them as early as possible.

        b.   Another rule is to put as much data as possible into a single send.

    D.   Write a host procedure to specify the computation on the host.

3.2 Exploration of Alternative Constructs.

In DINO, communication is currently designated by appending a "#" to a reference to a distributed variable. If the reference occurs in a write context, then the communication will be a send; if the reference occurs in a read context, the communication will be a receive. One consequence of this particular syntax is that there are always two different things happening in any communication. In the send case, a new value is assigned to a distributed variable and that value is sent to other processors. In the receive case, a new value is received for a distributed variable, and that value is produced for consumption in an expression.[2]

Our issue is to explore an alternative syntax that would separate out these two functions. So we propose an alternative syntax to DINO where a send is generated by the statement "Send(X);" where "X" is some distributed variable. This statement would not assign a value to the variable. Similarly, "Recv(X)" would generate a receive but not produce a value. Write and read would be designated with normal C syntax. We call the current syntax of DINO the "#" alternative and the proposed change the "Send/Recv" alternative.

We selected four examples, all designed to exhibit strong or weak points of one alternative or the other. We then constructed one DINO programs for each example using each of the alternative approaches. Here is the walkthrough generated by using the "#" alternative with our simplest example.

3.3 Example Walkthrough.

Our example problem is a trivial one. We wish to pass an integer from one processor to another. The integer should start at "1" and be incremented each time it is received. Each processor should retain its integer and when the entire computation is finished, all these integers should be returned to the host as a vector. We use the "#" alternative in this example.

We set out each conceptual step that we take, annotating it with the references to the piece(s) of doctrine that were necessary to take that step:

- The algorithm for the solution and the way that it is parallelized are obvious from the problem statement. (II.A and II.B)

- Since the problem requires a result vector, we will use a vector as our data structure to store the integer retained by each processor. We call that vector A. (II.C)[3]

- Since our principal data structure is a vector and we will want to put one element of that vector on

---

2. Of course, in C, you can throw away this value.
3. We omit the portion of the walkthrough in which the user determines the appropriate DINO declarations to implement each of these steps for brevity's sake.

each environment, we chose a vector of environments (recall that in DINO, an environment is a virtual processor). (III.A)

- We partition our vector *A* one element to an environment. (III.B.1)

- Each environment will have a copy of the value of *A* from the environment to the left of it. We will use those copies to communicate the value of the data from left to right. (III.B.3)

- Initially we decide our code is "`A[id] = A[id - 1] + 1;`". Note that "id" is a declared DINO constant that evaluates to the index of the current environment. (III.C.1 and I)

- But then we realize that this doesn't work on the first environment. So our code becomes:

```
if (id == 0)
    A[id] = 1;
else
    A[id] = A[id - 1] + 1;
```

- Doctrine point III.C.2 tells us we need to receive the value of `A[id - 1]` but does not tell us just how to do this. In the "#" alternative we need to decorate the read of `A[id - 1]` with "#", resulting in the code "`A[id] = A[id - 1]# + 1;`", and we add a point of doctrine to cover this situation (see point III.C.2.c below). (III.C.2)

- Now point III.C.3 tells us we need to send `A[id]`; again, we need to add a point to doctrine to say just how to do this (see point III.C.3.c below). The resulting code has "`A[id]# = 1;`" and "`A[id]# = A[id - 1]# + 1;`". (III.C.3)

- However, we realize that this will generate an unnecessary send in the last environment. So we modify our code to look like:

```
if (id == 0)
    A[id]# = 1;
else
    if (id != N - 1)
        A[id]# = A[id - 1]# + 1;
    else
        A[id] = A[id - 1]# + 1;
```

- The host procedure (ignoring the output) is one line, "`test(a[])#;`". (III.D)

Note that in this process, we have discovered two additional points of doctrine that must be added -- how to receive a value and how to send a value. Because these points are specific to one of the alternatives we are testing, they will be added to the version of the doctrine that is associated with that alternative.

3.4   Results of other Examples.

In a manner similar to that set out above, we do walkthroughs for each example we wish to use, one for each alternative. In several additional places, we find that additions to the doctrine are necessary. The following are additional points of doctrine that are specific to the "#" alternative when we have completed walkthroughs for all the examples:

III.C.2.c.   To do a receive, locate a read of the variable you are interested in at the right place (lexically) and decorate it with a "#".

III.C.2.d. If there is no read at the right place, you can create one as in "A#;". This causes A to be read and its value to be discarded. A side effect is that a new value for A is received first.

III.C.3.c. To do a send, locate a write to the variable you are interested in at the right place (lexically) and decorate it with a "#".

III.C.3.d. If there is no write at the right place, you can create one as in "A# = A;" Only the side effect of sending the value of *A* will actually occur.

III.C.3.e. Remember in a simple C statement, reads occur before writes. Therefore if you decorate both a read and a write in one statement, the receive associated with the read will occur before the send associated with the write.

The following are additional points of doctrine that are associated with the Send/Recv alternative:

III.C.2.c. To do a receive, at the right place (lexically) insert the statement "Recv(X);" where *X* is the variable whose value is to be received.

III.C.3.c. To do a send, at the correct place (lexically) insert the statement "Send(X);" where *X* is the variable whose value is to be sent.

Finally, in the process of doing these walkthroughs, we found an additional point of doctrine that applied to both alternatives. It was:

III.C.1.a. Write code for the middle first. Middle is used in two senses: middle spacially meaning not on the edge of a data structure or not on the edge of a block of a data structure that is resident on one environment; middle temporally meaning not the first or last iterations of loops. Then look at all of the edge cases one at a time to see if the code must be modified.

3.5   Results and Implications for DINO

We can draw several useful results from this walkthrough analysis: (1) We have gained additional information about our proposed alternative constructs. (2) We have found some additional general information about DINO that is unrelated to the alternatives we set out to explore. (3) We have obtained information that would be extremely useful in constructing user documentation. We look at each of these in turn.

As we saw, the doctrine for the "#" alternative is more complex than that needed for the Send/Recv alternative. This suggests that this alternative would be harder to learn and use. We could respond to this finding by adopting the Send/Recv approach, or by seeking ways to make the "#" alternative work better. Much of the complexity of the "#" approach comes from the need to force sends and receives at points where there are no corresponding writes or reads, as covered in doctrine points III.C.2.d and III.C.3.d. We cannot present the analysis here, but we can identify what type of problem leads to these situations, and we can propose other language features that deal with them without the need for these points of doctrine. Thus the walkthrough analysis can help in developing design alternatives as well as choosing among them.

This example makes another point about the use of the walkthrough results. While the "#" needs more doctrine, it also sometimes needs less code. A final choice of design would have to weigh this fact as well as the walk-

through result. Implementation issues would have to be considered as well. So the point is not that the walkthrough analysis delivers the ultimate verdict on design alternatives. Rather, the value of the method is that it permits a more fully informed decision to be made.

The example also shows that the walkthrough turned up information for which we were not looking. The doctrine about treating the "middle" of a problem first does not bear on the design alternatives, but does point up an area of the overall design that might repay attention. Perhaps more support can be provided in the language for the "edges" of problems, for example by detecting and suppressing unneeded final sends. Because of the ability of the walkthrough to turn up issues like this we have found it useful as a way to explore a design, even when no specific alternatives are to be considered.

Finally, the example shows how an inventory can be made of what users really need to know to use DINO. In preparing documentation for the language we are including not only the usual definitions of language constructs but also the doctrine, in explicit form. We also are including walkthroughs of two sample problems, to help users understand how the doctrine can be used to solve problems. Historically, this kind of direct guidance to users has been the province of programming courses and textbooks. We want DINO users to find out as soon as they get the software what they need to know to do useful work.

4. Discussion of The Programming Walkthrough Analysis.

We believe the value of the Programming Walkthrough Analysis is that it provides a systematic method for identifying usability problems in a language design. As such it provides a way for designers to include usability in the design debate along with expressiveness and implementability. It can be used to compare design alternatives, as in the example; to help formulate new alternatives in response to problems; to look for problem areas in a design even when no alternatives are under consideration; and to guide the development of documentation.

We find that these benefits more than repay the modest investment required, but there are limitations that must be kept in mind. First, the results will only be as good as the problems that are analyzed. In comparing alternatives problems must be chosen that place the alternatives in a realistic context, as they might appear when actual users confront the language. Using only very simple problems, such as the one that generated our example walkthrough above, runs the risk of missing key issues that would arise in real use.[4]

A second limitation is that intuition and designer's judgement play a big part in the use of the method. One could say that the method replaces a designer's vague and unfocussed intuitions about usability with more concrete and focussed ones. The designer does not have to decide whether alternative A is more natural than alternative B, but does have to decide whether the doctrine for A is more complex than that for B. An important gain is that considering how alternatives A and B work out in the process of writing particular programs is much easier than deciding their merits in the abstract. It also appears useful to recognize, as the method requires, that a language design must be evaluated in such a way that not just the design itself, but also the knowledge needed to apply it, must be weighed.

5. Future Extensions.

Writing isn't the only thing people must do with programs. We think the walkthrough approach can be extended to examine other tasks, including first of all reading programs. The same principles can be applied: examine

---

4. Other examples we used in the analysis were more complex.

the process of reading and inventory to knowledge needed to carry out the task successfully. Similarly, debugging is also a critical task, and one that is especially difficult for parallel languages. In a like manner, the walkthrough approach can be extended to examine the usability of language constructs from the perspective of detecting and correcting programming errors.

References

[1]   B. MacLennan. *Principles of Programming Languages*. Holt, Rinehart, and Winston. New York. (1987)

[2]   A. Newell and S. Card. "The Prospect for Psychological Science in Human-Computer Interaction". *Human-Computer Interaction*. Vol. 1. Page 209. (1985)

[3]   R. Sebesta. *Concepts of Programming Languages*. Benjamin/Cummings Publishing Company, Inc. Redwood City, CA. (1989)

[4]   E. Soloway and K. Ehrlich. "Empirical Studies of Programming Knowledge". *IEEE Transactions on Software Engineering*. Vol. 10. Page 595. (1984)

[5]   M. Rosing, R. Schnabel, and R. Weaver. "The DINO Parallel Programming Language". Technical Report CU-CS-457-90. Department of Computer Science, University of Colorado. (April 1990).