The ParaScope Editor: User Interface Goals

Kathi Fletcher
Ken Kennedy
Kathryn McKinley
Scott Warren

CRPC-TR-90052-S
1990

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

The ParaScope Editor: User Interface Goals

Kathi Fletcher
Ken Kennedy
Kathryn McKinley
Scott Warren

Abstract

This paper describes a new, more unified user interface design for the ParaScope Editor, an exploratory parallel programming tool being developed in the ParaScope programming environment. It discusses in detail the work model it is intended to support, and how program information is to be presented to users.

1 Introduction

The ParaScope Editor is designed to provide users with the means to efficiently convert serial Fortran programs to parallel programs for shared-memory multiprocessors. Automatic translators are not sufficient because they are often overly conservative in their estimation of what parts of the code can safely be made parallel. The user's knowledge about the program must be tapped in order to find more parallelism. The ParaScope Editor provides the user with three views of the program.

1. The *source code* display shows the actual program text. In addition, it indicates the loop or program region which is selected for consideration and highlights the source and sink of a selected dependence.

2. The *dependence* display shows a list of all dependences within the selected regions. In particular, if a loop is selected, it shows all the dependences carried by that loop.

3. The *variable* display shows information associated with each variable referenced in the selected region. The information available includes the classification of the variable as *shared* or *private* as well as the reason for its classification.

The dependence and variable information is provided to the programmers so that they may discover overly conservative information which inhibits parallelism. Programmers can edit the program information and select from a variety of source transformations which may lead to additional parallelism. The number of dependences associated with a loop is often quite large and can be overwhelming, so ways of organizing and selectively viewing the dependences are provided.

In the new design, the same program views are maintained, but additional functionality and consistency are provided. For example, in the current version, dependences can be deleted using a query dialog. Deletion of a dependence removes it from the dependence graph and forever from the user's view. Instead of removing dependences, the new design calls for a sophisticated marking mechanism which allows dependences to be hidden from user and/or system consideration. Dependences are however easily returned for reconsideration by either. Also, dependence classification and variable classification currently use two distinct mechanisms.

The complete functionality and user interface of the existing version of the ParaScope Editor (PED) are described elsewhere [BKK+89, KMT91]. This document describes a new, unified user interface design which will facilitate the parallel programming process in the ParaScope Editor. In particular, this document proposes a unified work model for the classification of dependences and variables. Many other issues for the user interface which arise in presenting program information and modifying programs are also addressed. The new model provides a more convenient and consistent way for users to understand and process the information presented to them.

This design resulted from a series of meetings and experiences with PED by a small working group. This group consisted of the authors of this document, who are tool developers and a user interface expert, and a few applications researchers who used PED to assist them in parallelizing programs. The resulting document will be the working design for the next generation of the PED user interface.

2 The Work Model - Dependence and Variable Classification

When writing and modifying code, programmers typically execute a series of experiments and ideas before obtaining the desired resultant code. The same process occurs in ParaScope when users are editing source code, experimenting with transformations, and editing program dependence information. Source to source transformations enable the user's code to remain recognizable throughout the editing process. The work model for editing program information must take this incremental editing process in to account. Most importantly, it should be easy for the user to keep track of which information has been changed and the thought process leading to the change. Keeping a history allows users to reconsider previous choices and to experiment by creating alternate program versions.

The new work model achieves these goals by making all dependences available for viewing, including those that the user has rejected. Each dependence has a classification mark associated that may be edited by the user. This mark indicates whether the user wants the dependence to be honored or ignored. A comment field is also provided with each dependence and can be used to record the "reason" the user changed the classification of the dependence.

Variable classification works similarly. The user can edit the classification information associated with a variable and include a comment. Throughout the document, explanations will be given in terms of classification of dependences, but they apply to variable classification as well.

2.1 Marking - Accepted, Rejected, Pending

Each dependence or variable will be returned from dependence analysis with a system generated mark of accepted or pending. Those dependences (or shared variables) which can be proven to exist by the dependence analyzer will be marked accepted and all others will be marked pending.

The user can change the mark of any dependence to accepted, rejected, or pending. Rejected dependences do not disappear, they are simply marked rejected. The user can later look at rejected dependences and change them back to pending or accepted. In addition, the user can include a comment which will be displayed with the dependence. This comment field is intended to indicate the reason that the dependence was accepted, rejected, or pending. However, the user can put anything in this field.

The user will have full control over the mark assigned to a dependence. Users can change a mark as many times as they like. Because the user may want to mark groups of dependences, it is helpful to have the ability to warn the user of possible anomalies that may result. (See 2.2.4 System Warnings)

The analysis of the safety and profitability of transformations in PED will take into account all dependences marked pending or accepted, and ignore rejected ones.

2.2 A Typical Session

During a typical editing session, the user will systematically go through the list of pending classifications, marking them as accepted or rejected. The dependence view filter will be set to exclude rejected dependences so as a dependence or variable is marked rejected, it will "disappear" from view, leaving only the remaining pending and accepted dependences. This

mode of filtering and marking will be the default, but users will be able to view and mark dependences as they see fit by setting a view filter and mark filter.

2.2.1 Marks Automatically Generated Due to Reclassification of Variables

Currently in PED, when the user moves a shared variable into the private list, all dependences which were incident on that variable are removed. Removing the dependences, however, can lead to transformations seeming legal which should be inhibited.

However, with the new proposal dependences do not disappear, but are simply marked pending, accepted or rejected. In the case of a variable being made private, the incident dependences should be marked in such a way that they do not prevent parallelization but are still used in calculating the legality of code transformations.

Changing a variable from shared to private and back to shared should leave the system in exactly the state it was in before these operations were performed. In other words, the reclassification of variables should be invertible. Additionally, the state of the dependences and the variable classifications should be consistent. If a variable is in the private list, it makes no sense for some dependences which were initially incident on the shared variable to remain as accepted dependences. If the program were reanalyzed with the variable specified as private those dependences would not be regenerated.

To satisfy these two conflicting constraints, PED will "remember" the user's previous classification of dependences when they are rejected because a variable was moved from shared to private. The system will work as follows:

- When a user moves a variable from shared to private, all dependences incident upon that variable will be marked rejected regardless of the user's previous classification. The user's old classifications will be stored.

- If the user moves a variable which was originally shared from private back into shared, the user's previous classifications on the associated dependences will be restored.

- If the user attempts to "accept" one of the dependences which were rejected because the variable was moved from shared to private the user will be informed that remarking this dependence will cause the variable to be made shared again. Then the user can cancel the edit or continue. If the user chooses to proceed, the variable will be made shared and the dependence classifications and marks associated with the variable will be restored to their previous values. The check for this condition will be made only when the user moves dependences from rejected to accepted or pending, which presumably will not be a common occurrence.

2.2.2 Dependence Count Associated with each Variable

Associated with each variable will be a field containing the number of dependences incident on that variable. Each time the user reclassifies a dependence the counter of the associated variable will be decremented or incremented accordingly.

A variable with a dependence count of zero can be made private. As a short cut to the user, a button at the top of the variable display can be used to automatically reclassify as private all those variables which have zero associated dependences, and which meet all the criterion used to determine that a variable can be made private.

2.2.3 Marks in the Context of Incremental Analysis

Dependence analysis is not currently capable of taking into account any semantic information which may be inferred from the user's marking to dependences. It simply generates the dependence graph. In the future, when PED is able to accept semantic assertions from the user about the program rather than, or in addition to, commands to delete individual dependences, it may be worthwhile to use this information during dependence analysis.

In the process of creating and editing a program, parts of the program will need to be reanalyzed to keep dependence information up to date. The user will not want to have to reconsider unchanged dependences that have been previously considered. Upon reanalysis, in either a batch or incremental system, the existing dependence graph will be retained and then compared with any new edges returned from analysis. New dependences whose dependence type, statement numbers, reference offsets, and reference names match a previously marked dependence will be classified according to the user's original specification, but will have a mark set in the "reconsider" field. Generated dependences that PED cannot match with an old dependence will be marked pending or accepted according to the result of analysis.

Reconsider marks can be manipulated in the same manner as status marks. In addition, there is an easy, one step method to clear all the reconsider marks. If the user changes the status mark or the comment field of a dependence, its reconsider mark will automatically be cleared.

After dependence analysis, it is also necessary to reclassify variables as shared or private. First as many dependences as possible will be matched and marked according to the user's previous categorization. If all of the previous dependences incident on the variable are consistent with the new dependences, PED will preserve user's previous variable classification. Otherwise the user's previous classification is ignored.

2.2.4 System Warnings

Because the user is being given a great deal of freedom to mark dependences and can use query shortcuts to mark many dependences at once, the user may mark dependences unintentionally.

Warnings are appropriate when the user's action is irrevocable. Other less catastrophic but still potentially hazardous actions will be indicated by giving the user a brief optional message followed by marking the reconsider field of the selected dependences. The warning will be included in the reconsider field.

The following are instances where a system warning or automatic tracking mechanism is useful.

- The user can mark a dependence as rejected that the system proven to exist. Some method of tagging this occurrence would be helpful to prevent accidental remarking and to flag potential errors for debugging.

- If the user is about to accept a dependence which was rejected because a variable was made private, the user will be warned that accepting this dependence will result in making the variable shared and restoring any other rejected dependences incident on the variable to pending or accepted.

2.3 Undo Feature

The user will be able to undo at least the last marking step. By maintaining a stack of the dependences and their original classification that were affected by marking at each step, multiple undos can be supported.

2.4 History Versus Time Independence

Each dependence will have only one classification mark and one comment with the exception of dependences automatically marked because a variable is moved from shared to private. The last mark and comment that the user applies to a dependence will be the only mark and comment available. No editable history will be kept of the marking steps that the user performs because the steps may be order dependent. Changing any portion of the history could lead to unexpected results.

To illustrate that the order in which marking steps are applied is important, consider the following example. Given the initial condition of pending dependences on the variable "a", the order in which the following two rules are applied is crucial.

1. Mark all pending dependences on variable "a" as rejected.
2. Mark all rejected dependences on variable "a" as accepted.

3 Displays

3.1 Dependences and Variables

It is necessary to separate filtering the view of the dependences and selecting a set of dependences to be marked. The user will usually want to look at a different set of dependences from the ones to be marked. For instance, the user might want to look at a superset of the dependences to be marked or a related set in another part of the program.

3.1.1 Filtering - view control

In order to be consistent with the ParaScope environment, the same view filtering mechanism that is present in the structured text editor, NED, will be used to filter the dependence and variable displays. In NED four filters can be used, bold, show, dim, and hide. The current view filtering works like an if-then-else. The first rule which applies to a statement is performed and no others are tried. In addition to the current NED functions, the user will be able to sort the dependences along various criteria. The default view in the dependence display is to present the pending dependences on all of the variables in the selected loop.

3.2 Filtering and Marking

The same filtering mechanism as above may be used for marking groups of dependences. A query dialog will provide access to the mark filtering facilities. The user may view the dependences about to be marked by selecting a view button within the marking dialog. These dependences will then be displayed in a separate window.

3.2.1 Marking - Pending, Accepted, Rejected

ParaScope users will be able to mark dependences in at least three ways. They can point to a single dependence in the dependence view and mark it, or they can choose groups of dependences to be marked using click and drag. Editing one field in a group of selected dependences will cause all of them to change correspondingly in that field. Variable classification editing will behave analogously. A target box next to each dependence or variable classification will darken to indicate that the entity is selected for marking or editing. This target box will distinguish between selection of a dependence or variable for display in the text pane, and selection for editing program information.

A query dialog will be provided in order to give the user the power to mark sets of logically related dependences or variables. These methods will also be available to clear the reconsider mark.

- **Comment**

  Each dependence will have a "comment" field. This is a purely textual item with no associated semantics interpreted by the system. The field is provided as a convenience for the user. The field is envisioned to store the "reason" the user chose the classification. It could also be used to explicitly record the filter criteria used to mark the dependence. The filter criteria will not be maintained by the system.

3.2.2 Features of the Dependence Display

At the top of the dependence view display, the current dependence view filter criteria and a dependence count will be indicated. The following information will be available for each dependence : *line number, dependence type, src and sink name, level, common block, dependence vectors, regular section information, status mark, comment, time stamp, user,* and *reconsider field.*

- **Dependence Pane Information Line**
  - **Current View Filter Criteria**
    An indication of the current view filter criteria will appear at the top of the dependence display. The criteria display will take advantage of abbreviations in order to encapsulate all the information but will expand upon request.
  - **Total Number/Total Pending**
    The total number of dependences and the total number that remain to be classified will be available for the entire program or for the current view.
- **Dependence Information**
  - **Line Number**
    The line number of the source.
  - **Type**
    True, anti, output, call, or control.
  - **Src/Sink**
    The source and sink variables as well as their subscript expressions.

    - **Bindings across routines**
      If the dependence arises from the use of a global variable within a subroutine call, the subroutine call is displayed.

- **Level, and Common Block**

  The loop level on which the dependence occurs, and the common block name that the variable belongs to are displayed as is currently done.

- **Dependence Vectors**

  The most precise information available about a dependence's direction and distance are combined and presented here.

- **Regular Section Descriptors**

  These are displayed for dependences on arrays whose source or sink is a call site. This information will be available locally upon request.

- **Status**

  The dependence's status is displayed here (pending, accepted, or rejected).

- **Comment - system/user**

  It is possible to have both a system and user comment but since the comment has only the semantic information that the user attaches to it, just one field will be used. If a variable is moved from shared to private, the user's previous mark and comment are stored, and a system mark and comment are generated to replace them.

- **Time Stamp**

  The time stamp field contains the last time at which the marking on the dependence was changed by the user. It refers only to changes in the classification mark, ignoring other edits of the dependence information.

- **User**

  The user field contains the user id of the last user to change the dependence's status.

- **Reconsider Field**

  This field will contain a mark that is set if reanalysis has recomputed this dependence and found that it "matched" a dependence that the user had previously categorized. The reconsider mark can be cleared using the marking filter mechanisms. For example, the user may want to clear the reconsider mark on all matching dependences previously marked accepted since it is certainly safe to leave them accepted. Also included in the field will be an uneditable system comment which explains the reason for the mark being set.

3.2.3 Features of the Variable Display

The following will be included with each variable entry: the *variable name,* whether it is *shared or private,* whether the user *accepts or rejects* the system's classification, a *comment* about the classification, a *time stamp* of the most recent classification, and the *user* that reclassified the variable, and a *reconsider field* similar to that for dependences.

- **Name**

  The variable name and, if the variable occurs within a subroutine call, the name of the called routine are displayed here.

- **Shared, Shared Pending, Private**

  This field shows whether the variable is shared or private. Initially the field will be set by the system, but the user may reclassify any variable. If any of the dependences causing the variable to be shared are proven dependences, the variable is classified shared, otherwise it is shared-pending.

- **Comment - user**

  This field can be filled in at any time that the user wants to record information about the classification of the particular variable.

- **Info**

  This field contains the same information which is currently displayed to the user about a variable such as "defined before, used after". It is filled in at the time of analysis and is uneditable.

- **Time Stamp**

  The time stamp field contains the last time at which the marking on the variable was changed by the user. It refers to changes of the variables classification but does not reflect changes to other information included with each variable.

- **User**

  The user field indicates the last user to change the status.

- **Reconsider Field**

  The mark of the reconsider field is set on a variable if all of the dependences incident upon it match old dependences that the user previously classified. It is redundant with all the individual reconsider marks on the dependences, but it is useful for the user to notice that this variable classification depends on dependence edges which have been reanalyzed.

3.3 Changes to the Text Display

- **Line Numbers**

  Line numbers are optionally displayed in the text window. The dependence display includes the line number of the source to enable sorting dependences by line number and easy dependence navigation.

- **Loop Icon**

  Currently, a loop is selected by the same method that a user would select the cursor position for text editing. In the new model, a loop icon will be placed at the head of each loop and the user will click on the icon in order to select that loop. The icon of the selected loop will change to indicate that it is the selected loop. A persistent loop icon

will indicate the selected loop for the entire time the loop is selected. The loop icon will be different for loops which can be made parallel and loops which have dependences that prevent parallelization.

- **Annotations**

  A column to the left of the text displays information about the emphasized object in the text. Information such as "src", "sink", or "loop" is displayed. These sideline annotations are in addition to annotations embedded within the text. Embedded annotations are more specific but sideline annotations stand out better. To conserve screen real estate and avoid an overly cluttered display, sideline annotations will overwrite the line number in the line number display. A user can view line numbers, annotations, or both.

- **Indicating the src and sink of a dependence**

  Dependence indicators are embedded in the program text. The source reference of a selected dependence is underlined with the tail of an arrow and the sink with the head of an arrow. A reference which is both the source and sink of a dependence will be underlined with a complete arrow.

- **Ellipsis**

  In order to be able to display the statements containing the src and sink of a dependence on one screen, intervening text may need to be omitted. Ellipses are placed to indicate text omissions using NED technology. The amount of context displayed around the statements of interest can be manipulated by the user. The context can be a fixed number of lines or a function of the structure of the text. Ellipses can nest and the user can condense and expand them. If the source or sink of a dependence which is to be displayed falls within an ellipsed portion of text, the text will remain ellipsed and the source or sink name is indicated with a sideline annotation next to the ellipse.

- **Program Editing and Filtering**

  Complete editing and filtering functionality are to be added to PED. This enables general editing and parallelizing in one session. Users will be able to filter, search, and manipulate the text display just as in NED.

4 Additional User Interface Functionality

The following are suggestions to improve the current PED user interface. These do not involve any new model or theory, but are straightforward extensions to the current interface.

4.1 General Display Functions

A great deal of information about dependences and variables is available to the user, and the amount is increasing. It is impossible to cohesively display all the information at once, therefore the user must be able to choose what to see. The following two features allow the user to choose how to view information.

- **Control over which fields are displayed**

  The user will be able to select which fields of any display - text, dependences, and variables - actually show up on the screen. A simple dialog is provided, showing the list of possible fields with a way for the user to select those to present. One of two mechanisms will be used in the dialog. The user will either check off those to display, or, alternatively, move fields between a show box and a hide box. It would be nice if the user could layout the order in which fields were displayed as well. This, however, is difficult to add and is not of research interest.

- **Window spawning**

  The user can spawn off windows of text, dependences, and variables enabling more control of what can be seen at once. As yet to be determined is whether windows should spawn off individually or in related groups. Implementation of this feature will be postponed until other more critical features of the new design are completed. If a general *preference editor* is available, spawning would be handled using it.

4.2 Navigation

The following navigation aids will be provided in a future implementation of PED.

- **Searching via the standard "Find" dialog**

  Users will search through the text, dependence, and variable display using the NED "Find" dialog.

- **From uses to definitions and definitions to uses**

  Users will be able to navigate from a use of a variable to definitions of the variable or from a definition of a variable to its uses. The definitions and uses are a conservative estimate of the actual ones.

- **From source or sink in the dependence display to source or sink in text**

- **From a variable in the variable display to the variable in the text**

  Users will be able to select a variable in the variable display and navigate to uses and definitions. If the variable is classified by ParaScope as shared, the user will be able to navigate to the statement(s) which necessitated the shared status.

- **The call graph** - with parameter bindings

  Users will be able to navigate around the call graph, diving into called procedures, functions, and subroutines, and ascending to caller routines.

4.3 Storing the Ped session

Currently the PED user may only save the source of a PED session into the file system. It will be possible to save the following information into the ParaScope database between editing sessions. Most of this work is straightforward.

- **The Source**

  When regular section information and data windows are available, efficiency could be improved by storing subroutines separately. This way the whole program would need not reside in memory during editing. Call sites would already have information computed about the callee. The time required to bring in the source from the disk and the frequency of refreshing the source must be weighed against the savings in memory.

- **Current Dependence and Variable Markings**

  In the new dependence model, rejected dependences remain in the dependence graph but are ignored when calculating program transformations. At the end of an editing session, the user's markings must be stored along with the dependence graph. Keeping around rejected dependences does not cause the dependence graph to increase in size because dependences are only kept as long as conservative analysis cannot determine that they do not exist.

- **Macros for Viewing and Marking Dependences**

  Users may have generic viewing and marking filters that will be run often during their PED sessions. It would be helpful to store these sequences as "macros".

- **Window Position, Sizes, Field Selection**

  Users will not want to have to customize their PED session each time they edit a file so this information must be saved and restored between sessions.

5 Future Directions for the Ped User Interface

The following is a brief list of user interface issues which will be addressed in the future, but are beyond the scope of this document.

- **Semantic User Assertions**

  Eventually, the user will make general statements about program semantics and behavior and PED will determine which dependences can be removed based on these user assertions.

- **Editable Assertion Histories**

  When PED is able to "understand" semantic user assertions, the user will create a pool of facts about the program and should be able to add to, delete from, and modify these facts at any time.

- **Regions**

  In order to facilitate experimentation in distributed memory computing, and in task level parallelism, the user must be able to select program parts other than loops for purposes of analysis (for example subroutines). A convenient easy interface for selecting these "regions" must be developed, addressing such issues as region structure summarization and region expansion and narrowing.

- **Detecting and Displaying Redundant Synchronization**

  Analysis of event style synchronization is currently supported only on demand. More comprehensive analysis and display of its results are needed.

6 Summary

This document is intended as a blue print for user interface of the next generation of the ParaScope Editor. This interface was designed in response to user needs and requests and hopes to facilitate easier and more in depth analysis and transformation of programs as they are developed for shared-memory parallel machines.

References

[BKK+89] V. Balasundaram, K. Kennedy, U. Kremer, K. S. McKinley, and J. Subhlok. The ParaScope Editor: An interactive parallel programming tool. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[KMT91] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.