iPablo User's Guide

Keith A. Shields*

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

November 6, 1992

Copyright © 1992 The University of Illinois Board of Trustees.
All Rights Reserved.

*Supported in part by the Defense Advanced Research Projects Agency under DARPA Contract Number DABT63-91-K-0004, by the National Science Foundation under grants NSF CCR87-06653 and NSF CDA87-22836, by the National Aeronautics and Space Administration under NASA Contract Number NAG-1-613, and by an equipment grant from the Digital Equipment Corporation External Research Program.

Contents

1 Introduction **1**

2 Instrumentation Environment Overview **1**

3 An Instrumentation Example **3**
3.1 Starting *iPablo* . . . 3
3.2 Loading A Source Code File . . . 3
3.2.1 The File Selection Dialog . . . 4
3.2.2 File Parsing . . . 5
3.3 Exploring *iPablo* . . . 10
3.3.1 The Menu Bar . . . 10
3.3.2 The Routine List . . . 11
3.3.3 The Source Code Viewer . . . 11
3.3.4 Line Instrumentation Buttons . . . 11
3.4 Instrumenting Source Code Lines . . . 12
3.5 Generating An Instrumented Source File . . . 12
3.6 Exiting *iPablo* . . . 16

4 Generating Instrumented Source Code **16**

5 Global Instrumentation Options **17**

6 Selectively Instrumenting Specific Routines **18**

7 Instrumenting A Routine Subset **19**

8 *iPablo*'s Command Line Options **20**
8.1 Editing `cpp` Options Within *iPablo* . . . 21

9 An Advanced Example **21**
9.1 Instrumenting Example Code . . . 21
9.2 Compiling The Example Code . . . 22
9.3 Executing Example Code . . . 23
9.4 Processing Example Trace Output Data . . . 23

List of Figures

1 An Overview of the Pablo Instrumentation Environment 2
2 Matrix Multiply Example (Part 1) 3
3 Matrix Multiply Example (Part 2) 4
4 Matrix Multiply Example (Part 3) 5
5 The *iPablo* Main Window 6
6 The *iPablo* File Selection Dialog 7
7 The *iPablo* Working Dialog 8
8 *iPablo* After `MatrixMult.c` Has Been loaded 9
9 An *iPablo* Help Dialog 10
10 *iPablo* Trace and Count Symbols 13
11 *iPablo* Instrumented Source Information Dialog 14
12 *iPablo* Verify Dialog 15
13 A Portion of the Instrumented Matrix Multiply Source Code 24
14 The *iPablo* Subset Dialog 25
15 The *iPablo* `cpp` Editing Dialog 26

1 Introduction

This document is a guide to the use of the *iPablo* graphical user interface to the Pablo Instrumentation Environment. Although using *iPablo* is not mandatory, it will allow you to quickly and easily specify the most common source code instrumentations without laboriously modifying your source code manually.

Because this document is intended as a user's guide to the *iPablo* graphical interface of the Pablo Instrumentation Environment, its focus is on using *iPablo* to instrument application program source code in a variety of ways. The philosophy and design of the Pablo Instrumentation Environment itself, as well as some more advanced instrumentation features that are not currently supported by the *iPablo* graphical interface, are not discussed here. Interested readers may wish to refer to the *Pablo Instrumentation Environment User's Guide* for more details.

iPablo is a Motif-based, X Windows application that currently supports the interactive instrumentation of procedure calls[1] and outer loops in C source programs. Future versions of *iPablo* will support the interactive instrumentation of Fortran 77 programs as well.[2]

At present, *iPablo* supports the specification of two types of source code instrumentation: `COUNT` and `TRACE` instrumentation points. These instrumentation points can be specified globally for all procedures and/or outer loops, locally for a specific procedure, or on a line-by-line basis. Marking a code location to be traced directs the Pablo Instrumentation Environment to generate a dynamic, timestamped trace of the execution of that code point; the generated data will include the time the event occurred, a unique instrumentation point identifier, and the event's duration. In contrast, marking a code location to be counted will generate a single dynamic event that includes the number of times the specified code was executed.

In §2, we will briefly describe the Pablo Instrumentation Environment and *iPablo*'s place within it. The remainder of this document will focus on learning by doing. We will discuss the major features of *iPablo* through the use of several small instrumentation exercises. By the end of this document you should be able to quickly and easily instrument large codes in a variety of ways.

In §3 we use a simple matrix multiplication program to demonstrate the basic features of *iPablo*. §4 contains a more in depth look at what happens when you generate instrumented source code. This is followed in §5 with a demonstration of *iPablo*'s global instrumentation features, and in §6 by a discussion of *iPablo*'s routine specific instrumentation capabilities. Finally, we will instrument a larger, iterative partial differential equations (PDE) solution code and the mechanisms to create an executable version of the instrumented source code. This PDE code and associated performance data is also used in the tutorials on the Pablo Instrumentation Environment and the Pablo Performance Visualization Environment.

2 Instrumentation Environment Overview

iPablo is but one small small component of the Pablo Instrumentation Environment. Figure 1 illustrates the components of the instrumentation environment, and the interaction of the *iPablo* graphical user interface with the source code parser and the performance data capture library.

[1] We use procedure in its generic sense, including both procedures and functions.

[2] In a collaborative effort with the Rice Center for Research in Parallel Computing, we also are integrating Pablo with the Rice Fortran D compiler, providing support for automatic instrumentation of data parallel Fortran programs.

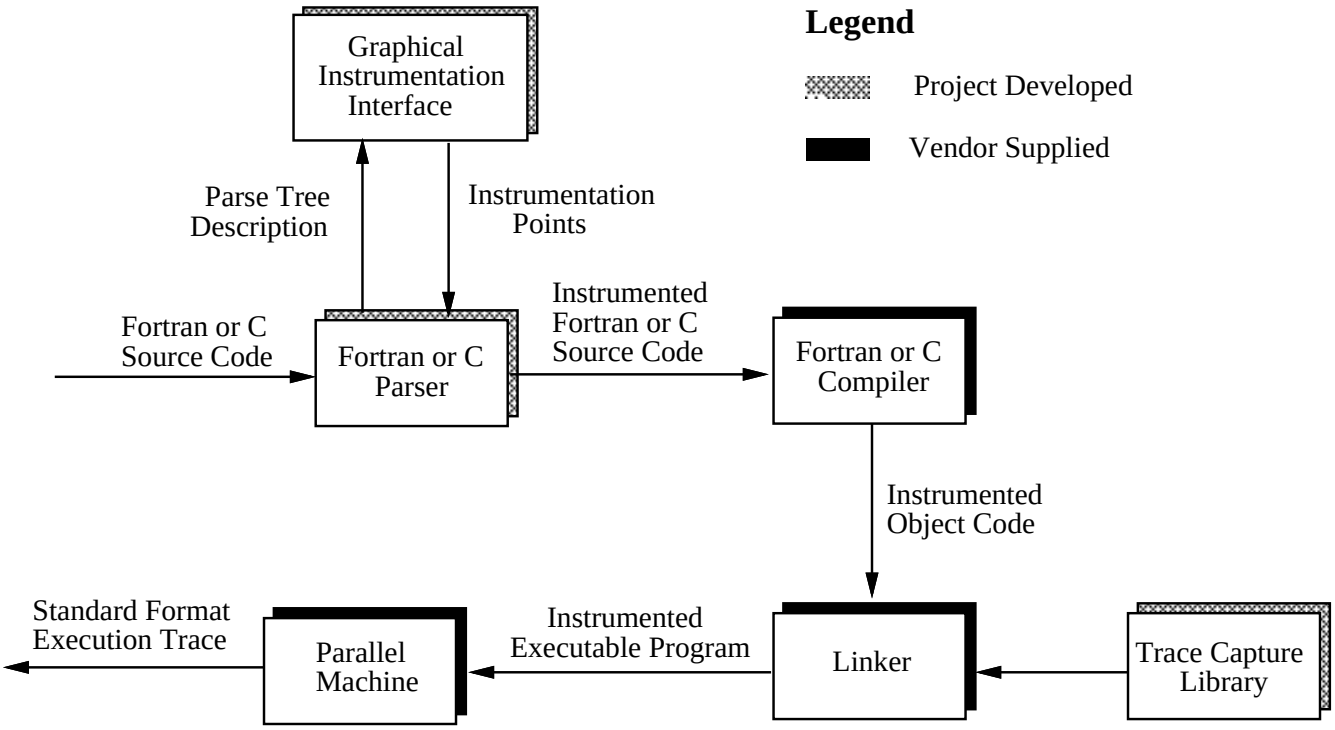

Figure 1: An Overview of the Pablo Instrumentation Environment

At present, the instrumentation environment includes a C parser that identifies the potentially instrumentable points in the specified source code files. The *iPablo* interface allows one to interactively select and enable all or a subset of these instrumentation points. These instrumentation points are then used to generate a modified version of the source code that contains calls to the Pablo performance data trace capture library at the instrumentation points. This modified source code is then compiled using the target machine's native compiler and linked with the Pablo trace capture library. During program execution, each execution of an instrumented construct passes data to the trace capture library, where it is buffered, timestamped, and written to secondary storage for subsequent analysis by the Pablo Performance Analysis Environment.

At present, versions of the trace capture library exist for single processor Unix systems (e.g., Unix workstations) and for the distributed memory Intel iPSC/2 and iPSC/860; all other portions of the Pablo instrumentation system are independent of a particular architecture. All machine-specific details are isolated in portions of the Pablo trace capture library, allowing one to quickly port the instrumentation system to new platforms.

We will soon begin extending the trace capture library to include the Thinking Machines CM-5. Similarly, C is the only language for which a working instrumentation parser presently exists in the Pablo instrumentation environment. However, because both the *iPablo* interface and the performance data capture library are source code language independent, the completion of a Fortran instrumentation parser, now underway, will immediately permit interactive instrumentation of Fortran source code as well.

```
/*  Matrix Multiplication */

#include <stdio.h>

void printArray(matrixName, array, n)
     char *matrixName;
     int  *array;
     int  n;
{
  int i, j;

  printf("\n%s:\n\n", matrixName);

  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      printf(" %d ", array[n * i + j]);
    }
    printf("\n");
  }
}
```

Figure 2: Matrix Multiply Example (Part 1)

3 An Instrumentation Example

We begin our tutorial overview of the *iPablo* instrumentation interface by using a simple matrix multiplication code fragment as an example. This `MatrixMult.c` fragment, shown in Figures 2–4, will form the basis for several extended instrumentation examples.

3.1 Starting *iPablo*

Once you have successfully built and installed *iPablo*, you are ready to run the program. Because *iPablo* will write files to the directory in which it was invoked, it is best to start *iPablo* in a directory that contains source code that you wish to instrument. You can start *iPablo* from the command line by typing:

```
iPablo
```

A window similar to that in Figure 5 should appear on your workstation screen. *iPablo*'s main interface consists of four major regions. These regions are of little interest unless a source file has been loaded. As a result, the menu bar at the top of the window is the only sensitized (i.e., active) area. This menu bar is used to access the majority of *iPablo*'s functions, including loading a source file, which is the first step in generating instrumented code.

3.2 Loading A Source Code File

To specify a file to instrument, one must use the **`Load Source File`** option under the **`File`** menu to load the file; **`MatrixMult.c`** in this example. The easiest way to access the menu bar is via

```
int *multArrays(leftArray, rightArray, n)
     int *leftArray;
     int *rightArray;
     int n;
{
  int *resultArray, i, j, k;

  resultArray = (int *) malloc(sizeof(int)*n*n);

  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        resultArray[n * i + j] = 0;
        for (k=0; k<n; k++) {
            resultArray[n * i + j] += (leftArray[n * i + k]) *
                                      (rightArray[n * k + j]);
          }
      }
  }
  return resultArray;
}
```

Figure 3: Matrix Multiply Example (Part 2)

the mouse. Move the pointer up to the portion of the menu bar labeled **File** and press the left mouse button to see the pulldown **File** menu. While holding the left mouse button down, move the pointer until **Load Source File** is highlighted and then release the left button.

3.2.1 The File Selection Dialog

Selecting **Load Source File** will create a file selection dialog; at this point your screen should look similar to Figure 6. The top portion of the file selection dialog is used to indicate a directory path and file specification. These determine which directories and files you will see in the two lists below the filter window. Because *iPablo* currently supports C source programs, the default filter will be all programs in the current directory with a **.c** extension.

You can move through the directory hierarchy by selecting a directory in the directory list with the left mouse button and then pressing the **Filter** button. This effectively moves into that directory and finds all files that meet the filter requirements of having a **.c** extension. Double clicking with the left mouse button on a directory has the same effect as pressing **Filter**.

Clicking on a file in the file list with the left mouse button causes that file to be selected. The selected file will be shown in the selection window below the lists. Pressing **OK** will cause the selected file to be loaded into *iPablo*. Double clicking on the desired file with the left mouse button will do the same thing as choosing **OK**. Pressing **CANCEL** at any time during the file selection process will remove the file selection dialog and abort the source file load sequence.

Using the techniques described above, move from your current directory to the root directory of the main *Pablo* distribution. This will be the directory that was specified when the *Pablo* distribution was installed. Move down from this directory until you reach

```
main(argc, argv)
     int argc;
     char *argv[];
{
  int n = 2;
  int A[2][2] = {{1, 2}, {3, 4}};
  int B[2][2] = {{5, 6}, {7, 8}};
  int *C;
  char buf[80];

  printf("PROGRAM BEGIN\n");
  sprintf(buf,"Matrix A");
  printArray(buf, A, n);
  sprintf(buf,"Matrix B");
  printArray(buf, B, n);

  C = multArrays(A, B, n);
  sprintf(buf,"The Result of A times B is");
  printArray(buf, C, n);

#ifdef BTIMESA
  C = multArrays(B, A, n);
  sprintf(buf,"The Result of B times A is");
  printArray(buf, C, n);
#endif /* BTIMESA */

  printf("PROGRAM END\n");
}
```

Figure 4: Matrix Multiply Example (Part 3)

distribution-root/`Instrument/Parser/Examples/MatrixMultiply`

where *distribution-root* is the path to the root of the source code distribution in your installation.

When you reach the `MatrixMultiply` directory, you should see `MatrixMult.c` in the file list. Click on it with the left mouse button and press `OK` (or alternatively just double-click on the file name) and `MatrixMult.c` will be loaded.

3.2.2 File Parsing

Once you have successfully loaded `MatrixMult.c`, it will automatically be parsed to inform *iPablo* about instrumentable structures within the code. In general, the amount of time required for the parsing process depends on a program's size and structure. `MatrixMult.c` is simple enough that the time to parse it is negligible.

While a file is being parsed, a working dialog will be appear to inform you that parsing is in progress, as illustrated in Figure 7. During this time, the parser has control of the program. As a result, the X Windows System may not be able to immediately update any of the *iPablo* windows,

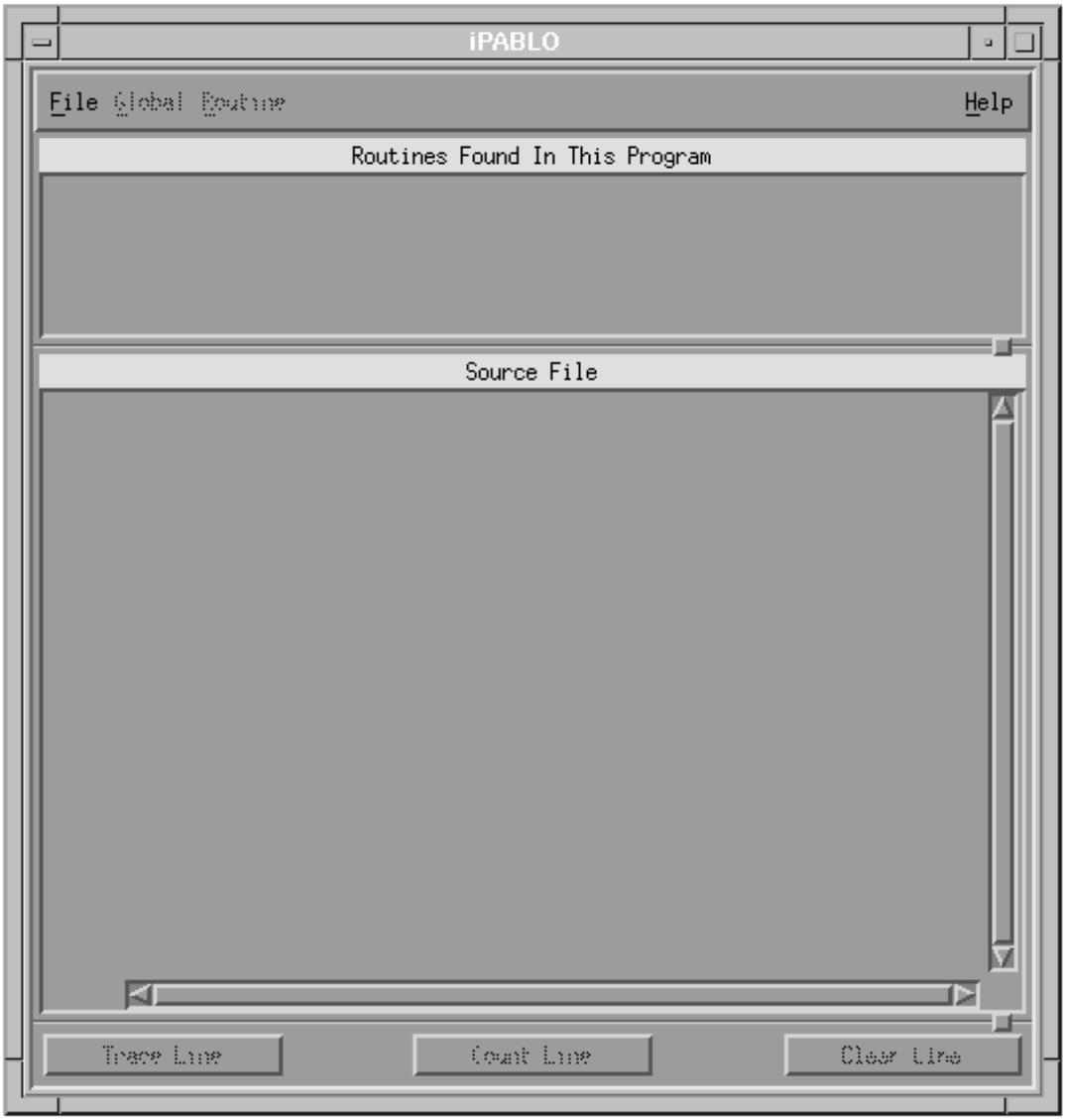

Figure 5: The *iPablo* Main Window

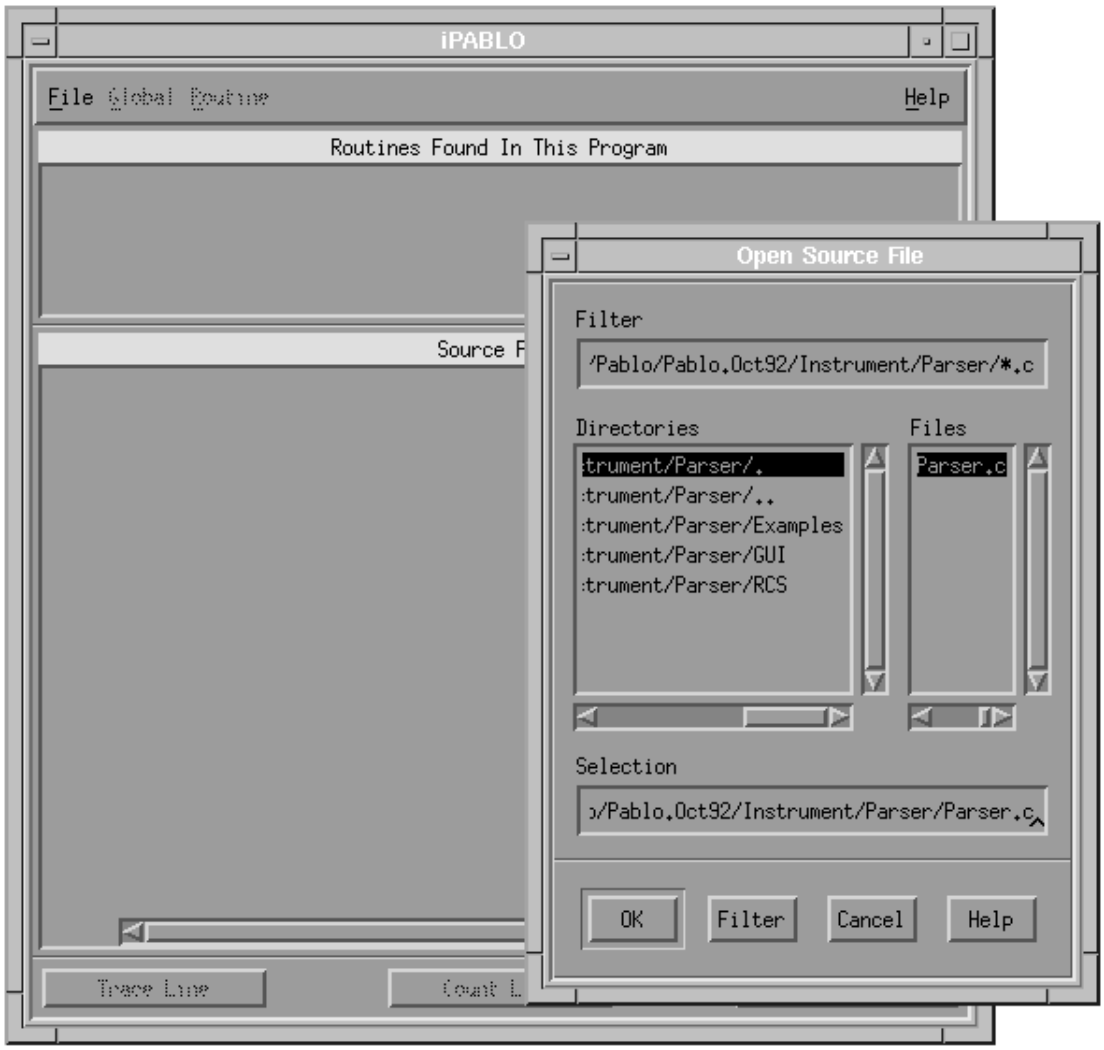

Figure 6: The *iPablo* File Selection Dialog

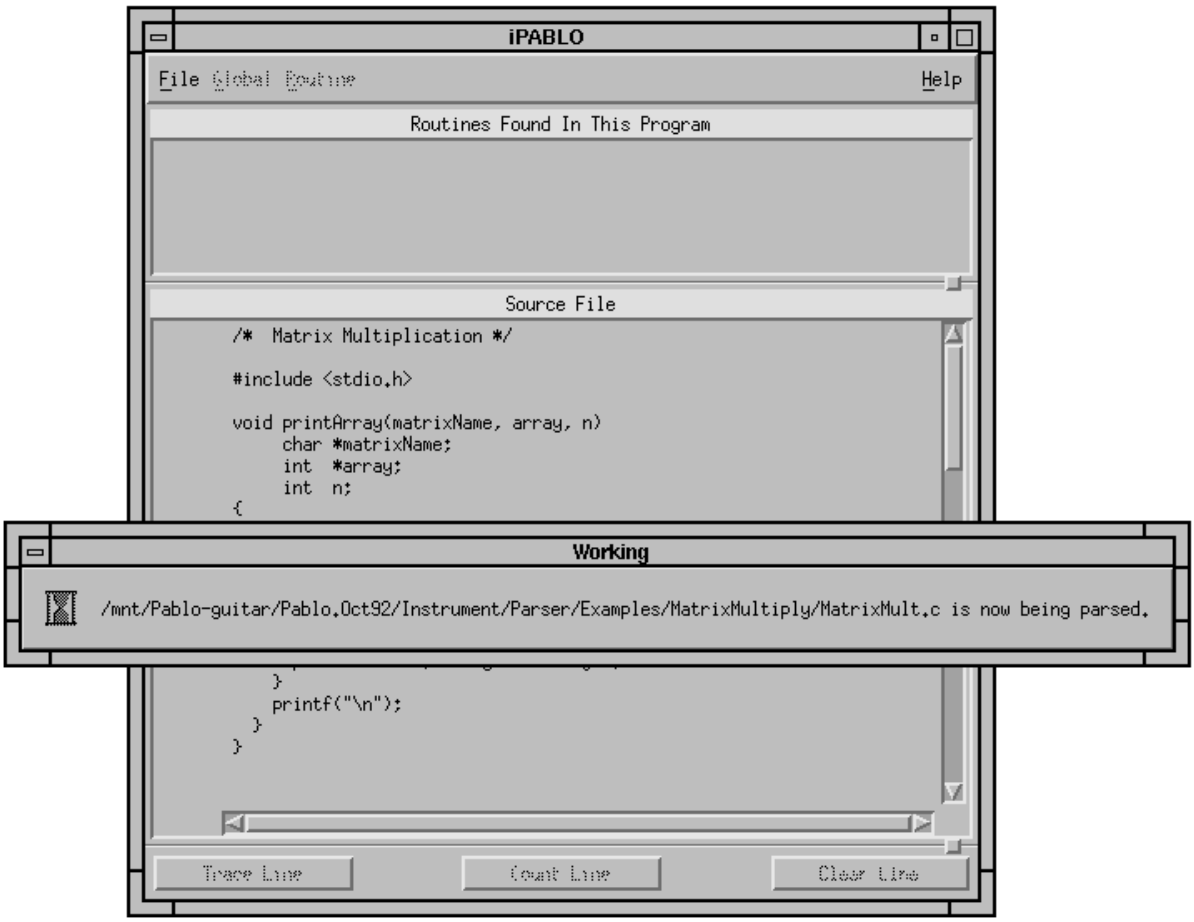

Figure 7: The *iPablo* Working Dialog

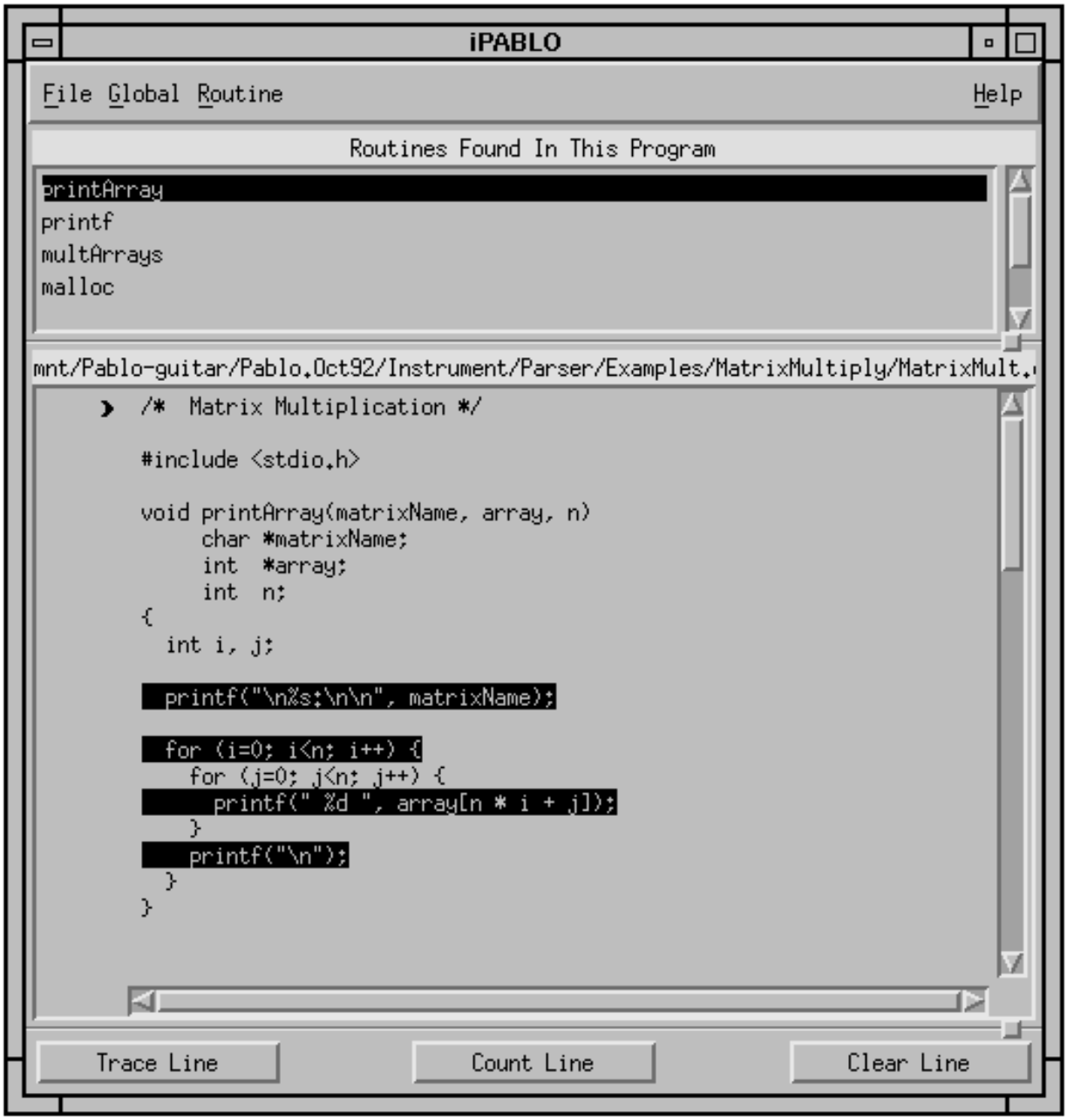

Figure 8: *iPablo* After `MatrixMult.c` Has Been loaded

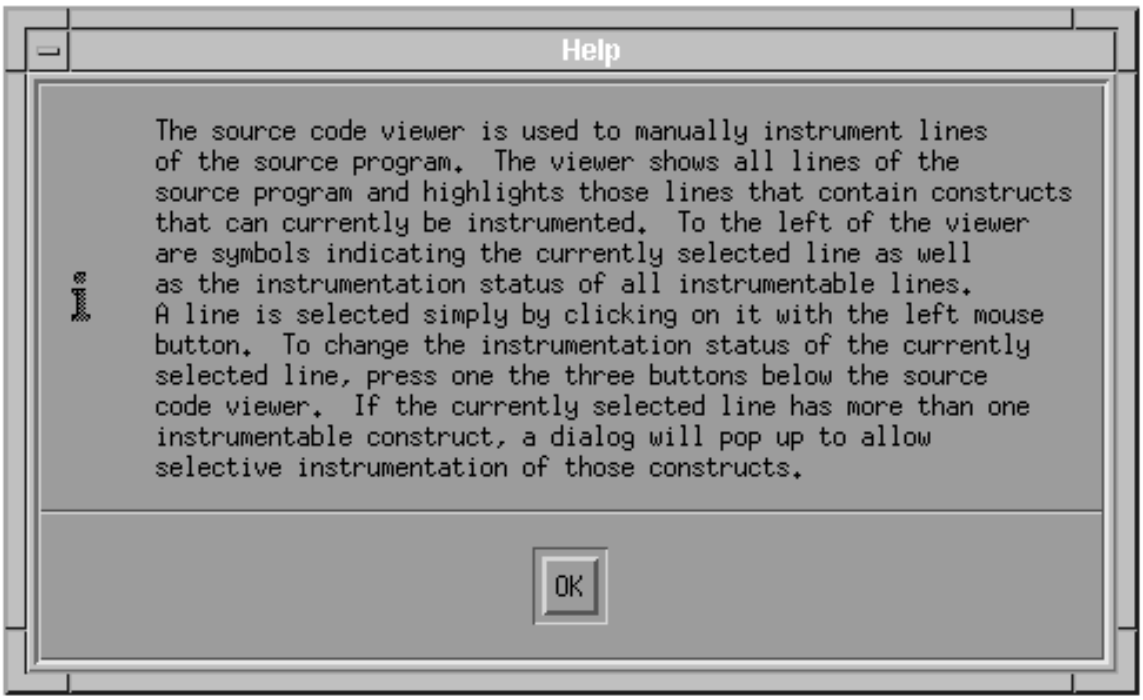

Figure 9: An *iPablo* Help Dialog

so they may look awkward if they are obscured by other windows and then unobscured. This is simply an artifact of the X Window System and not a bug. Following the successful parsing of `MatrixMult.c`, the *iPablo* interface should appear similar to Figure 8.

At this point, you may well ask what happens if there are syntax errors in the source code. *iPablo* will inform you if it is unable to properly parse a file, but the design assumption of the instrumentation system is that the code is free of syntax errors. If not, you should use the native compiler on your system to correct those errors before attempting to instrument your code — performance data capture and analysis follows, rather than preceeds, debugging.

3.3 Exploring *iPablo*

Now that we have successfully loaded and parsed `MatrixMult.c`, we can turn our attention to the features of the *iPablo* instrumentation interface. Below, we describe both the components of the interface and their operation.

3.3.1 The Menu Bar

As was mentioned earlier, the menu bar at the top of *iPablo*'s main window is used to access most of the major funtions in *iPablo*. You have already seen its use in §3.2.

Looking more closely, we see that the menu bar has four pulldown menus. We have already used the `File` menu to load `MatrixMult.c`, and we will return to it shortly. Because the `Global`

and `Routine` menus are used for more sophisticated instrumentation, we defer their discussion until later. Not surprisingly, the `Help` menu provides access to interactive help. Selecting one of these topics will cause a help dialog on that topic to appear. Click on `OK` when you are done with the help dialog to pop it down. Figure 9 shows an example help dialog.

3.3.2 The Routine List

Below the menu bar is the routine list. This is a scrollable list of all routines found within the source program.

Clicking on the scrollbar, which is located on the right side of the list, with the left mouse button will cause the list to scroll. The exact behavior of the scrolling depends on which part of the scroll bar you select. Clicking on the top or bottom arrow will cause the list to scroll down or up by one item. Pressing and holding the left button on the slider, located in the middle of the scrollbar, while moving the mouse up or down will cause the list to scroll more rapidly.

An item in the routine list can be selected by clicking the left mouse button when the pointer is over the item.

Is is important to realize that the routine list contains *all* procedures found in the source code file, whether they are called by other procedures, make calls to other procedures, or both. Simply put, the routine list includes all procedures whose body was found in the current file and all procedures called from procedures in the current file.

3.3.3 The Source Code Viewer

Below the routine list is the source code viewer. This is a scrollable region of text used to instrument specific lines of the source file. The scrollbars for the source code viewer behave exactly the same as the ones for the routine list.

The first thing you will notice about the source code viewer is that parts of the source code are highlighted. Upon closer examination, you will find that the highlighted lines of text correspond exactly to the procedure calls and outer loops of the program. Therefore, the highlighted lines of text are precisely those lines that can be instrumented.

You will also notice that there is a arrow to the left of the first line of the code. This is used to indicate the currently selected line of code. A line of code is selected simply by clicking the left mouse button on any part of the line; try moving to another line in the file.

3.3.4 Line Instrumentation Buttons

At the very bottom of the *iPablo* window is a set of three buttons. These buttons are used to indicate which form of instrumentation should be applied to the currently selected line.

As noted in the introduction, *iPablo* currently allows one to specify two different sorts of instrumentation events for a particular instrumentable structure. Marking a code location to be traced will generate a dynamic, timestamped trace of the execution of that code point; the generated data will include the time the event occurred, a unique instrumentation point identifier, and the event's duration. In contrast, marking a code location to be counted will generate a single dynamic event that includes the number of times the specified code was executed.

Two of the Line Instrumentation Buttons correspond to specifying *Trace* and *Count* events for a particular line of the source code. The third button is used to clear any instrumentation that is

being applied to a particular line of code.

3.4 Instrumenting Source Code Lines

Now that we have explored some more of *iPablo*'s features, we can proceed with this example and actually instrument some code.

If you no longer have `MatrixMult.c` loaded in *iPablo*, go ahead and reload it, referring to earlier parts of this section if needed. If you still have `MatrixMult.c` loaded, scroll the source code viewer until you are at the top of the file. You should now be looking at the definition of the `printArray` procedure.

We will begin by instrumenting two lines of code. Select the line containing the first `printf` statement. This line has the form:

```
printf("\n%s:\n\n", matrixName);
```

Once you have selected the line, go to the Line Instrumentation Buttons and choose `Trace Line`. You should see a symbol appear to the left of the line selected arrow. This symbol is used to indicate that the given line has been traced.

Now select the line with the `For Loop`, which is just below the `printf` you just instrumented. This line has the form:

```
for (i=0; i<n; i++) {
```

Note too that this is the only `For Loop` highlighted for this function; it is the only *outer* loop found in `printArray`().

Once you have selected this line, press the `Count Line` button at the bottom of the *iPablo* screen. You should see the symbol used to indicate that a line has been counted appear to the left of the line selected arrow. Your screen should now look similar to Figure 10.

3.5 Generating An Instrumented Source File

Now that you have successfully instrumented `MatrixMult.c`, you need to create a version of the file that incorporates the instrumentation. This is done with the `Generate Instrumented File` option under the `File` menu. When you choose this option, an information dialog will appear telling you the name of the instrumented version of the source file. Currently, the name of this file takes the form:

```
Instrumented_<filename>
```

where `filename` is the name of the original source code file. Notice that your original source code file is unchanged; *iPablo* saves the newly instrumented code in a separate file.

In the case of `MatrixMult.c`, a dialog of the form shown in Figure 11 will appear. Clicking on `OK` will dismiss the dialog and save the instrumented source code.

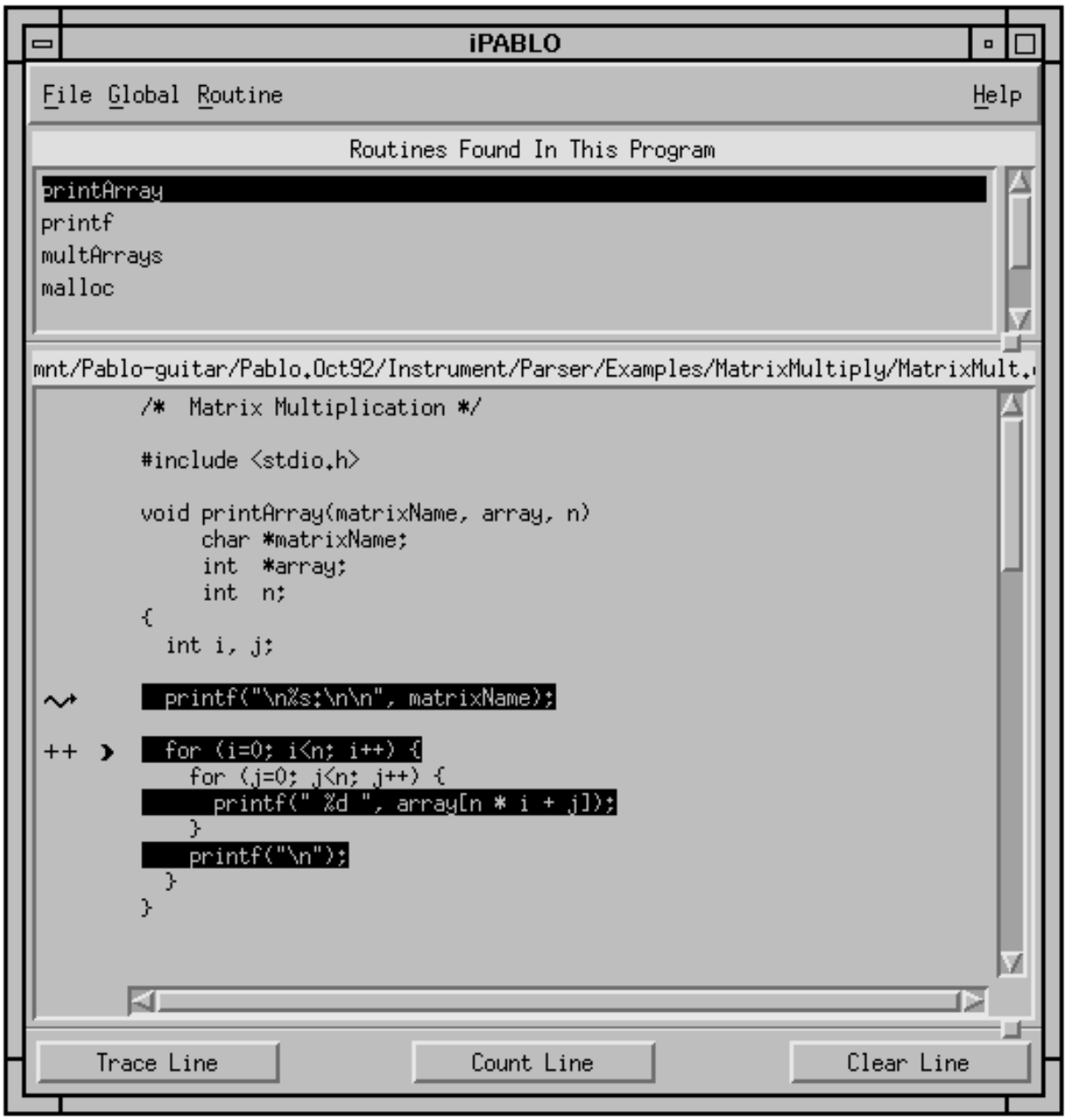

Figure 10: *iPablo* Trace and Count Symbols

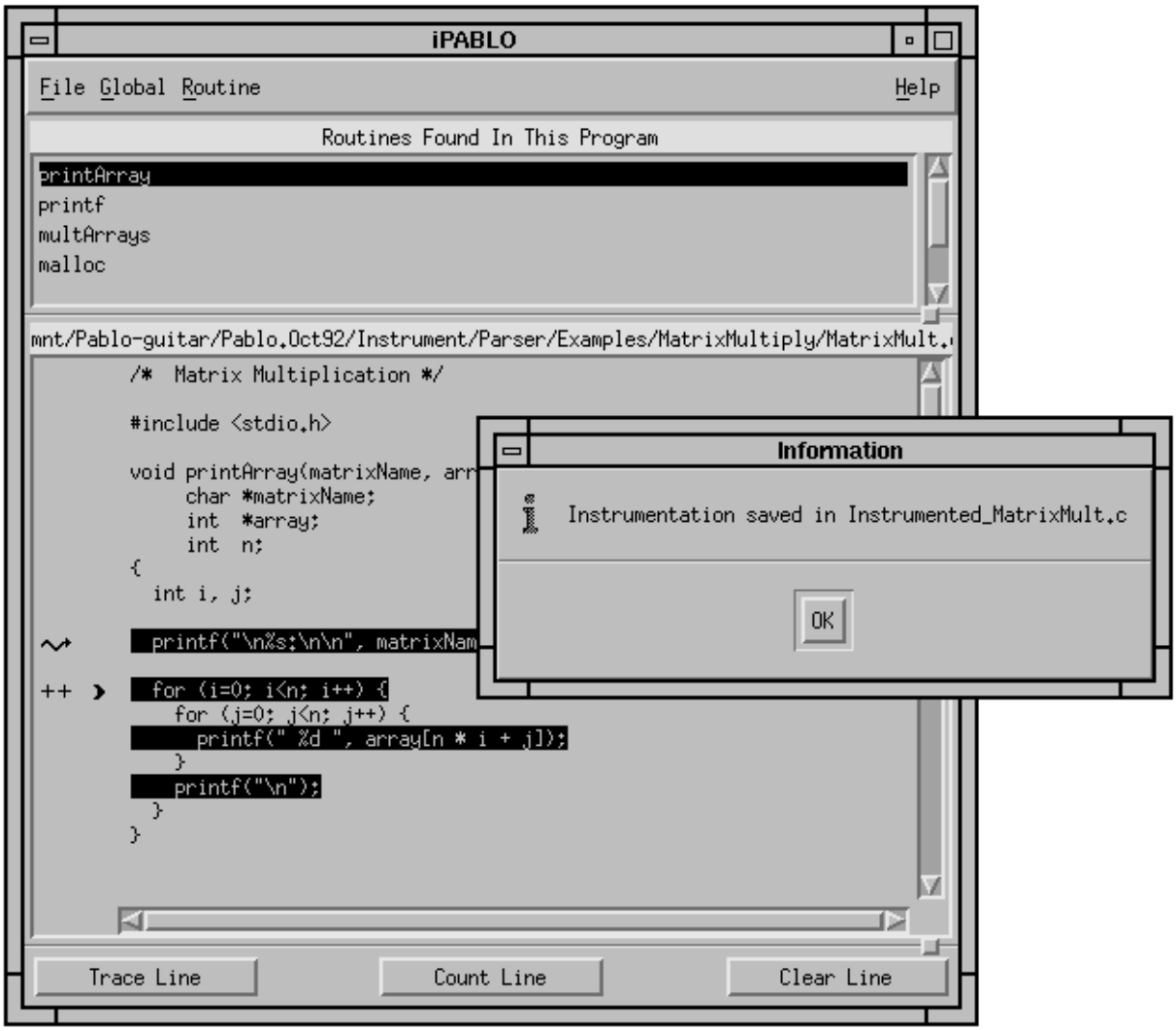

Figure 11: *iPablo* Instrumented Source Information Dialog

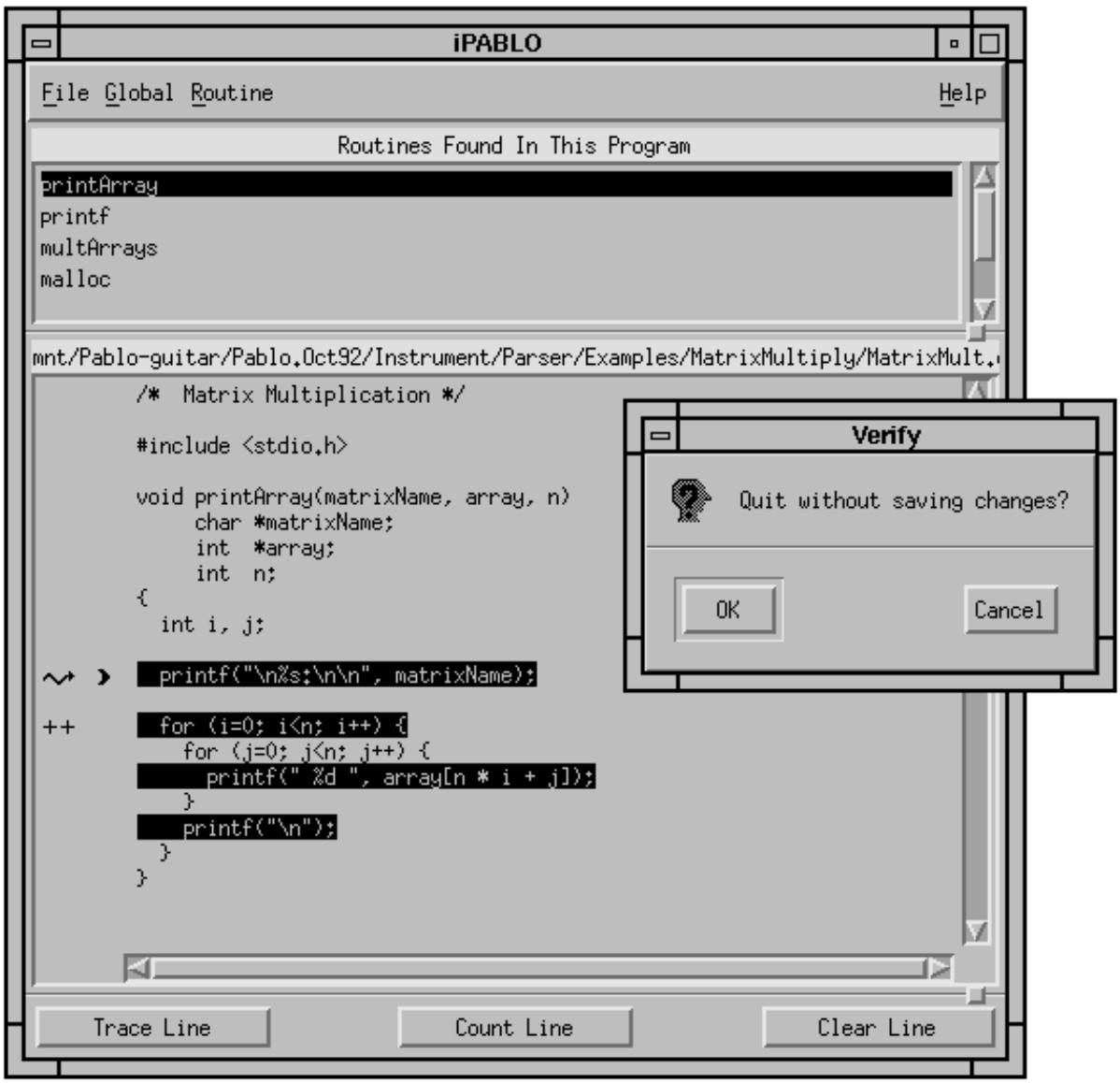

Figure 12: *iPablo* Verify Dialog

3.6 Exiting *iPablo*

Now that you have successfully generated an instrumented version of the original source code, this example is complete. All that remains is to exit *iPablo*; this is done via the `Quit` option under the `File` menu. Because you just saved the instrumented `MatrixMult.c` code, *iPablo* will exit immediately.

If you had changed the enabled instrumentation since the last time the instrumented code was saved via `Generate Instrumented File`, *iPablo* would have prompted you to verify exit using the dialog shown in Figure 12

4 Generating Instrumented Source Code

Now that you have completed the first instrumentation example, let's examine the effects of *iPablo*'s actions. The *iPablo* instrumentation interface writes all files to the directory from which it was invoked, regardless of the directory from which the source code file was loaded (i.e., if you used the Motif file selector dialog to traverse a directory tree prior to loading a source file, the instrumented code will still be written in the directory from which *iPablo* was invoked). In the directory where *iPablo* was invoked, you will notice that *iPablo* created several files in addition to `Instrumented_MatrixMult.c`, the instrumented `MatrixMult.c` source code. We will examine each of these files and explain their functions.

An examination of the `Instrumented_MatrixMult.c` file will show that it is quite different from the original `MatrixMult.c` code. To properly parse the C source file, the C instrumentation parser must include all header files and expand all macros. This inclusion and expansion is performed by the C preprocessor (`cpp`), which is called by *iPablo* before parsing begins. The input to the parser is simply the output of `cpp`.

Because the parser cannot distinguish among the user-written code, include files, and macro expansions, it is not possible for the instrumentation system to synthesize an instrumented source code file that is identical to the original source save for the addition of instrumentation code. Instead, *iPablo* and the instrumentation parser must generate an instrumented file based on the output of `cpp` with included header files and expanded macros. Although this may seem momentarily distressing, all is not lost. As we shall see, one can repeatedly change the instrumentation points in the source code without knowledge of the unaesthetic code generated by the instrumentation system.

If you scroll through the `Instrumented_MatrixMult.c` file generated by the instrumentation system, you will see eventually encounter source code that is reminiscent of the original `MatrixMult.c` code; see Figure 13. Reflecting the instrumentation we earlier specified in the example, you will see that calls to the Pablo trace library have been inserted to bracket the first `printf` and the first `For Loop` in the `printArray` procedure. During execution, these *traceEvent* and *countEvent* routines pass the necessary data to the Pablo trace library for recording. At this juncture, the parameters to these calls are unimportant. It is only important to understand that instrumented procedure calls and outer loops are "wrapped" by calls to the Pablo trace library.

The next file of interest is `InstrumentationInitialization.c`. This file contains the initialization code for the Pablo trace library. The Pablo C instrumentation parser synthesizes a new `main()` function for each program.[3] The synthesized `main()` function initializes the identifiers used

[3] Had the `MatrixMult.c` program consisted of multiple files, only one new `main()` routine would have been created.

within the Pablo trace library to uniquely specify each instrumentation point. The synthesized `main()` function then calls a renamed version of the original program's `main()`.

Because an application program may span multiple files, the *iPablo* instrumentation interface must record and correlate identifiers across all the files, or equivalently, across multiple instrumentation sessions (e.g., if one instrumented some of the files one day and the remainder the next). The correlation of identifiers across all the files that constitute a program is maintained via a `.project` file. This file is used to correlate all event identifiers within a given directory. Whenever you load a source code file in *iPablo*, the `.project` file is referenced to ensure that all event identifiers are unique and that the same event (such as a procedure call) found in multiple source files will be given the same event identifier. The `.project` file and the identifiers found in the generated `InstrumentationInitialization.c` file are the same — the event identifier information in the `.project` file is used to synthesize the `InstrumentationInitialization.c` file.

Although a single `.project` file and `InstrumentationInitialization.c` file may suffice for all instrumentable files within a given directory, *iPablo* must also record which constructs in each source code file were instrumented. In general, the names of these files take the form:

.*fileName*.config

where *fileName* is the name of the source code file. Hence, in our example, the instrumentation configuration for the `MatrixMult.c` code will saved in the file `.MatrixMult.c.config`. The saved configuration, if it exists, is automatically loaded each time *iPablo* is invoked from the current directory and the the `MatrixMult.c` file is loaded. If you do not wish the configuration information to be loaded with the source code file, simply delete the configuration file.

5 Global Instrumentation Options

Now that you have completed the first example and are more comfortable with some of *iPablo*'s basic features, we are now ready to move forward and discuss other instrumentation options supported by *iPablo*

In the first example, we used the buttons at the bottom of the *iPablo* interface to instrument single lines of source code. Although this provides local control and permits one to selectively instrument particular code locations, it is not practical to instrument large programs via this mechanism — the user interaction costs are too high.

Global instrumentation allows one to apply a particular form of instrumentation (*Trace* or *Count*) to *all* procedure calls and/or outer loops in the current source file. Under the `Global` menu on the menu bar, you will find entries for

- `Calls`
- `Outer Loops`
- `Calls And Outer Loops`

Each of these menu entries has a submenu with entries for

- `Trace All`
- `Count All`

- `Clear All`

The behavior of these options is precisely what one would expect — they globally count or trace procedure calls, outer loop, or both.

To demonstrate the use of the global instrumentation, let us return to the `MatrixMult.c` code. First, start *iPablo* and load the `MatrixMult.c` file. If you have forgotten how to do this, refer to §3 for details.

As we discussed in the previous section, information regarding the current instrumentation status of `MatrixMult.c` was saved when the instrumented source file was generated; that configuration is automatically loaded when the `MatrixMult.c` file was loaded. Therefore, you will see the trace and count symbols we specified in the previous instrumentation of the `MatrixMult.c` file. To remove this instrumentation, select each instrumented line and press the `Clear All` button.[4]

After you have cleared the previous instrumentation, choose the `Trace All` option under the `Calls` portion of the `Global` menu. This will trace all function calls within the `MatrixMult.c` file. If you now scroll through the file, you will find that all procedure calls have the trace symbol next to them.

If you now choose the `Count All` option under the `Outer Loops` menu, you will cause all outer loops in the `MatrixMult.c` code to be counted. Note than none of the inner loops are affected because they are not instrumentable structures.

Finally, to clear all the instrumentation points, choose `Clear All` under `Calls And Outer Loops`.

6 Selectively Instrumenting Specific Routines

Now that you have mastered instrumenting constructs on a global level, it is time to look at another option that allows you to apply instrumentation at the procedure level. This can be very useful because you will often want to instrument just a few of the functions or procedures in a code, either because they are the only of interest or because the potential volume of performance data precludes the instrumentation of all routines.

Instrumenting particular routines relies on the routine list, the scrollbox just below the Motif menu bar; see §3.3.2 for details. Any selective instrumentation of routines will apply only to the currently selected routine on the routine list. You can use the options under the `Routine` menu to choose the type instrumentation to apply to the selected routine. Because the routine list contains both calling and called routines (i.e., routines found in the current file and routines called from the current file), the `Routine` menu has options for both `Calls To` and `Calls In` the currently selected routine, as well as an option for `Outer Loops Of` the currently selected routine.

For example, if one wished to trace all the procedure calls made in the `main()` function of the `MatrixMult.c`code, one would first select the `main()` procedure in the routine list and then select the `Trace All` option under `Calls In` of the `Routine` menu. If you do this and then scroll through the `MatrixMult.c` file, you will find that all functions called by `main()` have been traced.

Now suppose that one wished to count all calls to `printf`, regardless to the physical location in the code. We would first make `printf` the currently selected routine by choosing it from the routine list and then choose the `Count All` option under `Calls To` in the `Routine` menu. If you

[4] We discuss a faster way to clear all the instrumentation points within a source file in the next section.

do this and then scroll through `MatrixMult.c`, you will see that all calls to `printf` have been instrumented to record counts.

Assuming you instrumented the `MatrixMult.c` code as describe above (i.e., you enabled tracing for the calls made in `main()` and counting for the invocations of `printf`), an examination of the `MatrixMult.c` code would show that all function calls are traced, except for the calls to `printf`, which are counted. Because the *iPablo* interface does not permit simultaneous tracing and counting of a given construct, the most recently applied instrumentation choice takes precedence. Although `printf` was included when all functions called by `main()` were traced, the counting of calls to `printf` occurred last, so that choice is active.

7 Instrumenting A Routine Subset

In previous sections, we have discussed how to instrument all procedure calls that occur in a given procedure as well as how to instrument a single procedure call that occurs in a given procedure. Between these two lies a third alternative — instrumenting "some" of the procedures calls that occur in a given procedure. This is possible via a special dialog created by the `Instrument Subset` option under the `Calls To` and `Calls In` submenus of the `Routine` menu.

As in the previous examples, we illustrate the process of instrumenting a procedure call subset using the `MatrixMult.c` file. First, invoke *iPablo* and load the `MatrixMult.c` file. To ensure a common starting point, clear all instrumentation points you might have set in the file. Once you have done this, choose `main()` in the routine list and then select the `Instrument Subset` option under `Calls In` of the `Routine` menu. This informs the *iPablo* interface that you wish to instrument some of the procedures called by the `main()` routine. The `MatrixMult.c` is small enough that this is not really necessary, but there are many cases where this type of selective instrumentation functionality is extremely useful. A new dialog that allows selective instrumentation of a subset of procedures will now appear; your workstation screen should appear similar to Figure 14.

Below some informative labels at the top of the dialog is a scrolled list of procedures that contains all procedures called by `main()`. Elements of this list may be selected simply by clicking on a specific item with the left mouse button. You may also select a set of procedures that appear contiguously in the list by pressing the left mouse button and moving up/down the list while continuing to hold the left mouse button down. Once you have selected the desired set of functions, release the left mouse button and the items will be selected. Procedures that do not appear contiguously on the list can be selected by holding down the *Control* key while pressing the left mouse button on the procedure you wish to select.

Below the list of functions is a set of buttons labeled `Type Of Instrumentation To Apply`. These buttons are used to select the type of instrumentation that will be applied to the selected procedure calls. Because only one type of instrumentation can be applied to any given procedure call, only one of these buttons can be selected at a time. Below these buttons are two larger buttons labeled `Selected` and `Except`. These buttons let you indicate which group of items in the list should be instrumented (i.e., the selected items or everything but the selected items).

In this example, the subset list has four functions (`printf`, `sprintf`, `printArray` and `multArrays`). If you select both `printArray` and `multArrays`, select `trace` as the type of instrumentation to apply, and then chose `Selected`, you will trace all calls to `printArray` and `multArrays` found in `main()` while having no effect on `printf` and `sprintf`.

If you now select **printArray** and **multArrays** again, then select **Count** as the type of instrumentation to apply and then **Except**, you will count all calls to **printf** and **sprintf** without affecting the instrumentation status of **printArray** and **multArrays**.

8 *iPablo*'s Command Line Options

In the previous examples, we invoked *iPablo* with no command line options and we had to load **MatrixMult.c** by hand. In general, we would like to be able to specify the file to be loaded from the command line.

Invocations of *iPablo* can take three forms:

```
iPABLO

iPABLO filename

iPABLO cpp-options filename
```

We have already seen the first case. The second case simply allows you to specify a file that you wish to instrument. The file will then be automatically loaded and parsed. If the file is not in the current directory, a pathname may also be included so the file can be found. This option allows us avoid having to use the file selection dialog to initially load a file.

The third case also involves specifying the file to be loaded, but it also accepts **cpp** options that will be passed to the C preprocessor. The currently supported **cpp** options are:

- -D to define a value
- -U to undefine a value
- -I to specify an include path

These options are well documented in many places so we will refrain from discussing them in detail here. *iPablo* will ignore any unsupported **cpp** options.

We can see how these command line options work in practice by running *iPablo* with a few of them. We will be using **MatrixMult.c** again, so we can specify it as our file option. **MatrixMult.c** does not make use of any nonstandard include files so we do not need to specify any -I options. However, **MatrixMult.c** does have some additional statements that are compiled and executed only if **BTIMESA** is defined, so we can make use of the -D **cpp** option.

From the command line, invoke *iPablo* as follows:

iPABLO -DBTIMESA *distribution-root*/**Instrument/Parser/Examples/MatrixMultiply/MatrixMult.c**

where *distribution-root* is the path to the root of the source code distribution in your installation.

iPablo should come up with **MatrixMult.c** already loaded. By using the -D option, we told **cpp** that the region of code in **main()** between the *#ifdef* and *#endif* would need to parsed as well. This gives us several more instrumentable structures than we would have without the -D option since the region of code affected by -D had function calls in it.

8.1 Editing cpp Options Within *iPablo*

One problem with specifying `cpp` options from the command line is that different files may require different `cpp` options. It would be quite inconvenient if you had to restart *iPablo* every time you wanted to load a file that required `cpp` options other than the ones you specified on the command line.

As a result, *iPablo* supports editing of `cpp` options. The `Edit CPP Options` selection under the `File` menu will cause a dialog similar to the one in Figure 15 to be popped up.

This dialog provides a text field that allows you to edit `cpp` options. Initially, it will have any `cpp` options you specified when you invoked *iPablo*. You can edit these options whenever you like, and the new `cpp` options will be used the next time you load a file. [5] Press `OK` when you are done editing the `cpp` options.

9 An Advanced Example

In the Pablo source code distribution is an example of a parallel application that has been instrumented using *iPablo*. This example is an iterative elliptic partial differential equation (PDE) solver, which has been implemented for the Intel iPSC hypercube; the example is described in detail in the document entitled *An Informal Guide to Using Pablo*. Source code for this example can be found in the subdirectory `Parser/Examples/PDE` of the Pablo source code distribution.[6] The section below uses this as an advanced example of *iPablo*'s capabilities.

9.1 Instrumenting Example Code

While in the same directory as the PDE example source code, and with the *iPablo* program installed in a directory specified in your execution `PATH` environment variable, invoke the instrumenting parser interface from the shell by typing `iPablo`.

Before loading the source code file to be instrumented, it is necessary to define some C preprocessor (`cpp`) options. In addition to the instrumentation of procedure calls and loops with *iPablo*, the PDE example includes the tracing of calls to message passing routines.[7] To this end, the following `cpp` options should be defined either from the command line as discussed in §8 or by using `Edit CPP Options` as discussed in §8.1.

- `-DMESGTRACE`
- `-I../../../Tracelib/Extensions/SystemDepend/Intel/Includes`

The syntax of these options is precisely that supported by the C preprocessor; they are passed unchanged to the preprocessor by the *iPablo* interface.

After specifying the C preprocessor options, load the file `pdeNode.c`. For the sake of illustration, suppose we wished to trace the calls to a group of five specific procedures in this code. This can be done in many ways, but it is simplest to select `main` as the current routine, then instrument a

[5] It is not possible to have these modified options immediately applied since the file would have to be reparsed for the new `cpp` options to have any affect

[6] The `Examples` directory also contains the `MatrixMult.c` code used in the earlier examples.

[7] The mechanism for instrumenting message passing routines and the associated extension to the Pablo instrumentation library are discussed in the *Pablo Instrumentation Environment User's Guide*.

subset of the procedure calls within that function. The selected procedure calls are to `InitGrid`, `Iterate`, `IterateEdge`, and `Convergence`. Then select `Converged` as the current routine and trace all calls to that procedure. Finally, scroll down through the text of the program to the point (within `main`) where a loop beginning `for(Iter = 1; ... )` is defined. Selecting that as the current line, click on `Count Line`. Now generate the instrumented source code file and exit *iPablo*. The instrumented code is deposited in `Instrumented_pdeNode.c`.

9.2 Compiling The Example Code

Because the example (PDE) code was developed for an Intel hypercube, it executes in single-program-multiple-data (SPMD) mode — a copy of the code executes in parallel on each of the hypercube nodes. This, in turn, means that a copy of the Pablo trace capture library, which was linked with the program, also executes on each hypercube node. Each copy of the executing code, with its associated trace library, will generate a performance data trace file, reflecting the activity on that hypercube node. One must uniquely specify the name of the trace file produced by each hypercube node.

As §2 explained, the *iPablo* instrumentation interface and the C instrumentation parser are independent of a particular machine architecture; only the Pablo trace library contains architecture dependent features. Because of this architecture independence, it is not possible for the *iPablo* interface or the C instrumentation parser to synthesize machine-specific file names. Indeed, the parser is unaware that the code can potentially execute in parallel.

Because each of the parallel nodes executing this program should produce a different trace output file, we must manually edit the synthesized `InstrumentationInitialization.c` file to specific unique trace file names. To do this, look for a comment line reading:

```
/* call setTraceFileName( fileName ) here */
```

and replace it with the following:

```
char    traceFile[32];
sprintf( traceFile, "Jacobi\%d.bin", mynode() );
setTraceFileName( traceFile );
```

This code fragment uses the identity of each node to construct a unique file name; this name is passed to the Pablo trace capture library during initialization.

The instrumented code can now be compiled. The `Makefile` provided with the example code will help in compiling this example. By assumption, the Pablo instrumentation library has already been compiled for an Intel iPSC system and installed in your site's library directory. At most, it may be necessary to change the `TRACELIBDIR` and `EXTENSIONDIR` macros in `Makefile` to correspond to the local configuration, or perhaps the name of the Intel cross-compiler. Once this has been done, all that is necessary to compile this instrumented example is to execute `make all` in this directory on a machine which has the Intel iPSC cross-compiler. The `make` output will show the progress as each source code file is compiled and then linked to form two executable files, `pdeHost` and `pdeNode`.

9.3 Executing Example Code

As this is code for iPSC systems, it is necessary to place these files on a file system accessible to one of these hypercubes and then login to that SRM (hypercube front-end). After successfully allocating a subcube with the **getcube** command, the instrumented example is invoked by executing **pdeHost**. The program will prompt for a few parameters. As a starting point, one could enter 128 rows, 1.0 for the range, check for convergence at every iteration (1), and 200 for the maximum number of iterations. The algorithm will not actually converge with these parameters, but the program will produce some trace output data. A typical iPSC/860 executing the instrumented PDE example on 32 nodes with these parameters would finish in about ten minutes.

9.4 Processing Example Trace Output Data

When execution has completed, there will be several binary SDDF format trace output data files in the current directory. They can be merged to form a single, time ordered trace output file as follows:

```
SDDFmerge -o Jacobi.binary Jacobi*.bin
```

provided that the **SDDFmerge** program, provided as part of the Pablo source code distribution, has been compiled and installed on the current host, that the current host has the same byte and word ordering as the iPSC system[8] and that the individual trace data files are all accessible. See the document *Pablo Instrumentation Environment User's Guide* for further details.

Binary SDDF format files can be converted into ASCII SDDF format files with the **SDDFconverter** program. This program is interactive, just answer the prompts.

[8] DECstations have the same byte order as the Intel processors, but Sun SparcStations have a different byte ordering.

```
void printArray(arrayName, array, n)
     char *arrayName;
     int  *array;
     int  n;
{
#line 9 "PabloParseCode.c"
        int _printf_ReturnValue_;
#line 9 "/mnt/Pablo-guitar/Pablo.Oct92/Instrument/Parser/Examples/MatrixMultiply/MatrixMult.c"

  int i, j;


#line 12 "PabloParseCode.c"
(traceEvent(0, (char *)0, 0), _printf_ReturnValue_ =
#line 12 "/mnt/Pablo-guitar/Pablo.Oct92/Instrument/Parser/Examples/MatrixMultiply/MatrixMult.c"
printf("\nArray %s:\n\n", arrayName)
#line 12 "PabloParseCode.c"
, traceEvent(1, (char *)0, 0), _printf_ReturnValue_)
#line 12 "/mnt/Pablo-guitar/Pablo.Oct92/Instrument/Parser/Examples/MatrixMultiply/MatrixMult.c"
;


#line 14 "PabloParseCode.c"
{countEvent(20);
#line 14 "/mnt/Pablo-guitar/Pablo.Oct92/Instrument/Parser/Examples/MatrixMultiply/MatrixMult.c"
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      printf(" %d ", array[n * i + j]);
    }
    printf("\n");
  }
#line 19 "PabloParseCode.c"
countEvent(21);}
#line 19 "/mnt/Pablo-guitar/Pablo.Oct92/Instrument/Parser/Examples/MatrixMultiply/MatrixMult.c"

}
```

Figure 13: A Portion of the Instrumented Matrix Multiply Source Code

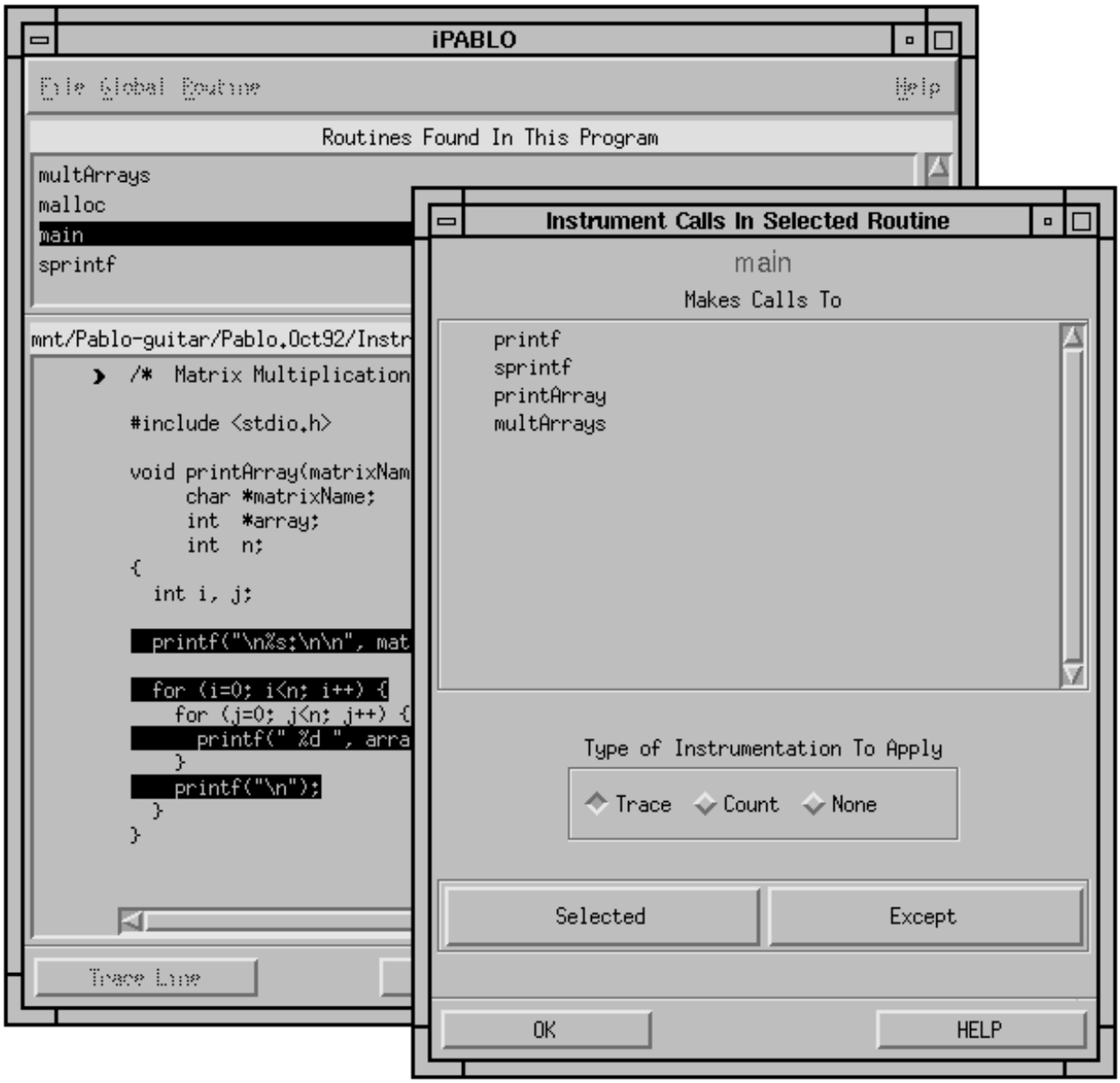

Figure 14: The *iPablo* Subset Dialog

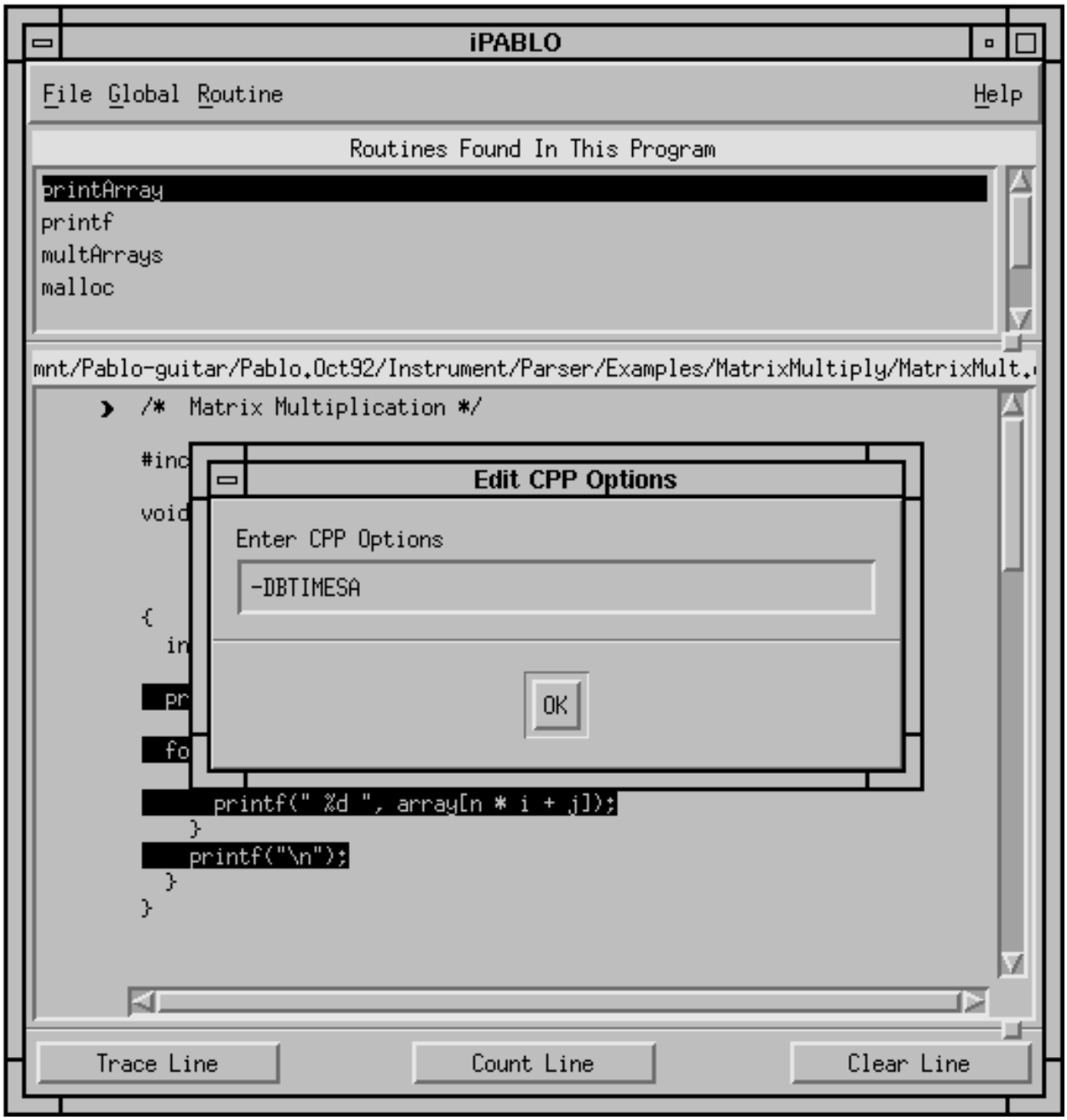

Figure 15: The *iPablo* **cpp** Editing Dialog