Course Title: Debugging Serial and Parallel Codes

1 Capabilities of Debugger Software

1.1 Capabilities of Debugger Software

Introduction

The traditional method for finding bugs in code is to place several write statements throughout the code and run it again to see what values the variables have at those locations. However, there is a much better way to debug code -- use debugging software. Debugging software makes it easier and faster to find problems in your code. You can even "look" inside your code while it is executing. For example, with three simple debugging commands you can have your program run to a certain line and then pause. You can then see what value **any** variable has at that point in the code. In addition, once you are running your code line-by-line within debugging software, there is much more information you can find about your code's execution than you could ever get through placement of write statements.

In the following sections of this lesson, we discuss many of the capabilities found in debugger software. Since it is not really possible, we do not cover all the capabilities found in every single debugger. Instead we discuss some important ones and leave it up to you to explore the others on your own.

Objectives

In this lesson, you will learn the basic capabilities of debugger software and how to use them to find bugs in your programs. Specifically, you will learn how to execute code, analyze code, alter code execution, and retrieve available code information from within debugger software.

1.2 Basic Debugger Overview

Debugging software is ubiquitous and you can find a debugger for almost any language, compiler, and/or software package that you use. Debuggers not only exist for all the major and minor programming languages (Fortran 77/90, C, C++, Java, Perl, Matlab®, Octave, Portland Group Compilers, KAI compilers, etc.) but are often built into the software.

Debugger interfaces come in two varieties: command-line and graphical user interface (GUI). You can choose what interface you want to use. You get all the capabilities of the debugging software regardless of how you choose to interact with it. However, GUI debuggers make debugging easier since you only have to click on buttons and icons to activate the debugging commands rather than remember specific commands.

Debuggers also work with parallel code. With parallel debuggers you can do the same kinds of analyses (while your program is running) as you can do with serial code. Except with parallel debuggers you can control the execution of the code on **each** processor. You can even perform different investigations on different processors. Totalview® is in many ways the premiere parallel debugger. It has an impressive set of windows for your code analysis with each one being able to switch from one processor to another. In addition, many of the popular serial debuggers have had commands added to them so that you can also analyze parallel programs.

To run your code in a debugger, you do need to compile your program in a certain way. However, it is very easy to do. Typically, you only need to use one compiler option to make your code "debuggable". And for almost all compilers this option is '-g', which is somewhat of a de facto standard.

It is also easy to use a debugger with your executable code. Typically, all you need to do is type the name of the debugger followed by the name of your executable. Upon doing this, you enter the debugging software with your program sitting at the beginning waiting to be run, paused, and analyzed.

1.3 Executing Code within Debugger Software

There are a few basic commands for executing code within debugger software. With these commands you can:

- list your source code with the lines numbered
- stop the program at a certain point
- execute the code line-by-line
- stop the program when the value of an expression changes
- jump to a certain line of code and start executing

Listing Source Code

The first step you want to take when debugging your code is to generate a line-numbered listing of your source code. By

looking at your program statements, you can decide how you wish to execute them. If the debugger has a GUI interface, your source code is displayed in a window with each line numbered.For a command-line debugger there is a '*list*' command that displays the first part of your program. At any time during your debugging session you can run 'list' and see which line you are on and a certain number of lines surrounding it (a typical number is 10). Typing 'list' again will display more of your code.

Stopping a Program at a Certain Point

Typically, you have some idea where a bug might be in your program. This could come from a run-time error message, strange looking output, or even a gut feeling. Therefore, the quickest way to locate it is to tell the debugger to run your program up to a certain line and then stop. To do this, you specify a *breakpoint* at the suspect line in your code. This is done with a debugger command typically called '*break*'. Think of a breakpoint as a stop sign. When your executing program hits a breakpoint, the execution stops. After you have set the breakpoint, you can begin the execution of your code (from line 1) with the '*run*' command. All of the lines preceding the breakpoint are executed inside the debugger just as if they had been executed normally. You can set as many breakpoints as you like anytime within the debugging session. Use the debugger command 'continue' when you are done analyzing your code at one breakpoint and want to move to the next.

Executing the Program Line-by-Line

When you want to carefully study your code's execution, it helps to run it line-by-line. You can do this using the '*next*' debugger command. When using this command, the current program line is executed and the debugger moves onto the next line and waits. By repeating the next command you execute your program line-by-line to see what is happening in it.

A command that is very similar to 'next' is the '*step*' command. This command enables you to analyze the subprograms (functions or subroutines) within your code. Say that while you are executing your code line-by-line using 'next', you notice that the line you are about to execute will transfer code execution into a subprogram. If you want to analyze the subprogram code, use the 'step' command. Repeated use of 'step' will execute the subprogram line-by-line and then return to the calling program when finished.

If you do not want to analyze a subprogram within your code, then you can use the 'next' command to move on to the next line in the main program. This is useful when you might have used a subprogram numerous times in several codes and you know the bug is not there. In a sense, you *skip over* the subprogram call. However, the subprogram is still executed. You are just not analyzing it within the debugger software.

Stopping when an Expression Value Changes

In addition to stopping your code at a specified line as is done with the 'break' command, you may also want to stop your code when the value of an expression changes. You do this by using the '*watch*' command. The 'watch' command takes an argument, which is an expression, and stops the program when the value of the expression changes. This procedure is called **setting a watchpoint**. Notice that when using '*watch*' you are not telling the debugger where to stop program execution (as with break) but rather the debugger tells you where in the program the expression changed value. In effect, the debugger runs your code line-by-line **watching** for the expression to change.

An example of when you would want to use the 'watch' command would be if you notice that somewhere in your code a variable *x* got the value 7. Furthermore, you know that if the code was working correctly, *x* would always be 6 or less. To find where this is happening in your code, you could set a watch point using the command:

```
watch x=7rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
```

1.4 Analyzing Code within Debugger Software

In the previous section, we described how to run your code within a debugger so that you can locate the point in it that you want to analyze further. There are several analysis techniques supported by debuggers. They are:

- printing the value of specified variables
- formatting print output
- printing expression values
- analyzing program memory

Print the Value of Specified Variables

Probably the most used and most powerful debugger command for analyzing code is the '*print*' command. By using the 'print' command you can see the value of a specific variable after your program has been partially executed. You can then compare the actual value with the expected value. Of course, if they are different then you have found your bug. As you would suspect, print expects a variable name as its argument. For example, if you want to know the value of the integer

variable *count*, the 'print' command would be:

```
print count
```

An extremely powerful feature of the 'print' debugging command is that it shows you the value of any variable or piece of data connected with the program. If a program identifier can hold a value, the value can be seen by using 'print'. A partial list of the variable types 'print' can show are:

- basic variables - real, integer, character, complex, logical
- array variables - entire array or subsections, multi-dimensional
- structure variables - containing members that are a mixture of types, Fortran 90 and C/C++
- pointer variables - actual memory address stored in the pointer is displayed
- objects - data members of an object-oriented instantiation of a class

Formatting Print Output

For every value obtained by the 'print' command, there is a default format based on the type of the variable. For example, it is common practice to show the contents of an integer variable as a decimal value. Every debugger allows you to change from the default format to see the variable's value in any format you want. To specify a certain format, you must give an argument to the 'print' command. The format symbol is often just a single character. How extensive the formatting can be varies depending on the debugger software. The table below shows some formats that are common between debuggers:

Integer Formats	Hexadecimal, Decimal, Octal, or Binary bases The character with the matching ASCII value
Real Formats	Fixed point Exponential (Scientific Notation) Number of decimal places can be specified
Character Formats	The character itself or The ASCII code for the character
Logical Formats	true, false 1,0
Address Formats	Absolute Memory Address (Hexadecimal) Relative Memory Address: offset from nearest preceding variable
Array Formats	As stored in memory As pictured by the programmer (if different)
Complex Format	(real part, imaginary part)

Printing the Values of Expressions

The argument for the 'print' command can be an expression that uses the operators of the language in which the program is written. First, the expression is evaluated and then its resulting value is displayed by print. For example, in your program you are looking at different points on an ellipse where the values of two variables - *peri_dist* and *apo_dist* - continuously change but their sum is always equal to the same constant value. In this case, it could be useful to have the debugger execute the command:

```
print peri_dist + apo_dist
```

at several places in the code to see if the distances are being calculated correctly or not.

The expression given to 'print' does not have to involve a numerical calculation as shown above; it can also be a relational or conditional expression. For instance, in your code there is the following while loop:

```
do while ( iter <= 150 .and. done==.false.)

     ...

end do
```

and you are not sure that your program is even executing it. The output from the following debugging command:

```
print iter <= 150 .and. done==.false.
```

will tell you if it is or not.

Analyzing Program Memory

Sometimes it is necessary to see what value is stored in a specific memory address. The debugging command for this is

usually called '*examine*', which takes a specific memory address as its argument. One area where examine is particularly useful is in looking at values of pointer variables. Say your program has a real variable called *speed* that is initialized to 55.0. In addition, your program has a (pointer to a real) variable called *rp* that is given the address of *speed*. The same memory location should now be accessible (and changeable) by either the variable *speed* or indirectly by the pointer *rp*. If you suspect that a pointer and its target variable are not connected in this way you can directly look at the memory address to confirm whether the value is correct or not.

The data used in a program can be found in several areas:

- Permanent Memory
- The Stack
- The Heap
- Microprocessor Chip Data Caches
- Microprocessor Chip Registers

Debugging software enables you to see what values are stored in any of these areas. For some data areas, 'examine' is used and for others 'print' is used. In addition, there are other commands that we have not discussed.

More important than the specific commands used, are the reasons you might want to examine these different data areas. Here is just a partial list of reasons to examine some of the data areas:

- *Stack:* The stack is a part of memory that comes into play when a subprogram is called. By seeing the stack contents, you can check the connection between actual and dummy arguments, local variable values, values returned, as well as parameters of the stack itself (size, memory location, etc.)
- *Heap:* The heap is a part of the stack used when memory is dynamically allocated for arrays. You analyze it if you think something is wrong when you bring an array into existence (or close it) during the course of your program
- *Data Caches:* One reason to study data caches is to learn their properties (size, line length) and the manner in which they are filled with data. A more important reason rarely has to do with debugging, but with code performance. For a microprocessor-based machine, it is essential that cache data is being used correctly. Ideally, all data critical to primary calculations should be in the cache and be used and reused to ensure that your program runs quickly.
- *Registers:* In several programming languages (C/C++) you can "ask" your program to store a variable in a register and not in permanent memory. A common variable designed for register storage is the iteration variable used in an unconditional loop (i.e., i,j,k) Iteration variables are, in general,not of interest except when the loop is being run so you do not really want to waste precious permanent memory storage on them. By using a debugger, you can check to see if the compiler actually put them in register storage as you asked.

1.5 Altering Code Execution within Debugger Software

A very powerful feature of debugger software is that once you step through and analyze your program to find your bug, you can alter your code to correct the bug and then execute the rest of it to see if there are any more bugs.

Changing a Variable Value

The typical command name for changing a variable value is 'set' or 'set var'. As an example, you find in your program that the real variable *grav* has a value of "9.8" but should equal "1.63". You can use the '*set*' *command*

```
set grav=1.63
```

to give *grav* the new (correct) value. If you have not set any stopping points beyond this point in the code, you can execute the rest of your code by entering the '*continue*' command. Hopefully this fix will cause your program to work correctly. However, do not forget that you still need to find out how *grav* got the wrong value to begin with.

Variations of the Set Command

The '*set*' command can do more than just change the value of variable during program execution. One of its uses is to specify a memory address to directly alter the contents of that address. You can also use the 'set' command to leave a subprogram prematurely and specify a new return value. This is helpful when you suspect that your error is within the subprogram and you want to alter what it calculates. To do this you enter the command:

```
return value
```

This causes the subprogram to immediately return to the next line of the main code, stop execution at that point, and pass back the return value to the main program.

A variant of 'set' that should be used with extreme caution, and only if you know the assembly language of the machine the code is running on, is often called 'set write'. This command allows you to use the debugger to actually modify the **machine code** within the executable file itself! As a rule, all debuggers open the executable in read-only mode, but 'set write' can override this. Often leading to serious problems when used incorrectly.

1.6 Retrieving Additional Information

In addition to the previously described information you can get from debugging software, there is considerably more that you can get. We now discuss two information retrieval commands 'info' and 'help'.

The 'info' Debugger Command

The '*info*' command can provide you with a variety of information. The following is a small listing of some types of information that is available:

- information on the user's debugging session
- how the debugger has been customized
- the nature, content and extension of files being used by the debugger
- the source language being used
- the debugger "environment"
- the contents of the symbol table
- a history of all print values displayed
- the UNIX environment of the shell the debugger is running in
- the history of the UNIX commands typed
- the characteristics of the hardware the debugger is running on

To use 'info' just give it a keyword argument. The keyword represents the information desired. For example, the following debugging command:

```
info breakpoint
```

list all the breakpoints you have set in your code and their locations. While the command:

```
info float
```

lists the contents and layout of the floating point functional unit in the microprocessor chip being used.

The HELP Debugger Command

Now that we have finished describing several debugging commands that are available to you, it is appropriate to let you know that descriptions of all of these commands and many more can be found on-line from within the debugging software itself. Just enter the command '*help*' to find out what they are and how to use them. Typically, the first help screen shows a list of broad topics. You can then use the command 'help topic_name' to get a screen of subtopics related to the topic you want to know more about. To get to the lowest level of detail, type 'help subtopic_name'. At this point you should get a complete description of the function and syntax of a specific debugging command.

2 Debugging Serial Code

2.1 Debugging Serial Code

In the following sections, you will learn several methods for debugging serial code.

2.2 Checking Variable Values

2.2.1 Checking Variable Values

Introduction

When you have an error in your code, examining variable values as your code executes is most likely the first step you take to find the problem. Using debugging software enables you to examine variable values while executing your code. This is a much more efficient and less error prone method of debugging a program than the traditional method of inserting print statements throughout your code.

Objectives

In this lesson, you will learn how to debug a C program using The GNU Project Debugger (GDB) to examine variable values. A sample code is provided that you can download and debug as you follow the lesson.

2.2.2 Sample Code

The C code below, variablePrinting.c, is a very simple program that:

1. generates a 10-element array,
2. passes this array to a function that squares each element in the array and stores these values in a second 10-element array, and then
3. computes the difference between the element values in the two arrays.

We use this sample code to show an example debugging session using GDB to examine variable values.

```
#include <stdio.h>
#include <stdlib.h>

int indx;

void initArray(int nelem_in_array, int *array);
void printArray(int nelem_in_array, int *array);
int squareArray(int nelem_in_array, int *array);

int main(void)
{
  const int nelem = 12;

  int *array1, *array2, *del;

/* Allocate memory for each array */
  array1 = (int *)malloc(nelem*sizeof(int));
  array2 = (int *)malloc(nelem*sizeof(int));
  del = (int *)malloc(nelem*sizeof(int));

/* Initialize array1 */
  initArray(nelem, array1);

/* Print the elements of array1 */
  printf("array1 = ");
  printArray(nelem, array1);

/* Copy array1 to array2 */
  array2 = array1;

/* Pass array2 to the function 'squareArray( )' */
  squareArray(nelem, array2);

/* Compute difference between elements of array2 and array1 */
  for (indx = 0; indx < nelem; indx++)
  {
    del[indx] = array2[indx] - array1[indx];
  }

/* Print the computed differences */
  printf("The difference in the elements of array2 and array1 are:  ");
  printArray(nelem, del);

  free(array1);
  free(array2);
  free(del);

  return 0;
}

void initArray(const int nelem_in_array, int *array)
{
  for (indx = 0; indx < nelem_in_array; indx++)
  {
```

```
    array[indx] = indx + 2;
  }
}

int squareArray(const int nelem_in_array, int *array)
{
  int indx;

  for (indx = 0; indx < nelem_in_array; indx++)
  {
    array[indx] *= array[indx];
  }
  return *array;
}

void printArray(const int nelem_in_array, int *array)
{
  printf("[  ");

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

2.2.3 Debugging the Sample Code

After compiling and running the sample code we obtain the following output:

```
$ gcc -o variablePrinting variablePrinting.c

$ ./variablePrinting

   array1 = [  2  3  4  5  6  7  8  9  10  11  ]
```

The difference in the elements of *array2* and *array1* are:

del = [0 0 0 0 0 0 0 0 0 0]

The result for the array *del* certainly does not look correct. If *array1* has elements *{ a1k }* and *array2* has the elements *{ a2k} = { (a1k)2 }* then *{ delk } = { 0 }* only if *{ (a1k)2} = { a1k}*, which tells us that *{ a1k } = { 1 }*, for *k = 0, 1, . . . , (nelem − 1)*. But we can see from our output that not a single one of the elements of *array1* has the value 1. Therefore, we must have a coding error in our program.

To identify this error we recompile our code using the debugging option '-g' and analyze it using GDB. In this example we illustrate how to debug a code by using a debugger to print out the values of selected variables during the execution of a program.

One possibility that could account for the values of all elements of *del* being zero is a coding error in our function squareArray(). If, for some reason, this function fails to square the elements of *array1* and, instead, is simply returning an array, *array2*, whose elements are identical to those of *array1* then

del[k] = array2[k] − array1[k] = array1[k] − array1[k] = 0

for all *k*.

Therefore, we might first want to examine the values of the elements of the array *array2* to see if *{ a2k } = { a1k }*. We can do this in either of two ways. The first way is to set a breakpoint on the line of our code where we call the function squareArray() then print out the value of *array2[indx]* as we step through all *nelem* iterations of the for() loop within the body of this function.

```
int squareArray(const int nelem_in_array, int *array)
     {
       int indx;

       for (indx = 0; indx < nelem_in_array; indx++)
       {
```

```
        array[indx] *= array[indx];
      }
      return *array;
    }
```

The second way is to set a breakpoint immediately after the call to squareArray() and print out the elements of *array2* returned by the call to squareArray(). This example will illustrate both ways of examining these values *{ a2k }*.

After recompiling our code we run it using GDB.

```
$ gcc -g -o variablePrinting variablePrinting.c

$ ./variablePrinting
(gdb)
```

First, we issue the 'list' command (or 'l') to display our source code in order to identify the line of code on which we want to set our initial breakpoint. Note that following the 'list' command we have included the function name *main* so that the listing of code will begin immediately prior to the first line of this function.

```
(gdb) l main
6       void initArray(int nelem_in_array, int *array);
7       void printArray(int nelem_in_array, int *array);
8       int squareArray(int nelem_in_array, int *array);
9
10      int main(void)
11      {
12        const int nelem = 12;
13
14        int *array1, *array2, *del;
15
(gdb)
16      /* Allocate memory for each array */
17        array1 = (int *)malloc(nelem*sizeof(int));
18        array2 = (int *)malloc(nelem*sizeof(int));
19        del = (int *)malloc(nelem*sizeof(int));
20
21      /* Initialize array1 */
22        initArray(nelem, array1);
23
24     /* Print the elements of array1 */
25       printf("\n");
(gdb)
26       printf("   array1 = ");
27       printArray(nelem, array1);
28
29     /* Copy array1 to array2 */
30       array2 = array1;
(gdb)
31
32    /* Pass array2 to the function 'squareArray( )' */
33       squareArray(nelem, array2);
34
35     /* Compute difference between elements of array2 and array1 */
36       for (indx = 0; indx < nelem; indx++)
37       {
38         del[indx] = array2[indx] - array1[indx];
39       }
40
(gdb)
```

If we want to step into and through our function squareArray(), we need to set our initial breakpoint on line 33 and then rerun our code.

```
(gdb) b 33
Breakpoint 1 at 0x8048469: file variablePrinting.c, line 33.
(gdb) run
Starting program: variablePrinting

   array1 = [  2  3  4  5  6  7  8  9  10  11  12  13  ]
```

```
Breakpoint 1, main () at variablePrinting.c:33
33        squareArray(nelem, array2);
```

We can then step into this function by issuing the 'step' (or 's') command.

```
(gdb) s
squareArray (nelem_in_array=12, array=0x80498d8) at variablePrinting.c:65
65        for (indx = 0; indx < nelem_in_array; indx++)
(gdb) s
67          array[indx] *= array[indx];
```

At this point we begin printing out the values of the elements of *array* on both the left- and right-hand side of the expression on line 67. We also track our progress thru the for() loop by printing out the value of the variable *indx* during each iteration of the loop. We can print the value of each of these variables by issuing the 'print' (or 'p') command followed by the name of the variable whose value you want to print; e.g.,

```
(gdb) p indx
(gdb) p array[indx]
```

The print command

```
p v
```

prints out the value of the variable *v* in the same format as a printf() statement in C, that is

```
printf("%d\n", v)
```

if *v* is an integer variable or

```
printf("%f\n", v))
```

if *v* is a double.

In this example, we are only printing out the values of two variables. However, as the number of variables becomes more than just a few, this method of printing a sequence of variable values will begin to require a lot of typing. Fortunately, GDB provides two alternative methods for printing out a series of variable values without having to type multiple print commands each time you want to examine these values. One method involves creating a print macro issuing the GDB 'define' command. The second method involves issuing the GDB 'display' (or 'disp') command. The use of both of these methods to print out the values of *indx* and *array[indx]* during execution of the for() loop on line 33 is illustrated below.

First, we define a print macro named *var*.

```
(gdb) define var
```

After issuing this command GDB will then respond with the statement

```
Type commands for definition of "var".
End with a line saying just "end".
>
```

We then simply type a printf statement identifying the variables we want to include and the format in which we want each of these variables printed,

```
Type commands for definition of "var".
End with a line saying just "end".
>printf "indx = %d\n  array[indx] = %d\n", indx, array[indx]
>end
(gdb)
```

Note that we notify GDB that our definition of *var* is complete by typing 'end'.

Now, each time we want to print out the values of *indx* and *array[index]* we simply type the name of our print macro, *var*.

```
(gdb) var
indx = 0
 array[indx] = 2
(gdb) s
65        for (indx = 0; indx < nelem_in_array; indx++)
```

```
(gdb) var
indx = 0
 array[indx] = 4
(gdb) s
65          array[indx] *= array[indx];
(gdb) var
indx = 1
 array[indx] = 3
(gdb) s
65        for (indx = 0; indx < nelem_in_array; indx++)
(gdb) var
indx = 1
 array[indx] = 9
(gdb) s
65          array[indx] *= array[indx];
(gdb) var
indx = 2
 array[indx] = 4
(gdb) s
65        for (indx = 0; indx < nelem_in_array; indx++)
(gdb) var
indx = 2
 array[indx] = 16
(gdb) s
67          array[indx] *= array[indx];
(gdb) var
indx = 3
 array[indx] = 5
(gdb) s
65        for (indx = 0; indx < nelem_in_array; indx++)
(gdb) var
indx = 3
 array[indx] = 25
(gdb) s
67          array[indx] *= array[indx];
(gdb) var
indx = 4
 array[indx] = 6
(gdb) s
65        for (indx = 0; indx < nelem_in_array; indx++)
(gdb) var
indx = 4
 array[indx] = 36
(gdb) s
67          array[indx] *= array[indx];
```

Although it is already obvious that our function squareArray() appears to be behaving properly; viz., the function is correctly computing the squares of the elements of array, we print out the values of our variables during the remaining iterations of this loop just to illustrate the use of the 'disp' command. For each variable we want to print out we type the command 'disp variable-name',

```
(gdb) disp indx
1: indx = 5
(gdb) disp array[indx]
2: array[indx] = 7
```

Once we have notified GDB of the variables we want displayed, GDB will automatically print out the value of these variables each time the program pauses; e.g., each time we issue the step (s) command to step through our for() loop as illustrated below.

```
(gdb) s
65        for (indx = 0; indx < nelem_in_array; indx++)
2: array[indx] = 49
1: indx = 5
(gdb) s
67          array[indx] *= array[indx];
2: array[indx] = 8
1: indx = 6
(gdb) s
```

```
65        for (indx = 0; indx < nelem_in_array; indx++)
2: array[indx] = 64
1: indx = 6
(gdb) s
67          array[indx] *= array[indx];
2: array[indx] = 9
1: indx = 7
(gdb) s
65        for (indx = 0; indx < nelem_in_array; indx++)
2: array[indx] = 81
1: indx = 7
(gdb) s
67          array[indx] *= array[indx];
2: array[indx] = 10
1: indx = 8
(gdb) s
65      for (indx = 0; indx < nelem_in_array; indx++)
2: array[indx] = 100
1: indx = 8
(gdb) s
67         array[indx] *= array[indx];
2: array[indx] = 11
1: indx = 9
(gdb) s
65     for (indx = 0; indx < nelem_in_array; indx++)
2: array[indx] = 121
1: indx = 9
(gdb) s
69      return *array;
2: array[indx] = 0
1: indx = 10
(gdb)
```

Clearly, using the 'disp' command for printing multiple variables requires even less typing than using a 'print' macro. However, since each of the variables are automatically printed every time the program pauses, it is most useful only when examining variables whose values are changing frequently; e.g., during the execution of a for() loop. Otherwise, we would probably not want to see a set of constant values being printed out over and over throughout an entire debugging session. Fortunately, there is an easy way to tell GDB to stop displaying a given variable. We simply invoke the command 'undisp' followed by the number *n* that is printed to the left of the variable name,

n: variable-name = value

For example, we can cancel the printing of the variables *indx* and *array[indx]* by issuing the commands

```
(gdb) undisp 1
(gdb) undisp 2
(gdb)
```

Examining the values of *array[indx]* we see that each element of *array2* returned by squareArray() is equal to the square of the corresponding element in *array1*. Consequently, we know that our coding error is not located anywhere within the function squareArray(). Therefore, we need to continue debugging our program until we locate the coding error in our program. We leave that for you to do in the end-of-lesson exercise.

2.2.4 Debugging Exercise

In the example program illustrated in this lesson, variablePrinting.c, we found that our coding error was not located in the function squareArray() as we first expected. Beginning where we left off in the example, complete the debugging of this code and identify the coding error in the program.

2.2.5 Exercise Solution

In the debugging example we had completed examining the values of *array2* computed by our function squareArray() but had found no errors in this function. In the solution illustrated below, we complete our debugging of variablePrinting.c simply by examining variable values within the remainder of this program.

If we look back at our lesson example, we can see from the printout of our code that, following the call to squareArray(), our program computes the difference between the element values in the two arrays, *array2* and *array1*,

del[indx] = array2[indx] - array1[indx] , for indx = 0, 1, . . . , (nelem 1)

beginning at the for() loop on line 41.

To debug this next section of code, we step through this for() loop and examine the values of each variable on both the left- and right-hand sides of our expression for *del[indx]* to see if these values look correct for each value of *indx* in the this loop. We print out the values of *indx*, *array2*, *array1*, and *del*, using the GDB 'display' command.

```
(gdb) n
75      }
(gdb) n
main () at variablePrinting.c:41
41        for (indx = 0; indx < nelem; indx++)
(gdb) s
43          del[indx] = array2[indx] - array1[indx];
(gdb) disp indx
3: indx = 0
(gdb) disp array2[indx]
4: array2[indx] = 4
(gdb) disp array1[indx]
5: array1[indx] = 4
(gdb) disp del[indx]
6: del[indx] = 0
(gdb) s
41        for (indx = 0; indx < nelem; indx++)
6: del[indx] = 0
5: array1[indx] = 4
4: array2[indx] = 4
3: indx = 0
(gdb) s
43          del[indx] = array2[indx] - array1[indx];
6: del[indx] = 0
5: array1[indx] = 9
4: array2[indx] = 9
3: indx = 1
(gdb) s
41        for (indx = 0; indx < nelem; indx++)
6: del[indx] = 0
5: array1[indx] = 9
4: array2[indx] = 9
3: indx = 1
(gdb) s
43          del[indx] = array2[indx] - array1[indx];
6: del[indx] = 0
5: array1[indx] = 16
4: array2[indx] = 16
3: indx = 2
(gdb) s
41        for (indx = 0; indx < nelem; indx++)
6: del[indx] = 0
5: array1[indx] = 16
4: array2[indx] = 16
3: indx = 2
(gdb) s
43          del[indx] = array2[indx] - array1[indx];
6: del[indx] = 0
5: array1[indx] = 25
4: array2[indx] = 25
3: indx = 3
(gdb) s
41        for (indx = 0; indx < nelem; indx++)
6: del[indx] = 0
5: array1[indx] = 25
4: array2[indx] = 25
3: indx = 3
(gdb) s
43          del[indx] = array2[indx] - array1[indx];
6: del[indx] = 0
5: array1[indx] = 36
```

```
4: array2[indx] = 36
3: indx = 4
(gdb) s
41        for (indx = 0; indx < nelem; indx++)
6: del[indx] = 0
5: array1[indx] = 36
4: array2[indx] = 36
3: indx = 4
(gdb) s
43          del[indx] = array2[indx] - array1[indx];
6: del[indx] = 0
5: array1[indx] = 49
4: array2[indx] = 49
3: indx = 5
(gdb) s
41        for (indx = 0; indx < nelem; indx++)
6: del[indx] = 0
5: array1[indx] = 49
4: array2[indx] = 49
3: indx = 5
(gdb) s
43          del[indx] = array2[indx] - array1[indx];
6: del[indx] = 0
5: array1[indx] = 64
4: array2[indx] = 64
3: indx = 6
(gdb) s
41        for (indx = 0; indx < nelem; indx++)
6: del[indx] = 0
5: array1[indx] = 64
4: array2[indx] = 64
3: indx = 6
(gdb) s
43          del[indx] = array2[indx] - array1[indx];
6: del[indx] = 0
5: array1[indx] = 81
4: array2[indx] = 81
3: indx = 7
(gdb) s
41        for (indx = 0; indx < nelem; indx++)
6: del[indx] = 0
5: array1[indx] = 81
4: array2[indx] = 81
3: indx = 7
(gdb) s
43          del[indx] = array2[indx] - array1[indx];
6: del[indx] = 0
5: array1[indx] = 100
4: array2[indx] = 100
3: indx = 8
(gdb) s
41        for (indx = 0; indx < nelem; indx++)
6: del[indx] = 0
5: array1[indx] = 100
4: array2[indx] = 100
3: indx = 8
(gdb) s
43          del[indx] = array2[indx] - array1[indx];
6: del[indx] = 0
5: array1[indx] = 121
4: array2[indx] = 121
3: indx = 9
(gdb) s
41        for (indx = 0; indx < nelem; indx++)
6: del[indx] = 0
5: array1[indx] = 121
4: array2[indx] = 121
3: indx = 9
(gdb) s
```

```
47        printf("The difference in the elements of array2 and array1 are:\n\n");
6: del[indx] = 0
5: array1[indx] = 0
4: array2[indx] = 0
3: indx = 10
(gdb) undisp indx
(gdb) undisp array1
(gdb) undisp array2
(gdb) undisp del
```

Clearly, the values for *array1* do not look correct at all. When we debugged our function in the lesson example, we found that the values of *array2* passed to the function squareArray() were correct and that the values of *array2* returned from this function were also correct, viz., { a2k } = { (a1k)2 }. That is, we found that the value of each element in *array2* was to equal the square of the values of the corresponding element in *array1*. But as we see from the above printout of our variables, the elements of *array1* and *array2* are equal, { a2k } = { a1k }. Somehow the values of the elements in *array1* have been squared in addition to the elements in array2.

So how could this have happened? The array, *array1*, was never passed to the function squareArray(); only *array2* was passed in line 38 of our code. If we think about it a bit, this sounds very much like a pointer error. If for some reason *array2* and *array1* are pointing to the same location in memory and the value of one of the elements of *array2* is modified, then the value of the corresponding element in *array1* will equal the modified value in *array2*. To confirm our suspicion, we compare the memory address of both *array1* and *array2*. If we look back at the example in this lesson, when we first stepped into our function squareArray( ) from line 38, GDB gave us the address of when it was passed to this function,

```
(gdb) s
squareArray (nelem_in_array=12, array=0x80498d8) at variablePrinting.c:70
70        for (indx = 0; indx < nelem_in_array; indx++)
(gdb) s
72          array[indx] *= array[indx];
```

If we print out the address pointed to by *array1* and compare it with the address to which the pointer *array2* was pointing when it was passed to squareArray(), we find that the two addresses are identical.

```
(gdb) disp array1
1: array1 = (int *) 0x80498d8
```

If we want, we can confirm this by printing out the address to which *array2* is pointing in the expression for *del[indx]* on line 43 of our code,

```
(gdb) disp array2
2: array2 = (int *) 0x80498d8
(gdb)undisp 1
(gdb)undisp 2
```

As we can see, this is the same address as that passed to squareArray() on line 38.

The question is: How can these two arrays have the same memory address when the memory for these arrays were allocated separately on lines 17 and 18 of our code?

```
16      /* Allocate memory for each array */
17        array1 = (int *)malloc(nelem*sizeof(int));
18        array2 = (int *)malloc(nelem*sizeof(int));
19        del = (int *)malloc(nelem*sizeof(int));
20
```

Because the memory for *array1* and *array2* were allocated separately, they should have different addresses. We can confirm this by setting a new breakpoint at line 20 and printing out the values of *array1* and *array2* immediately after this memory allocation is made,

```
(gdb)clear 38
Deleted breakpoint 1
(gdb) b 20
Breakpoint 2 at 0x80483e9: file variablePrinting.c, line 20.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: variablePrinting
```

```
Breakpoint 2, main () at variablePrinting.c:22
22        initArray(nelem, array1);
(gdb) p array1
 = (int *) 0x80498d8
(gdb) p array2
 = (int *) 0x8049910
(gdb)
```

This confirms that, initially, the addresses of the two arrays are not the same. Since the address of *array1* is exactly the same here as in the computation of *del[indx]* on line 43, but the address of *array2* does change then the error in our code must be modifying the address of *array2*. To find out exactly where in our code the address of *array2* changes, we move down the code line-by-line, starting at line 20, examining the value of array2.

```
(gdb) n
24        for (indx = 0; indx < nelem; indx++)
(gdb) disp array2
1: array2 = (int *) 0x8049910
(gdb) s
26          array1[indx] = indx + 2;
1: array2 = (int *) 0x8049910
(gdb) s
24        for (indx = 0; indx < nelem; indx++)
1: array2 = (int *) 0x8049910
(gdb) s
26          array1[indx] = indx + 2;
1: array2 = (int *) 0x8049910
(gdb) s
24        for (indx = 0; indx < nelem; indx++)
1: array2 = (int *) 0x8049910

.   .   .
.   .   .

30        printf("\n");
1: array2 = (int *) 0x8049910
(gdb) s

31        printf("   array1 = ");
1: array2 = (int *) 0x8049910
(gdb) s
32        printArray(nelem, array1);
1: array2 = (int *) 0x8049910
(gdb) s
printArray (nelem_in_array=12, array=0x80498d8) at variablePrinting.c:79
79        printf("[  ");

.   .   .
.   .   .

85        printf("]\n\n");
(gdb) s
   array1 = [  2  3  4  5  6  7  8  9  10  11  12  13  ]

86      }
(gdb) s
main () at variablePrinting.c:35
35        array2 = array1;
1: array2 = (int *) 0x8049910
(gdb) s
38        squareArray(nelem, array2);
1: array2 = (int *) 0x80498d8
(gdb) l
```

What we find is that the address of *array2* changes after line 35, just before *array2* is passed to the function squareArray( ). If we look carefully at our code, the reason is rather obvious. When we wrote the code to copy *array1* to *array2* on line 35, our intent was to equate the corresponding elements in the two arrays, *array1* and *array2*. The code on line 35 does exactly this, but it does it by pointing the first element in *array2* to the address of the first element in *array1*. That is, writing

```
array2  =  array1
```

is equivalent to writing

```
&array2[0]  =  &array1[0]
```

And, since the elements in each array are contiguous, then this is also equivalent to writing

```
&array2[ k ]  =  &array1[ k ]  , for  k  = 0, 1, . . . ,  (nelem − 1) .
```

So, we now have two pointers, *array1* and *array2*, pointing to the same data (the data at 0x80498d8). Therefore, when the value of the data element at 0x80498d8 is modified by passing to the function squareArray(), both arrays will now point to the same modified data at 0x80498d8. Consequently, when the elements of *array2* are squared through the call

```
squareArray(nelem, array2)
```

on line 38, the elements of *array1* will also be squared and

```
del[ k ]  =  array2[ k ]  − array1[ k ]  =  0 , for  k  = 0, 1, . . . ,  (nelem − 1)
```

What we actually meant to do on line 35 is not set the addresses of the two arrays equal,

```
array2  =  array1  ,
```

but, instead, to set the element values of the two arrays equal,

```
array2[ k ]  =  array1[ k ]  ,  for  k  = 0, 1, . . . ,  (nelem − 1)
```

Therefore, to correct our code, we need to replace the statement on line 35 with the following code,

```
for (indx = 0; index < nelem; indx++)
{
  array2[ k ]  =  array1[ k ]
}
```

When we make this correction and then recompile and rerun our code, we obtain the following correct output.

```
$ gcc -g -o variablePrinting variablePrinting.c

$ ./variablePrinting

   array1 = [  2  3  4  5  6  7  8  9  10  11  12  13  ]
```

The difference in the elements of *array2* and *array1* are:

```
del = [  2  6  12  20  30  42  56  72  90  110  132  156  ]
```

2.3 Array Indexing Errors

2.3.1 Array Indexing Errors

Introduction

When you declare the size of an array you also implicitly define the range of allowed indices that are associated with the array's elements. Improper referencing of array elements causes array indexing errors. For example, if you reference an element outside your defined range of indices, you get an "out of bounds" run-time error. This is a common programming mistake. The consequences of array indexing errors vary widely from your program crashing before it completes to it running to completion, but producing incorrect results. The latter is the most harmful since you may not know that the results are incorrect.

One of the reasons array indexing errors are so common is that the default range of indices differs among the major programming languages. For example, in Fortran 77 the default index range is $i=1,...,N$ (where N is the array size); in C/C++ the default range is $i=0,...,N-1$; and in Fortran 90 the allowed index range can be *any* sequence of integers with *any* spacing. So if you routinely use several programming languages, it is easy to confuse the indexing rules.

Another common cause of array indexing errors is that the importance of defining array indices is not always recognized. Other aspects of code writing such as the algorithms and logic used are given more attention whereas array indexing is considered a rote task that is coded easily.

Objectives

In this lesson, you will learn what array indexing errors are, what causes them, and how to debug code with an array indexing error using The GNU Project Debugger (GDB). A sample program with an array indexing error is provided that you may download and debug on your own while following along with the procedure described here.

2.3.2 Sample Code with Array Indexing Error

The following program contains an array indexing error:

```
#include <stdio.h>
#define N 10

void main(int argc, char *argv[]) {
   int tock[N];
   int i;
   int oddsum,evensum;

   for(i=1;i<(N-1);++i) {
      if(i≤4) {
        tock[i]=[i*i]%3;
      }else{
        tock[i]=[i*i]%5;
      }
   }

   oddsum=0;
   evensum=0;
   for(i=0;i<(N-1);++i) {
     if(i%2==0) {
       evensum=evensum+tock[i];
     }else{
       oddsum=oddsum+tock[i];
     }
   }

   printf("oddsum=%d\n",oddsum);
   printf("evensum=%d\n",evensum);

}
```

This program is supposed to fill an integer array *N* elements according to an algorithm based on the mod operator (% in C) and should add the values of all of the array elements with even indices and output the result. It should also do the same with odd-indexed elements.

2.3.3 Debugging an Array Indexing Error Using GDB

After compiling and executing the code, we get the following result:

```
debug$ gcc -o array array.c
debug$ ./array
oddsum=5
evensum=224683
debug$
```

As you can see, the code does not generate compiler or runtime errors. However, something is wrong. By looking at the body of the first loop it looks like the array elements should be relatively small integers. So the value of *oddsum* is probably alright. But why is the value of *evensum* so large? The value of *evensum* is computed in our summation loop, so we suspect that the error occurs there. Therefore, we will concentrate our debugging effort on that section of code.

Now we will show you how to find and fix the error using GDB. As we said, we suspect that there is an error in the program's summation loop. There are five basic steps we will take to identify the error along with a final step of exiting the debugger:

1. Recompile using the '-g' option
2. Run the debugger
3. Identify the breakpoint

4. Execute the code up to the breakpoint
5. Step through the summation loop
6. Exit the debugger

Step 1: Recompile using the -g option

The first step we take is to generate debugging information while compiling our code. We enter the following command to recompile the code in array.c with the debugger option '-g' turned on:

```
debug$ gcc -g -o array array.c
```

Doing this generates information that is stored in a file describing the data type of each variable or function and the correspondence between line numbers in the source code and addresses in the executable code.

Step 2: Run the debugger

After recompiling the code using the '-g' option, we run the debugger by typing the 'gdb' command followed by the name of the executable program:

```
debug$ gdb array
```

The following output is displayed followed by the (gdb) prompt:

```
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
gdb is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for gdb.  Type "show warranty" for details.
This gdb was configured as "sparc-sun-solaris2.7"...
(gdb)
```

When the (gdb) prompt appears it indicates that the debugger is ready to analyze the code and is waiting for you to enter debugging commands.

Step 3: Identify breakpoint

Typically, you should first scan through the source code using the 'list' command until you see the line in your code that you suspect is causing the problem. By default, the 'list' command prints ten lines at a time, so we enter the command until we see the summation loop that we suspect is causing the problem:

```
(gdb) list
1       #include <stdio.h>
2       #define N 10
3
4       void main(int argc, char *argv[]) {
5          int tock[N];
6          int i;
7          int oddsum,evensum;
8
9          for(i=1;i<(N-1);++i) {
10            if(i<=4) {

(gdb) list
11              tock[i]=(i*i)%3;
12            }else{
13              tock[i]=(i*i)%5;
14            }
15         }
16
17         oddsum=0;
18         evensum=0;
19         for(i=0;i<(N-1);++i) {
20           if(i%2==0) {
(gdb)
```

From this listing we see that the summation loop starts at line 19 so that is the value we will use for our breakpoint.

Step 4: Execute the code up to the breakpoint

Now we run the code line-by-line up to the beginning of the summation loop and then stop. To do this, we set a breakpoint at line 19 using the 'break' command:

```
(gdb) break 19
Breakpoint 1 at 0x10778: file array.c, line 19.
```

Then we type the 'run' command to execute the program to the point that we just entered:

```
(gdb) run
Starting program: /nfs/homea/dje/WebCT/debug/array

Breakpoint 1, main (argc=1, argv=0xffbefb54) at array.c:19
19         for(i=0;i<(N-1);++i) {
```

```
(gdb) list
14            }
15         }
16
17         oddsum=0;
18         evensum=0;
19         for(i=0;i<(N-1);++i) {
20           if(i%2==0) {
21             evensum=evensum+tock[i];
22           }else{
23             oddsum=oddsum+tock[i];
```

Notice that the program executed as it normally would and stopped at line 19.

Step 5: Step through the summation loop

Before analyzing the summation loop, it is good idea to check out some of the array elements to see if they have the correct values. To do this, use the 'print' debugger command, which shows the value of a variable.

```
(gdb) print tock[2]
 = 1
```

```
(gdb) print tock[7]
(gdb) print tock[7] $2 = 4 (gdb)  = 4
(gdb)
```

From the initialization loop, you know that the value of *tock[2]* should be 4 mod 3, which equals 1 and the value of *tock[7]* should be 49 mod 5, which is 4. To this point, the results are as expected. We also want to check the value of *evensum* to make sure that it did get initialized to zero:

```
(gdb) print evensum $2 = 0
```

Using the 'step' command we now step through the summation loop and check the value of the variable *evensum*. The 'step' command enables you to "step" through your program executing one line at a time. After each execution, the next line to be run is shown. We enter the 'step' command three times to execute the first iteration of the loop (i=0):

```
(gdb) step 20 if(i%2==0) { (gdb) step 21 evensum=evensum+tock[i]; (gdb) step 19 for(i=0;i<(N-1);++i) {
```

We then check the value of evensum again:

```
(gdb) print evensum $3 = 224676
```

It is much higher than expected so we now suspect that the value of *tock[0]* is not correct. Using the 'print' command to check the value confirms that it is indeed too high:

```
(gdb) print tock[0] $4 = 224676
```

We have found the problem: the array element *tock[0]* was not initialized correctly. In fact, by looking at the initialization loop you can see that it was not initialized at all. The array index was set to start at 1 (like in Fortran) and not 0 (like in C). Therefore, the array element *tock[0]* contained a random, and incorrect, value rather than zero as was intended.

Step 6: Exit the debugger

Now that we know the problem, we exit the debugger using the 'continue' command. The 'continue' command tells the debugger to continue executing code until another breakpoint is encountered or the code finishes. Since we have not issued another breakpoint, our code finishes executing.

```
(gdb) continue Continuing. oddsum=5 evensum=224683 Program exited with code 01.
```

To return to the Unix shell, type the quit command:

```
(gdb) quit debug$
```

Debugged Code

The final debugged version of the program is:

```
#include <stdio.h> #define N 10 void main(int argc, char *argv[]) { int tock[N]; int i; int
oddsum,evensum; for(i=0;i<(N-1);++i) { if(i<=4) { tock[i]=(i*i)%3; }else{ tock[i]=(i*i)%5; } } oddsum=0;
evensum=0; for(i=0;i<(N-1);++i) { if(i%2==0) { evensum=evensum+tock[i]; }else{ oddsum=oddsum+tock[i]; }
} printf("oddsum=%d\n",oddsum); printf("evensum=%d\n",evensum); }
```

2.3.4 Debugging Exercise

This following program has an indexing error in it. The code reads in data from a file called "input.txt" and puts it in the array *tock*. Then the sum of all the elements is calculated and printed out.

```
#include <stdio.h>
#define N 50

int main(int argc, char *argv[]) {
   int tock[N];
   int i;
   int temp=3;
   long int sum;

   i=-1;
   while(temp > 0) {
      if (i>=0) {
        tock[i]=temp;
      }
      scanf("%d",&temp);
      ++i;
   }

   sum=0;
   for(i=0;i<N;++i) {
     sum+=tock[i];
   }

   printf("sum is %ld\n",sum);

   return(0);
}
```

The data is input using the while loop shown. All the data are positive random numbers between 0 and 99. There is a negative number at the end of the input file that is used as a signal to tell the program to stop reading. Here is a partial list of the file "input.txt":

```
40
84
35
45
32
89
 2
58
16
38
```

```
69
 6
...
```

The OSC Cray X1 and its software was used to test this program. The code compiles, run, and even produces the following output:

```
armstrong$ cc prob.c
armstrong$ ./a.out < input.txt
sum is 1090515646
armstrong$
```

As you can see from the output, the value of the sum is way too large given the numbers input. We leave it up to you to figure out why.

2.3.5 Exercise Solution

The input file only has 25 values in it, but the array *tock* can hold 50 integers. So after the array values are input in the while loop, what values are in the last half of *tock*? In C, arrays are not intialized to anything at declaration. Therefore, if only the first half of *tock* is filled with values, the last half contains whatever happened to be in those memory locations, which is junk. So when the summation is done over all 50 elements, 25 unwanted integers are addded to the sum (and some of these junk integers can be quite large). That explains why the incorrect sum is so large.

There are many ways to fix this problem. We use a rather simple approach in the solution program below. In this program, all 50 elements of *tock* are directly intialized to zero. Therefore, only zeros from the last 25 elements are added to the sum, which have no effect.

```
#include <stdio.h>
#define N 50

int main(int argc, char *argv[]) {
   int tock[N]={0};  /* HERE IS WHERE THE ONLY CHANGE IS MADE */
   int i;
   int temp=3;
   long int sum;

   i=-1;
   while(temp > 0) {
      if (i>=0) {
        tock[i]=temp;
      }
      scanf("%d",&temp);
      ++i;
   }

   sum=0;
   for(i=0;i<N;++i) {
     sum+=tock[i];
   }

   printf("sum is %ld\n",sum);

   return(0);
}
```

The following correct output is generated when this code is executed:

```
armstrong$ cc sol2.c
armstrong$ ./a.out < input.txt
sum is 1054
armstrong$
```

2.4 Fortran I/O Errors

2.4.1 Fortran I/O Errors

Introduction

Fortran I/O code is prone to a wide range of bugs. Probably the most common ones are associated with formatted READ or WRITE statements. This is because it is very easy to forget to account for spaces in externally generated text files or to make a mistake in assigning the precision of a real. Less common, but easy to fix, are bugs associated with mismatched unit assignments or trying to read more data than is in a file. Similar errors also arise in binary I/O. Compiling on different Fortran compilers with non-standard practices is another common cause of errors. For example, some compilers consider encountering the end of a file as an error while others do not.

Finding and fixing bugs in Fortran I/O code can be very simple, but it can also be very difficult. With improvements in runtime checking many simple errors can be detected automatically and appropriate error messages are generated that often point to the exact problem. However, if the errors are not automatically detected it is often not possible to use a debugger in the same way you would to find other common bugs such as incorrectly assigned variables. This is because the actual process of reading or writing data to a file is handled by system calls for which there is no source code to follow.

In this lesson, we use the TotalView® debugger to study one of the bugs in our sample program, but similar steps can also be used with numerous other debuggers. For ease of demonstration, we concentrate on ASCII files but similar procedures work equally well for binary files.

Objectives

In this lesson, you will learn about common errors in Fortran I/O, the types of messages they generate on different systems, and how to fix them. You will also learn how to find more difficult errors using debugger software. A sample program with Fortran I/O errors is provided that you may download and debug on your own while following along with the lesson.

2.4.2 Sample Code with File Errors

For this lesson we use an example program written in Fortran 90 that

1. reads in a set of data stored in ASCII format
2. does some simple data manipulation
3. writes the results as a binary file
4. reads the binary file back in before finally writing some of the fields to a formatted text file

As you will see, this program is very poorly written and full of errors.

```
PROGRAM sections

TYPE idrecord
      INTEGER id
      CHARACTER(len=16) :: name
      CHARACTER(len=3)  :: country
      INTEGER rating
      CHARACTER(len=2)  :: title
      INTEGER experience
END TYPE idrecord

PARAMETER (Nplayer=10)

TYPE(idrecord) test(Nplayer)


OPEN(UNIT=10,FILE="sectionsCT.txt")

OPEN(UNIT=11,FILE="out.data",FORM='unformatted')

DO I=1,Nplayer
READ(10,FMT='(I6,A16,A3,I4,A2,I3)')test(I)
   IF(test(I)%experience .LT. 30)THEN
                test(I)%experience=0
                ELSE IF (test(I)%experience.LT.100) then
                test(I)%experience=1
                ELSE
                test(I)%experience=2
  END IF
```

```
WRITE(11)test(I)

END DO

CLOSE(UNIT=10)
CLOSE(UNIT=11)

OPEN(UNIT=20,FILE="out.data",FORM='unformatted')
OPEN(UNIT=21,FILE="final.data",STATUS='new')

DO I=1,Nplayer
 READ(20)test(I)
      IF(test(I)%title .EQ. "U")THEN
                 test(I)%title=" "
      END IF

WRITE(22,FMT='(A2,A,A16,I3)')test(I)%title," ",test(I)%name,
   * test(I)%rating


END DO

CLOSE(UNIT=20)
CLOSE(UNIT=21)

END PROGRAM sections
```

The file containing the data to be manipulated is SectionsCT.txt:

```
100103   Rufenacht   SWZ   2493   GM   568
10362    Hamarat     OST   2630   GM   210
130233   Bures       CZE   2427   IM   208
130826   Kudela      CZE   2457   U    58
130865   Nyvlt       CZE   2510   U    96
130982   Lexa        CZE   2507   IM   62
140688   Tsvetkov    RUS   2447   GM   222
141166   Romanov     RUS   2523   SM   105
```

2.4.3 Debugging the Sample Code

When our sample code is compiled using the XLF compiler and run, it gives the following error:

```
Cu12:~/debugging128% badcode 1525-001 The READ statement
on the file sectionsCT.txt cannot be completed because the end
of the file was reached.  The program will stop.
```

There are two possible and common reasons why the XLF compiler gave this error message:

1. the program is trying to read more data than there is in the file
2. the file "sectionsCT.txt" does not even exist

To check for the latter case, it is easiest to make the OPEN statement more rigorous by adding a status check:

```
OPEN(UNIT=10,FILE="sectionsCT.txt",status='OLD')
```

Now, when we run the program the following more helpful error message is given:

```
Cu12:~/debugging134% badcode
1525-006 The STATUS= specifier in the OPEN statement for unit 10 cannot
be set to OLD because the file sectionsCT.txt does not exist.  The program
will stop.
```

Checking the files in the directory:

```
Cu12:~/debugging137% ls
SectionsCT.txt   badcode*         bugs.f
```

shows that the filename has been misspelled.

(Note that although FORTRAN 90 is not case sensitive, the underlying filesystem usually is case sensitive.)

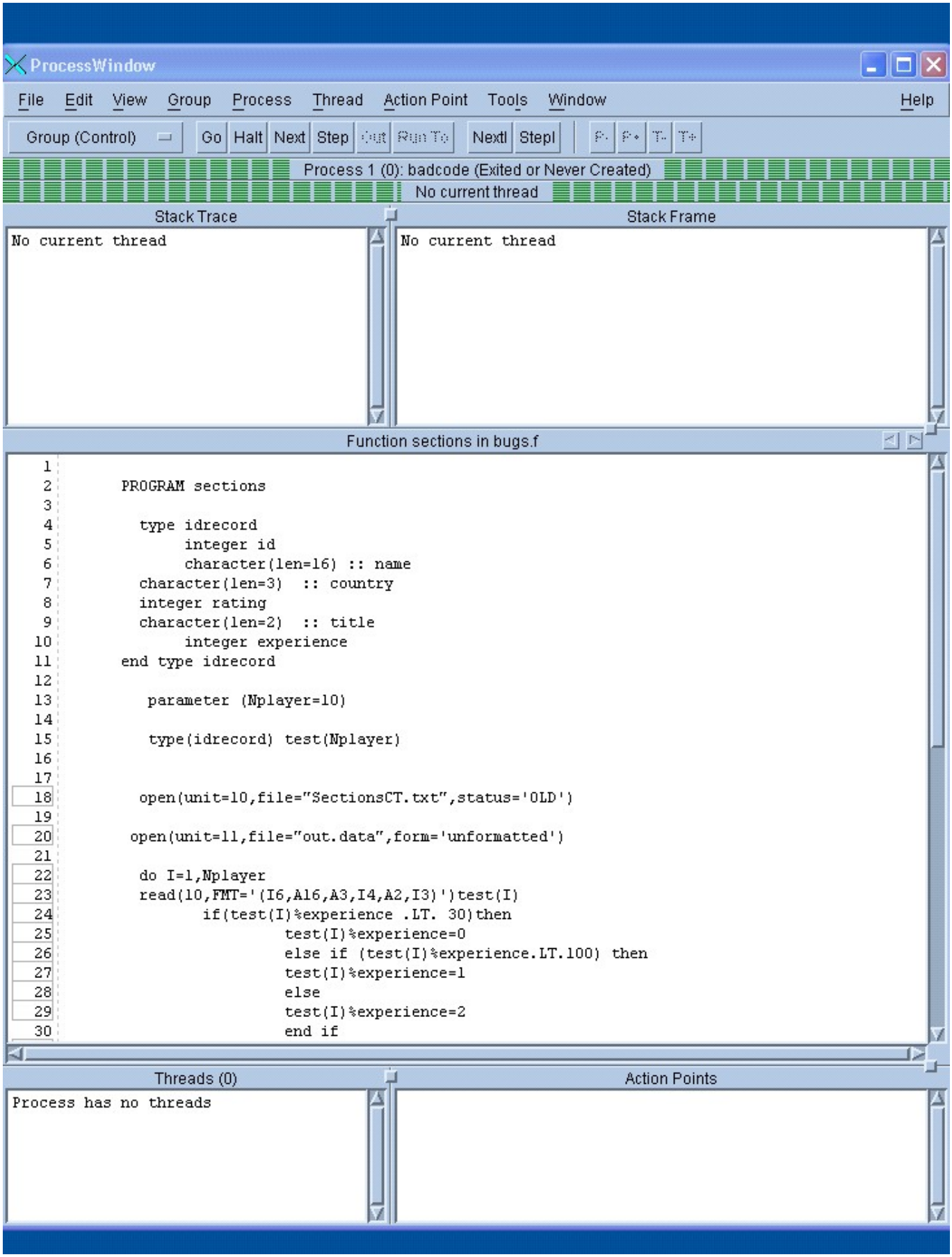
The correct name is "SectionsCT.txt", not "sectionsCT.txt" so we correct the OPEN statement as follows:

```
OPEN(UNIT=10,FILE="SectionsCT.txt",status='OLD')
```

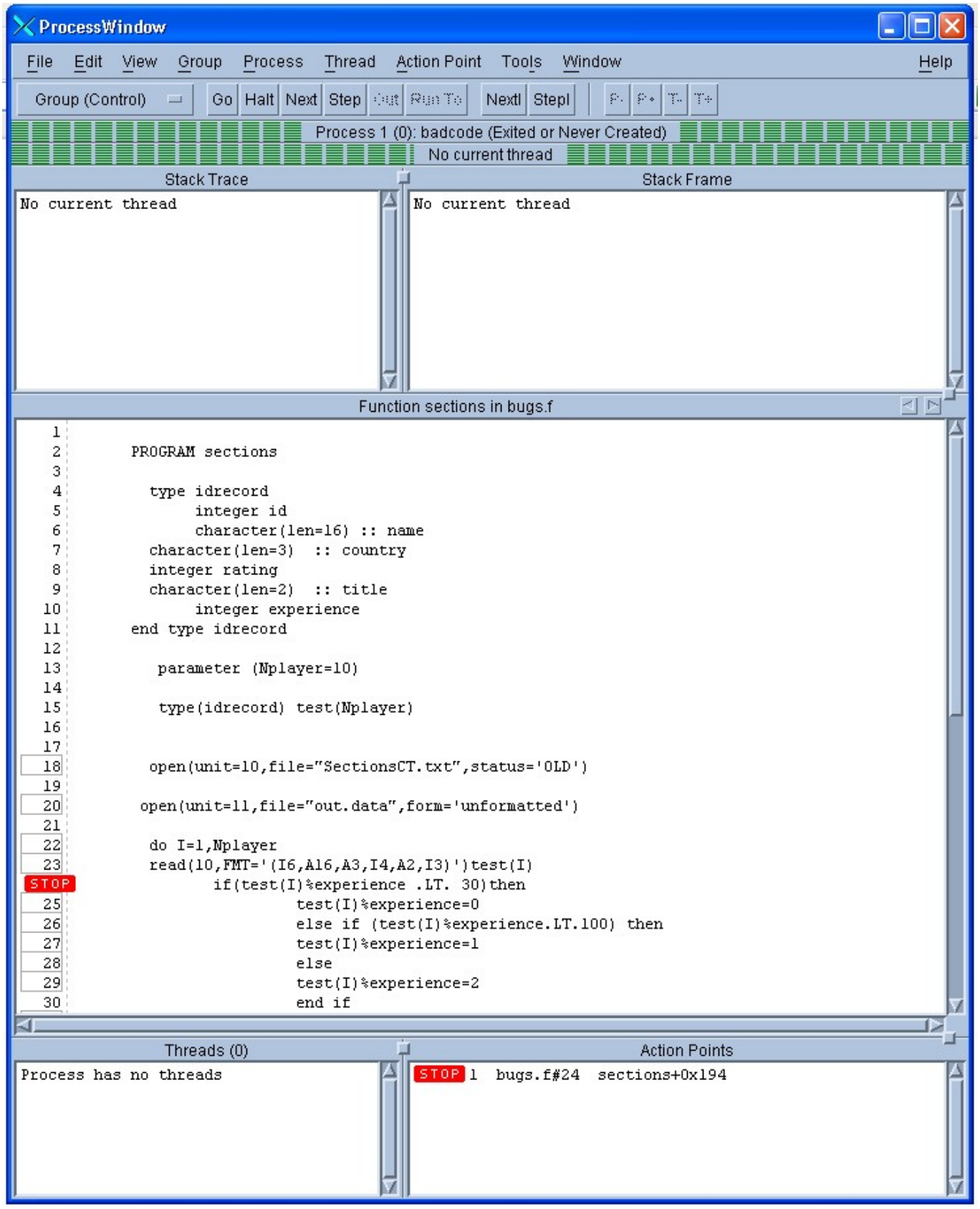
and run the program again, which yields the following screen full of error messages:

```
Cu12:~/debugging141% badcode
1525-097 A READ statement using decimal base input found the invalid
digit '    ' in the input file.  The program will recover by assuming a zero
in its place.
1525-097 A READ statement using decimal base input found the invalid
digit 'G' in the input file.  The program will recover by assuming a zero
in its place.
1525-097 A READ statement using decimal base input found the invalid
digit 'M' in the input file.  The program will recover by assuming a zero
in its place.
.........
1525-001 The READ statement on the file SectionsCT.txt cannot be
completed because the end of the file was reached.  The program
will stop.
```

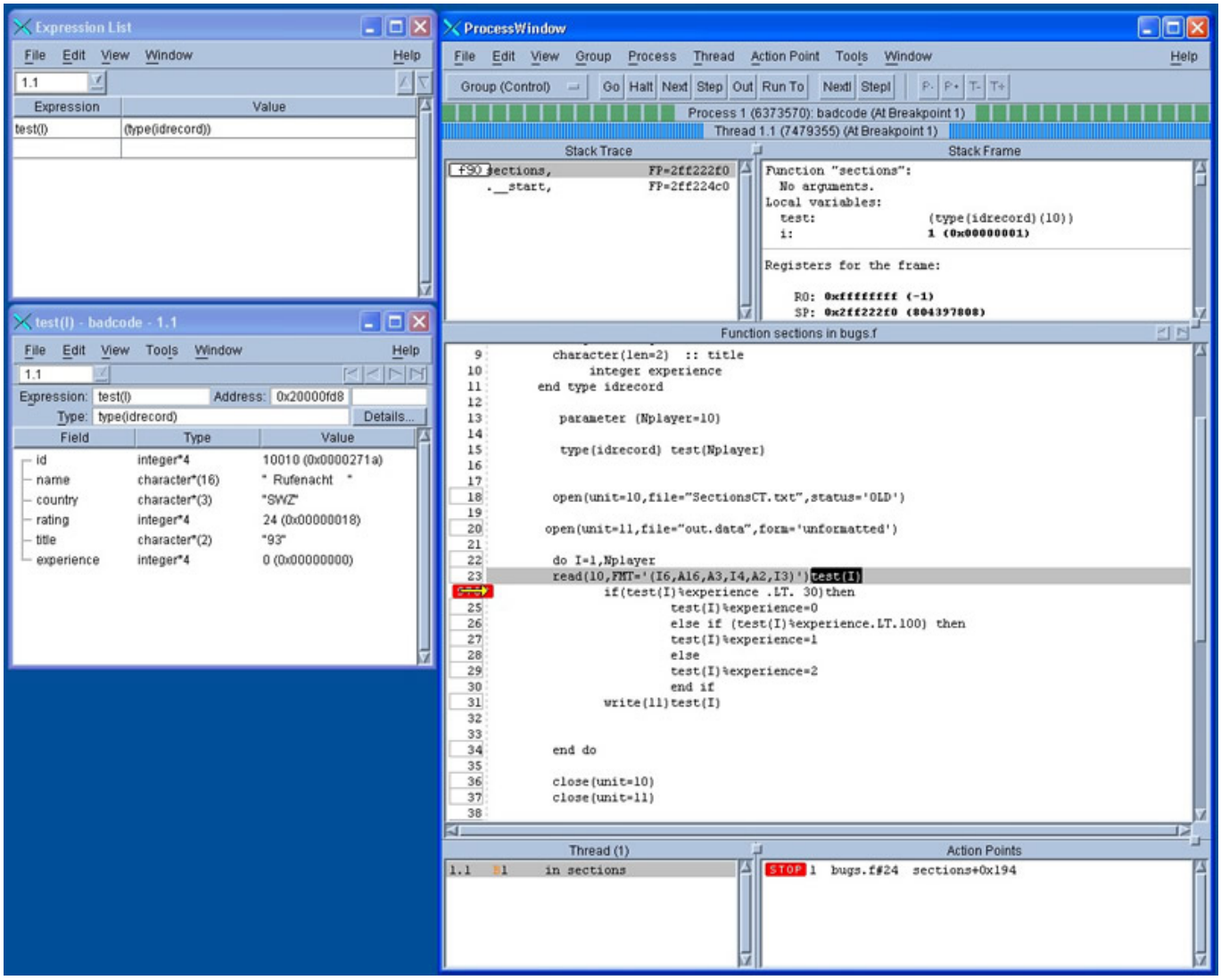
These messages indicate two errors that we will investigate one at a time. The first one indicates a problem with the format specification. We will use the TotalView® debugger to help identify and eliminate this bug. The figure below shows the process window in Totalview® containing the source code listing for the program.



To see how the data is being parsed on import, we need to stop the program after the READ statement by inserting a breakpoint and looking at the contents of *test(1)*. To insert a breakpoint using Totalview®, we click in the box to the left of line 24. The line number is replaced with the red "STOP" indicator as shown in the following figure:



We click "Go" to run the program up to the first time it reaches line 24. To see the values assigned to the fields of the first record in *test*, select *test(I)* and then click your right mouse button and choose "Add to Expression List" from the menu. A new window titled "Expression List" will be displayed. Double clicking on *test(I)* in the new window opens a 3rd window. This window shows the values of the fields for *test(I)*. These two new windows are shown on the left in the figure below:



View larger image

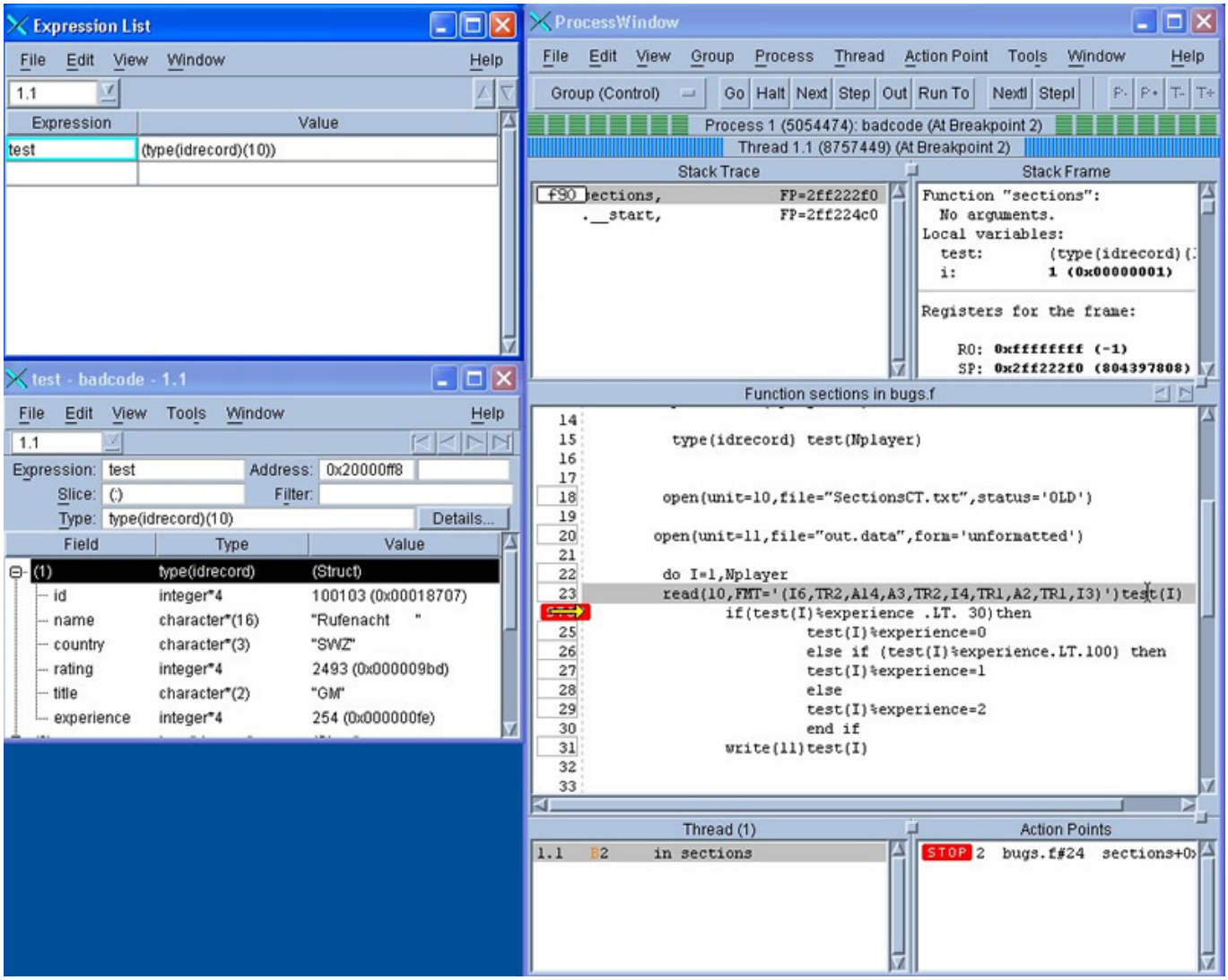
The following table shows the expected and actual results obtained:

	Expected	Actual
Id	100103	100103
Name	“Rufenacht ”	“ Rufenacht ”
Country	“SWZ”	“SWZ”
Rating	2493	24
Title	“GM”	“93”
Experience	254	0

By looking at these values we see that the fields are not getting parsed correctly. The data in the file are separated by spaces that are not accounted for in the READ statement. Adding these spaces to the format line gives:

```
READ(10,FMT='(I6,TR2,A14,A3,TR2,I4,TR1,A2,TR1,I3)')test(I)
```

The following figure shows that once this change is made the data is parsed as expected.



View larger image

Rerunning the program confirms that these errors have been eliminated, leaving the second runtime error that we saw before:

```
Cu12:~/debugging129% badcode
1525-001 The READ statement on the file SectionsCT.txt cannot be
completed because the end of the file was reached.  The program
will stop.
```

Since this is a course on debugging and not on good Fortran coding practices, we will find and remove this bug rather than changing the code to make sure it can never occur. The text of the error message indicates that the end of the file was reached before all of the expected data was read. From our previous correction of the formatting bug, we know that some of the data was read in correctly and contains data of the expected type. Looking at the data file we see that it contains 9 lines:

```
Cu12:~/debugging132% cat SectionsCT.txt | wc -l
9
```

Looking at lines 13 and 22:

```
PARAMETER (Nplayer=10)

    ... ... ...

DO I=1,Nplayer
```

we see that the program is trying to read 10 records. Correcting line 13 to read 9 records:

```
PARAMETER (Nplayer=9)
```

and rerunning the code results in the problem being eliminated with no error messages generated:

```
Cu12:~/debugging136% badcode
Cu12:~/debugging137%
```

However, as the following directory listing shows, there are still problems :

```
Cu12:~/debugging103% ls -l
total 96
-rw-r-----    1 ian      aef            348 Jun 03 15:24 SectionsCT.txt
-rwxr-x---    1 ian      aef          11174 Jun 03 16:41 badcode*
-rw-r-----    1 ian      aef           1482 Jun 03 16:41 bugs.f
-rw-r-----    1 ian      aef              0 Jun 03 16:41 final.data
-rw-r-----    1 ian      aef            216 Jun 03 16:41 fort.22
-rw-r-----    1 ian      aef            396 Jun 03 16:41 out.data
```

The output file "final.data" is a zero byte file and there is an unexpected file "fort.22". This file contains the information expected to be written to "final.data". The suffix, ".22" is a hint that there is a problem with the unit, in this case unit 22. Looking at lines 41 and 49:

```
Line 41: OPEN(UNIT=21,FILE="final.data",STATUS='new')
Line 49: WRITE(22,FMT='(A2,A,A16,I3)')test(I)%title," ",test(I)%name,
```

we find another error — unit 21 is being opened, but the write statement is using unit 22. Changing line 49 to:

```
WRITE(21,FMT='(A2,A,A16,I3)')test(I)%title," ",test(I)%name,
```

and rerunning the code gives another bug:

```
Cu12:~/debugging108% badcode
1525-107 The STATUS= specifier in the OPEN statement for unit 21 cannot
be set to NEW because the file final.data already exists.  The program
will stop.
```

The previous run of the program generated the zero byte file "final.data", however, the OPEN statement is defined such that it needs a new file:

```
OPEN(UNIT=21,FILE="final.data",STATUS='new')
```

Deleting "final.data" and running the program again finally gives us a working program with the expected results.

```
Cu12:~/debugging115% badcode

Cu12:~/debugging116% ls -l
total 64
-rw-r-----   1 ian      aef            348 Jun 03 15:24 SectionsCT.txt
-rwxr-x---   1 ian      aef          11174 Jun 04 09:45 badcode*
-rw-r-----   1 ian      aef           1482 Jun 04 09:45 bugs.f
-rw-r-----   1 ian      aef            216 Jun 04 09:51 final.data
-rw-r-----   1 ian      aef            396 Jun 04 09:51 out.data

Cu12:~/debugging117% cat final.data
GM  Rufenacht      2493
SM  Teichmeister   2536
GM  Hamarat        2630
IM  Bures          2427
    Kudela         2457
    Nyvlt          2510
IM  Lexa           2507
GM  Tsvetkov       2447
SM  Romanov        2523
```

The final debugged version of the program is:

```
PROGRAM sections
  type idrecord
         integer id
         character(len=16) :: name
```

```
  character(len=3)  :: country
  integer rating
  character(len=2)  :: title
          integer experience
end type idrecord

  parameter (Nplayer=9)

  type(idrecord) test(Nplayer)


  open(unit=10,file="SectionsCT.txt",status='OLD')

  open(unit=11,file="out.data",form='unformatted')

  do I=1,Nplayer
    read(10,FMT='(I6,TR2,A14,A3,TR2,I4,TR1,A2,TR1,I3)')test(I)
       if(test(I)%experience .LT. 30)then
          test(I)%experience=0
          else if (test(I)%experience.LT.100) then
          test(I)%experience=1
          else
          test(I)%experience=2
          end if
       write(11)test(I)
  end do

  close(unit=10)
  close(unit=11)

  open(unit=20,file="out.data",form='unformatted')

  open(unit=21,file="final.data",status='new')

  do I=1,Nplayer
    read(20)test(I)
       if(test(I)%title .EQ. "U")then
          test(I)%title=" "
       end if

  write(21,FMT='(A2,A,A16,I4)')test(I)%title," ",test(I)%name,
     * test(I)%rating

  end do

  close(unit=20)
  close(unit=21)

END PROGRAM sections
```

2.4.4 Debugging Exercise

The following code tackles a very typical problem in handling scientific data — reading in an instrument generated file and writing the data out in a modified format. However, there is an error in the code. See if you can identify the coding error and correct it so that the code runs successfully. The data for the program is in the file 15kdata.txt. For the sake of brevity the length of the datafile, 187 lines, has been hard coded into the program.

```
PROGRAM reformat


REAL x(187), y(187), sigy(187)

open(unit=10,file="15kdata.txt",status='OLD')

open(unit=11,file="data.15k",status='NEW')

write(11, *)" x       y        sig(y)"

do I=1,187
```

```
    read(10,FMT='(F8.4,F8.4,F8.4)')x(I),y(I),sigy(I)

    write(11,FMT='(F8.3,F8.3,A4,F8.3)')x(I),y(I)," +/-",sigy(I)

  end do

close(unit=10)
close(unit=11)

END PROGRAM reformat
```

2.4.5 Exercise Solution

The problem with the code in this exercise is that the formatted input and output use the wrong format descriptors. The data in the file is in exponential form, which requires the use of "E" not "F" in the format statement. The debugged code is shown below:

```
PROGRAM reformat

REAL x(187), y(187), sigy(187)

open(unit=10,file="15kdata.txt",status='OLD')

open(unit=11,file="data.15k",status='NEW')

write(11, *)" x       y        sig(y)"

  do I=1,187
    read(10,FMT='(F8.4,E13.4,E16.4)')x(I),y(I),sigy(I)

    write(11,FMT='(F6.4,F8.4,A4,F7.4)')x(I),y(I)," +/-",sigy(I)

  end do

close(unit=10)
close(unit=11)

END PROGRAM reformat
```

The program's output can be seen in the file data.15k.

2.5 Dummy and Actual Arguments Mismatch

2.5.1 Dummy and Actual Arguments Mismatch

Introduction

When a subroutine is called from within a program, the program call's argument list must match that of the subroutine it is calling. Since typically there are multiple items in a subroutine's argument list it can be easy to inadvertently create a mismatch between the subroutine's dummy argument list and the actual argument list passed by the corresponding program call. Some of the more frequent types of argument mismatches are:

- The number of terms in the argument list do not match
- The data types of the terms in the argument list do not match
- Data pass-by-value mismatch (When a C program calls a Fortran subprogram, all variables (including scalars) must be passed by reference instead of by value.)

Compilers react quite differently to these mismatches. Some may issue a warning while others may not. For C programs that use prototyping and Fortran 90 programs that declare subroutines via the f90 construct interface, the compiler can readily perform an argument list compatibility check to see if there are any mismatches. However, for C programs that do not take advantage of prototyping and Fortran 90 (and Fortran 77) programs with subroutines not declared in interfaces, the compiler does not have sufficient information to assist in identifying these discrepancies. Consequently, the linker may proceed to generate executables with the mismatch errors going unchecked. At runtime, this can lead to executables that produce erroneous calculations or that exit prematurely from computations due to an error such as a segmentation fault. In these situations, a debugger can be helpful in tracing the problem source.

Objectives

In this lesson, you will learn about common argument mismatch errors and how to identify them. Sample programs written in Fortran are provided to illustrate the types of error messages generated by argument mismatches as well as debugger sessions using gdb to show how to identify the errors.

2.5.2 Sample Code with Argument Mismatches

We now give code examples written in Fortran to examine the most common types of argument mismatches. C codes are also made available in links where Fortran codes are shown.

Please note that using a different compiler or debugger &#151 for that matter even a different version of the same compiler or debugger &#151 could result in a somewhat different behaviors in the debugger.

Number of Terms in Argument List Mismatch

The following code, miss1.f, is an example of a mismatch between the number of terms in the argument list of the calling routine and the subroutine: (C version : miss1.c).

```
program miss1
implicit none
integer i, n
parameter (n=10)
real A(n)
data A/1.,2.,3.,4.,5.,6.,7.,8.,9.,10./

write(*,*) A

i = 2
call foo(A,n)     ! missing third item, "i"

end

subroutine foo(A,n,i)
implicit none
integer n, i
real A(n)

write(*,*)'====> ',i,n,A(i)

end
```

In this example, the calling routine passes only two arguments whereas the subroutine is expecting three arguments. This code generally compiles and links to produce an executable with no complaints from the linker regarding the argument list mismatch. A notable exception is the GNU g77/gcc compiler, which generates an executable along with the following warning messages regarding the mismatch:

```
cootie:PACS/debugger % g77 -fversion

GNU Fortran (GCC) 3.3.1



cootie:PACS/debugger % g77 -o miss1 miss1.f

miss1.f: In subroutine 'foo':

miss1.f:11: warning:

         call foo(A,n)

              1

miss1.f:15: (continued):

         subroutine foo(A,n,i)

                    2
```

```
Too many arguments for 'foo' at (1) versus invocation at (2) [info -f g77 M GLOBALS]
```

Since the compiler gives this warning message, no debugger is needed to identify the problem.

Some compilers, while not able to provide diagnostics at compile time, produce a *segmentation fault* at runtime and effect a core dump. In these cases, a debugger can be used to identify where the error occurs and trace its cause. Below we show one such session in which the code is compiled with the Portland Group Fortran 77 pgf77 compiler and then debugged with the GNU Project Debugger (GDB):

```
cootie:PACS/debugger % pgf77 -o miss1 miss1.f -g

cootie:PACS/debugger % miss1

    1.000000       2.000000       3.000000       4.000000       5.000000

    6.000000       7.000000       8.000000       9.000000       10.00000

Segmentation fault (core dumped)


cootie:PACS/debugger % gdb miss1 core

GNU gdb 5.3

Copyright 2002 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are

welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB.  Type "show warranty" for details.

This GDB was configured as "i686-pc-linux-gnu"...

Core was generated by `miss1'.

Program terminated with signal 11, Segmentation fault.

Reading symbols from /lib/i686/libm.so.6...done.

Loaded symbols for /lib/i686/libm.so.6

Reading symbols from /lib/i686/libc.so.6...done.

Loaded symbols for /lib/i686/libc.so.6

Reading symbols from /lib/ld-linux.so.2...done.

Loaded symbols for /lib/ld-linux.so.2

#0  0x08049090 in foo (a=(134558200), n=134558184, i=10) at miss1.f:20

20            write(*,*)'====> ',i,n,A(i)
```

The listing above points to line 20 within the subroutine "foo" as the potential problem area. Using this information, we instruct the debugger to run the program and stop in "foo". At which point, we print the key information, i.e., *i, n, A(i),* to see if they have the values expected.

```
(gdb) break foo

Breakpoint 1 at 0x8049006: file miss1.f, line 18.

Current language:  auto; currently fortran

(gdb) run
```

```
Starting program: /project/scv/kadin/PACS/debugger/miss1

    1.000000       2.000000       3.000000       4.000000       5.000000

    6.000000       7.000000       8.000000       9.000000       10.00000



Breakpoint 1, foo (a=(134558200), n=134558184, i=10) at miss1.f:18

18            real A(n)

(gdb) list
13            end
14
15            subroutine foo(A,n,i)
16            implicit none
17            integer n, i
18            real A(n)
19
20            write(*,*)'====> ',i,n,A(i)
21
22            end

(gdb) step
20            write(*,*)'====> ',i,n,A(i)

(gdb) print i
 = 10

(gdb) print n
(gdb) break foo Breakpoint 1 at 0x8049006: file miss1.f, line 18. Current language: auto; currently fortran (gdb)
run Starting program: /project/scv/kadin/PACS/debugger/miss1 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000
7.000000 8.000000 9.000000 10.00000 Breakpoint 1, foo (a=(134558200), n=134558184, i=10) at miss1.f:18 18 real A(n)
(gdb) list 13 end 14 15 subroutine foo(A,n,i) 16 implicit none 17 integer n, i 18 real A(n) 19 20 write(*,*)'====>
',i,n,A(i) 21 22 end (gdb) step 20 write(*,*)'====> ',i,n,A(i) (gdb) print i $1 = 10 (gdb) print n $2 = 134558184
(gdb) print A(1) $3 = 134558184 (gdb) quit = 134558184

(gdb) print A(1)
 = 134558184

(gdb) quit
```

Since the values of the printed variables are not correct, we suspect that the variables were not passed into the routine correctly. Upon inspecting the argument list, we find that indeed there is a missing parameter in the argument list of the call to "foo".

Data Type Mismatch

Now we show an example program, miss2.f, with an argument list data type mismatch. (C version: miss2.c and foo.c) The subroutine "foo" expects an array of type real while the calling program mistakenly uses a double precision array.

```
program miss2
implicit none
integer i, j, n, m
double precision B(10)
data B/1.,2.,3.,4.,5.,6.,7.,8.,9.,10./

n = 10
i = 2

call foo(B, n, i)   ! passing wrong data type

end

subroutine foo(A,n,i)
implicit none
integer n, i
real A(n)
```

```
write(*,*)'i, n, A(i) in foo : ',i,n,A(i)

end
```

As in the previous example, the Gnu g77/gcc compiler produces a warning message on the mismatch in the code:

```
cootie:PACS/debugger % g77 -o miss2 miss2.f

miss2.f: In subroutine 'foo':

miss2.f:12: warning:

         call foo(B, n, i)

              1

miss2.f:16: (continued):

         subroutine foo(A,n,i)

                    2

Argument #1 (named 'a') of 'foo' is one precision at (2) but is some other precision at
(1) [info -f g77 M GLOBALS]
```

If, for instance, the Intel ifc compiler is used to compile the above program, no warning is given. As a result, a debugger is needed to trace the problem.

```
cootie:PACS/debugger % gdb miss2

(gdb) break foo

Breakpoint 1 at 0x804959b: file miss2.f, line 21.

(gdb) run

Starting program: /project/scv/kadin/PACS/debugger/miss2



Breakpoint 1, foo (a=(), n=10, i=2) at miss2.f:21

21            write(*,*)'i, n, A(i) in foo : ',i,n,A(i)

Current language:  auto; currently fortran

(gdb) print i

 = 2

(gdb) print n
```

cootie:PACS/debugger % **gdb miss2** (gdb) **break foo** Breakpoint 1 at 0x804959b: file miss2.f, line 21. (gdb) **run** Starting program: /project/scv/kadin/PACS/debugger/miss2 Breakpoint 1, foo (a=(), n=10, i=2) at miss2.f:21 21 write(*,*)'i, n, A(i) in foo : ',i,n,A(i) Current language: auto; currently fortran (gdb) **print i** $1 = 2 (gdb) **print n** $2 = 10 (gdb) **print A(i)** $3 = 1.40129846e-44 (gdb) **next** i, n, A(i) in foo : 2 10 1.875000 23 end = 10

```
(gdb) print A(i)

 = 1.40129846e-44

(gdb) next

 i, n, A(i) in foo :            2          10   1.875000

23            end
```

The printed value of *A(2)* is "1.875", which is incorrect since it should be "2". In addition, the value of A(2) obtained by

using the **"print"** command is also incorrect as well as different from that which uses the fortran **write** statement. Upon careful examination, we see that there is a mismatch in data type precision. Admittedly, this sample code is very small and tracing the problem through a debugger is a trivial matter. From a practical viewpoint, to avoid excessive stepping in larger codes you need to have some idea of the approximate location of the potential error before using the debugger.

Note also that when the variable you are trying to print is connected with the bug (like *A* in this example), the behavior of the debugger depends on the implementation and hence the printed value can vary accordingly.

Data Pass-By-Value Mismatch

The following code, miss3.c, is an example of a data pass-by-value mismatch:

```
#include <stdio.h>

int main()
{
  int i, n;
  double B[10];

  n = 10;

  for (i=0; i□n; i++) {
    B[i] = (double) i;
  }

  i = 2;

  foo_(B, n, i);     /* should pass pointers &n, &i */

  return 0;
}
```

The following Fortran subroutine, foo.f, is called by miss3.c.

```
subroutine foo(A,n,i)
implicit none
integer n, i
real A(n)

write(*,*)'i, n, A(i) in foo : ',i,n,A(i)

end
```

Executing and debugging the code gives:

```
cootie:PACS/debugger % g77 -c -g foo.f

cootie:PACS/debugger % gcc -o miss3 miss3.c foo.o -g -lg2c

cootie:PACS/debugger % miss3

Segmentation fault (core dumped)

cootie:PACS/debugger % gdb miss3 core

GNU gdb 5.3


  . . .

(gdb) break foo_

Breakpoint 1 at 0x804852e: file foo.f, line 6.

(gdb) run

Starting program: /project/scv/kadin/PACS/debugger/miss3
```

```
Breakpoint 1, foo_ (a=0xbfffd950, n=0xa, i=0x2) at foo.f:6

6             write(*,*)'i, n, A(i) in foo : ',i,n,A(i)

Current language:  auto; currently fortran

(gdb) print *i

Cannot access memory at address 0x2

(gdb) print *n

Cannot access memory at address 0xa

(gdb) print *A

 = (1)

(gdb) quit
```

Had the variables *i* and *n* been passed to the routine *foo* by reference:

```
foo_(B, &n, &i);
```

the print statements would have yielded the correct values (which are 2 and 10, respectively).

2.5.3 Exercises

1. Use your favorite debugger to test the miss1.f (or miss1.c) number of arguments mismatch program.
2. Use your favorite debugger to test the miss2.f (or miss2.c) data type mismatch program.
3. Use your favorite debugger to test the miss3.c data pass-by-value mismatch program.
4. Rewrite miss1.f in Fortran 90 and use module/interface to "prototype" foo.f so that the compiler can detect argument mismatch.
5. Modify miss1.c to prototype function foo so that the compiler can detect argument mismatch.

2.5.4 Solutions

1. Use your favorite debugger to test the miss1.f (or miss1.c) number of arguments mismatch program. There is no actual "solution" for this exercise. If you need help please see the section "Sample Code with Argument Mismatches".
2. Use your favorite debugger to test the miss2.f (or miss2.c) data type mismatch program. There is no actual "solution" for this exercise. If you need help please see the section "Sample Code with Argument Mismatches".
3. Use your favorite debugger to test the miss3.c data pass-by-value mismatch program. There is no actual "solution" for this exercise. If you need help please see the section "Sample Code with Argument Mismatches".
4. Rewrite miss1.f in fortran90 and use module/interface to "prototype" foo.f so that the compiler can detect argument mismatch. Here is the f90 program miss1_90.f90:
```
module mymodule
  interface
    subroutine foo(A,n,i)
      implicit none
      integer n, i
      real A(n)
    end subroutine foo
  end interface
end module mymodule

      program miss1_90
      use mymodule
      implicit none
      integer i, n
      parameter (n=10)
      real A(n)
      data A/1.,2.,3.,4.,5.,6.,7.,8.,9.,10./

      write(*,*) A

      i = 2
      call foo(A,n)     ! missing third term "i"

      end
```

foo.f :

```
      subroutine foo(A,n,i)
      implicit none
      integer n, i
      real A(n)

      write(*,*)'i, n, A(i) in foo : ',i,n,A(i)

      end
```

5. Modify miss1.c to prototype function foo so that the compiler can detect argument mismatch. Here is the C program miss1_c.c:

```
#include <stdio.h>

void foo(float* A, int* n, int* i);  /* prototyping */

#define n 10

int main()
{
  int i;
  float A[n];

  for (i=0; i<n; i++) {
    A[i] = (float) i+1;
  }

  i = 2;
  foo(A, n);   /* missing the third argument */

  return 0;
}
```

2.6 Debugging Infinite Loops with gdb

2.6.1 Debugging Infinite Loops with gdb

Introduction

An extremely common programming bug is the **infinite loop**, which is simply a looping construct that never terminates. Infinite loops are most common in C/C++ while (cond) { ... } loops and Fortran DO WHILE (cond) ... END DO loops where the termination condition is ill-formed or never reached.

Objectives

In this lesson, you will learn how to diagnose infinite loops in a numerical calculation example and a string handling example using The GNU Project Debugger (GDB).

2.6.2 An Infinite Loop in a Numerical Calculation

Consider the following C program that solves Laplace's equation on a uniform two-dimensional mesh:

```
#include <stdio.h>
#include <math.h>

#ifndef IMAX
#define IMAX 2001
#endif /* !IMAX */

#ifndef JMAX
#define JMAX 2001
#endif /* !JMAX */

#ifndef UMAX
#define UMAX 1.0
#endif /* !UMAX */

#ifndef EPSILON
#define EPSILON 0.00001
#endif /* !EPSILON */

#ifndef PI
#define PI 3.141596256
#endif /* !PI */

#define max(a,b) ((a>b)?a:b)

int main(int argc, char *argv[])
{
  double u[IMAX][JMAX];
  double du[IMAX][JMAX];
  double dumax=1.0+EPSILON;
  int i,j,it;

  /* Initialize u */
  for (i=0;i<IMAX;i++)
    {
      for (j=0;j<(JMAX-1);j++)
        {
          u[i][j]=0.0;
        }
      u[i][JMAX-1]=UMAX*sin(PI*(float)(i)/(float)(IMAX-1));
    }

  /* Main loop */
  it=0;
  while ( ( dumax>EPSILON && it<1000 ) )
    {
      dumax=0.0;
      for (i=1;i<(IMAX-1);i++)
        for (j=1;j<(JMAX-1);j++)
          {
            du[i&#93[j]=(u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1])/4.0-u[i][j];
            dumax=max(dumax,fabs(du[i][j]));
          }
      for (i=1;i<(IMAX-1);i++)
        for (j=1;j<(JMAX-1);j++)
          u[i][j]+=du[i][j];
      it--;
    }

  printf("e=%le after %d iterationsn",dumax,it);

  return(it);
}
```

This program uses a while loop to allow it to exit if either the residual falls below a fixed value or the maximum iteration count of 1000 is exceeded. However, if run in its current state, it will continue to run until the residual tolerance is reached regardless of the number of iterations required.

To debug this with GDB, we must first compile the code with the GNU C compiler without optimization and debugging

symbol generation enabled as follows:

```
troy@piv-login1:~/debug$ gcc -g -O0 inf-loop.c -o inf-loop -lm
```

We then need to start GDB on the resulting executable:

```
troy@piv-login1:~/debug$ gdb inf-loop
GNU gdb Red Hat Linux (5.3.90-0.20030710.40rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".

(gdb)
```

To track the behavior of the program, we set a watchpoint on one of the variables being used to decide if the loop should end or not. A watchpoint will cause the debugger to break program execution any time a specific variable is modified. The residual *dumax* is a poor choice for this, because it is modified once per grid point every trip through the loop and there are 2001^2 grid points by default. However, the iteration counter *it* should be changed only once per trip, so it is a much better candidate:

```
(gdb) break inf-loop.c:1
Breakpoint 1 at 0x80483cc: file inf-loop.c, line 1.

(gdb) run
Starting program: /c/troy/debug/inf-loop

Breakpoint 1, main (argc=134513612, argv=0x1) at inf-loop.c:27
27      {

(gdb) watch it
Hardware watchpoint 2: it

(gdb)
```

Once we have a watchpoint set, we need to continue running the program for a while and watch how its value changes:

```
(gdb) continue
Continuing.
Hardware watchpoint 2: it

Old value = 0
New value = -1
0x08048858 in main (argc=1, argv=0xbfffee74) at inf-loop.c:57
57            it--;

(gdb) continue
Continuing.
Hardware watchpoint 2: it

Old value = -1
New value = -2
0x08048858 in main (argc=1, argv=0xbfffee74) at inf-loop.c:57
57            it--;

(gdb)
```

Notice that the value of *it* is going down, not up. Closer inspection shows that we are decrementing our iteration counter with the -- operator, rather than incrementing it with the ++ operator. Hence, we have found our bug. To fix this, all we need to do is change *it*-- to *it*++.

2.6.3 An Infinite Loop in String Handling

Consider the following C program taken from RMS's GDB Tutorial: Infinite Loop Example:

```
#include <stdio.h>
#include <ctype.h>

int main(int argc, char **argv)
{
  char c;

  c = fgetc(stdin);
  while(c != EOF)
    {
      if(isalnum(c))
        printf("%c", c);
      else
        c = fgetc(stdin);
    }

  return 1;
}
```

This should take a string read from *stdin* and write it back to *stdout*, but instead it endlessly prints the first character of the input string. To debug our problem we follow the same steps we took for the Laplace's equation example in the previous section:

```
troy@piv-login1:~/debug$ gcc -g -O0 chario-infloop.c -o chario-infloop

troy@piv-login1:~/debug$ gdb chario-infloop
GNU gdb Red Hat Linux (5.3.90-0.20030710.40rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".

(gdb) break chario-infloop.c:1
Breakpoint 1 at 0x80483f4: file chario-infloop.c, line 1.

(gdb) run
Starting program: /c/troy/debug/chario-infloop

Breakpoint 1, main (argc=134513652, argv=0x1) at chario-infloop.c:6
6       {

(gdb) watch c
Hardware watchpoint 2: c

(gdb) continue
Continuing.
foo
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffff...
```

The "foo" string is necessary for the program to have something to do. However, once it gets the first character of that string, it will keep running forever until interrupted. Clearly a watchpoint is not going to work in this case, so we'll need to try a slightly different approach. Instead, we quit that gdb session, start a new one, and try setting break points on the specific lines where *c* should be updated:

```
...fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff...
^C
Program received signal SIGINT, Interrupt.
0x400fbd38 in write () from /lib/i686/libc.so.6
(gdb) quit
The program is running.  Exit anyway? (y or n) y

troy@piv-login1:~/debug$ gdb chario-infloop
GNU gdb Red Hat Linux (5.3.90-0.20030710.40rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
```

```
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".

(gdb) break chario-infloop.c:9
Breakpoint 1 at 0x8048404: file chario-infloop.c, line 9.

(gdb) break chario-infloop.c:15
Breakpoint 2 at 0x8048453: file chario-infloop.c, line 15.

(gdb) run
Starting program: /c/troy/debug/chario-infloop

Breakpoint 1, main (argc=1, argv=0xbfffcfe4) at chario-infloop.c:9
9         c = fgetc(stdin);

(gdb) continue
foo
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffff...
```

While this may seem like another failed attempt, it is not. It tells us one very important fact: the program never executes line 15, the second instance "c = fgetc(stdin);". If we look at that section of code more closely, we see it is in the else clause of an if construct that is always true, so it is never executed. Removing the else line fixes the program.

2.6.4 Debugging Exercise

The program ch3.5-ex1.c is similar to the Laplace solver example shown earlier, except that it does not use an iteration counter as an explicit termination condition. It also will run infinitely as it is currently written. In this exercise, identify the problem(s) with it.

2.6.5 Exercise Solution

The main calculation loop of ch3.5-ex1.c is as follows:

```
while ( dumax>EPSILON )
  {
    dumax=0.0;
    for (i=1;i
```

2.7 Pointer Errors

2.7.1 Pointer Errors

Introduction

A pointer is a variable that contains a memory address. Usually this memory address identifies the location of another variable in memory. Therefore, we can say that a pointer is used to "point" to the location of some other variable. (A pointer can actually be set to point to nothing, which is called a null pointer.)

Pointers represent an important tool for writing C and C++ programs. Pointers also can be used with Fortran 90/95; however, in this lesson we focus exclusively on C/C++ code. Pointers can be used to:

- improve coding efficiency
- allocate memory dynamically (using malloc() or the new operator in C++)
- provide a way for a function to modify the value(s) of its argument(s)

Consequently, a good conceptual understanding of pointers is absolutely necessary when programming in C/C++.

Unfortunately, as important as pointers are in writing C/C++ code, they can also cause major problems when used incorrectly. The incorrect use of a pointer can corrupt your systems memory, crash your program, or, in some extreme cases, even crash your entire system. In less drastic cases, an incorrectly used pointer may simply be caught by the compiler, which will prevent your code from compiling at all. In cases where your code does compile, running the executable will generate a *segmentation fault* and the run will simply fail to complete successfully. However, there are those really insidious pointer bugs for which there is no clue of their existence other than the fact that the program

completes successfully but gives totally incorrect results. This latter type of pointer errors can be especially difficult to debug.

Objectives

In this lesson, you will learn about errors that are commonly made when using pointers and how to use The GNU Project Debugger (GDB) to debug code containing these pointer errors. A sample program with pointer errors is given that you may download and debug on your own while following along with the procedure described here.

2.7.2 Common Pointer Errors

There are a number of different errors that you can make when using pointers. The most common ones are those listed and described below.

Failing to prefix each pointer name with the indirection operation, *.

When declaring multiple pointers within a single declaration, you must prefix each pointer name with the indirection operation. For example,

```
int *p, q;
```

declares a pointer to an integer, *p*, and an integer variable, *q*. *q* is not a pointer to *int* because the * operator does not distribute across multiple variable names. To declare both *p* and *q* as pointers to *int*, you must write

```
int *p, *q;
```

Failing to dereference a pointer.

You need to *dereference* a pointer, or access the value to which a pointer points, rather than use the pointer itself. The following example illustrates a failure to dereference a pointer:

```
int *p, x, y = 10;
p = &y;        /* initialize p to point to the address of the variable y  */

x = p + 5;    /* Error !!  p is an address, not an integer value  */
```

What should have been done (and was probably meant to be done) was to dereference *p* before attempting to add the integer 5,

```
x = *p + 5;
```

Attempting to dereference an uninitialized pointer.

A pointer must be initialized before it can be dereferenced. The following example shows a pointer being dereferenced before it is initialized:

```
int *p, y = 72;
*p = y;

printf("The value of *p is %d\n", *p);        /* Error !!   */
```

The pointer *p* was not initialized before it was used in the *printf* statement. Consequently, *p* does not point to any specific location in memory. The statement in the second line of this code fragment simply assigns the value 10 to some unknown memory location. Since this assignment statement does not tell *p* the location of this data in memory, *p* has no idea where it is supposed to be pointing. The *printf* statement then attempts to access integer data, **p*, from an unknown location in memory (which it cannot do) and then generates an error.

The code should have been written as follows:

```
int *p, x = 72;
p = &x;               /* initialize p to point to the address of the variable x  */

printf(" *p = %d\n", *p);       /*  This would print the statement  *p = 72  */
```

Attempting to dereferencing a null pointer

A null pointer is a pointer that points to nothing. The following is an example of dereferencing a null pointer:

```
int *p = 0;         /*  a null pointer  */
printf(" *p = %d\n", *p);       /*  Error !!  */
```

Attempting to dereferencing a pointer to void.

A void pointer is a pointer that points to an unknown data type. Consequently, the actual number of bytes of memory to which the pointer refers is also unknown. However, in order to dereference a given pointer, the compiler must know (from the specified data type in the pointer declaration) the number of bytes to be dereferenced for that pointer. Since the number of bytes cannot be determined from the data type void, attempting to dereference a void pointer causes the compiler to generate an error. The following code gives an example of dereferencing a void pointer:

```
void *p;           /*  a void pointer  */

printf(" *p = %d\n", *p);       /*  Error !!  */
```

Assigning a pointer of one type to a pointer of a different type.

One pointer should be directly assigned to a second pointer only if both pointers point to the same data type. Although casting a pointer to a compatible type can be used in an assigning pointers of different types, sometimes important data can be lost if you are not careful (e.g., casting a pointer to float to a pointer to int) leading to incorrect or inaccurate results. The one exception to this rule is assigning the pointer to void. Any pointer type can be assigned a pointer to void (without casting). In contrast, a pointer to void cannot be assigned directly to a pointer of any other data type without first casting the void pointer to the proper pointer type.

In the following the code fragment:

```
float *fl, x = 2.45
int *p;

fl = &x;               /* initialize fl to point to the address of the variable x  */
p = fl;        /*  Error !!  assigning pointer to float to pointer to int  */
```

the compiler will complain about incompatible pointer types. Although the code will very likely still run, It will probably generate incorrect results.

Initializing a pointer using a data value instead of the address of that data.

Pointers must always be initialized with an address (or with 0 for a null pointer), never with a value. The following example shows a pointer being initialized to a data value rather than an address:

```
int *p, x = 72;
p = x;               /*  Error !!   p is initialized to have the value of x instead
                                                                  of the address of x, &x    */
```

Using pointer arithmetic on a pointer that does not refer to an array.

In the following example:

```
int n, k, *p, *q, *r;
int array[4] = { 5, 10, 15, 20 };         / *  declare and initialize an array  */

p = &n;
q = array;       /*  q initialized to point to address of the first element in
                         array, &array[0]    */
```

using pointer arithmetic we could write:

```
k = *(q + 2);
```

assigning *k* the value of the third element in *array*, k = array[2] = 15. By using pointer arithmetic, we could also initialize the pointer *r* to make it point to the second element in *array, array[1] = 10*.

```
r = q + 1;          /*  this is the same assignment as r = &array[1]  */
```

Now if we try to do this same thing with pointer *p*, we get into serious trouble.

```
k = (p + 2);         /*  Error !!  */
r  = p + 1;           /*  Error !!  */
```

The problem is that *p* points to a single a block of memory of size sizeof(int) bytes. Outside of this single block of memory to which *p* is pointing (which is set during initialization, *p = &n*), the behavior of the pointer *p* is completely unpredictable. Consequently, adding 1 or 2 (or any number other than 0) to *p* takes us outside the sizeof(int)-byte block of memory to which *p* points. This is when we find we are using a pointer over which we now have no control.

In contrast, all the elements of our array are stored in a contiguous block of memory of length (*N*sizeof(int)*) bytes. The pointer q = array (= &array[0]) points to the first element in the array. Consequently, *(q + 1)* points to a location in memory which is offset from *&array[0]* by sizeof(int) bytes which is the address of the second element of the array, *&array[1]*. Similarly, *(q + 2)* points to a location in memory which is offset from *&array[0]* by *(2*sizeof(int))* bytes which is the address of the third element of the array, *&array[2]*. Therefore, as long as we remain within the (*N*sizeof(int)*)-byte block of memory defined for our array, the behavior of *p* will be predictable. However, if we continue to move along the elements of our array until we have moved outside of the block of memory in which our array is located (viz., we run off the end of the array), we find ourselves right back into the same situation we were in when we attempted to use pointer arithmetic on the pointer *p*.

```
k = *(q + 4);       /*  Error !!  takes us beyond the upper boundary of our
                                    array whose address is that of the 4th element
                                    of our array, (q + 3) = &array[3].     */
```

Attempting to modify the name of an array.

The following example attempts to modify the name of an array:

```
int q[4] = { 1, 2, 3, 4 };
int *p;
p = q;            /*  Initialize pointer p to point to first element in the array   */

printf("%d\n", *p);             /*  prints the first element of array q  */

q += 1;    /* Error !!  expression attempts to modify the name of the array  */
p = q;

printf("%d\n", *p);
```

However, the name of an array cannot not be modified using pointer arithmetic. That is, *(q + 1)* is not a pointer to the second element of the array *q*, *&q[1]*.

In contrast, if a pointer to an array is declared and initialized to point to the first element (or some other element) in the array, then pointer arithmetic can be used with this pointer to walk up and down the array. Therefore, to eliminate the error, the following change would need to be made in the above code fragment.

```
int q[4] = { 1, 2, 3, 4 };
int *p;
p = q;            /*  Initialize pointer p to point to first element in the array   */

printf("%d\n", *p);             /*  prints the first element of array q, q[0]  */

p += 1;       /*  point to a new memory location sizeof(int) bytes downstream
                  from &q[0] which is exactly the memory location of q[1]  */

printf("%d\n", *p);             /*  prints the second element of array q, q[1]  */
```

Note carefully that when *ptr* is a pointer, the value of *(ptr + 1)* is not *&ptr + 1* but is *&ptr + sizeof(pointer type)*, where pointer type is the data type of the value pointed to by *ptr*.

Subtracting (or otherwise comparing) pointers that do not point to elements of the same array.

You can only subtract, or otherwise compare, pointers that point to elements of the same array. In the following example, the statement x = r - p; is alright since *r* and *p* both point to elements of array *p*. However, the statement y = r-q; causes an error since *r* and *q* point to different arrays.

```
int x, y;
int *p, *q, *r;
p = (int *) malloc(7*sizeof(int));
q = (int *) malloc(5*sizeof(int));

r = &p[4];        /* Initialize r to point to the 5th element of array p  */
```

```
x = r □ p;        /*  print the number of elements between &p[4] and &p[0]  */
y = r □ q;        /*  Error !!    r and q are pointers to different arrays  */
```

2.7.3 Sample Code with Pointer Errors

We now provide an example program, pointerBugs.c, written in C that

1. constructs an array of 10 integers
2. sorts the array in ascending or descending order (this example sorts the array in ascending order)
3. calculates the average value of the 10 elements in the array
4. computes the frequency of occurrence of each different integer value in the array
5. checks the results of the frequency computation by calculating the sum of the individual frequencies (which must add up to 1.0)

Our sample code contains a number of bugs resulting from the incorrect use of pointers. It was purposely written so that it deviates from traditional programming practice by using pointer notation in place of array indexing when referring to the individual elements in the array. For example, **(a + k)* is used in place of *[k]* when referring to the value of the *(k+1)*st element (k^{th} index) of the array, *a[ ]*. Similarly, *(a + k)* is used in place of *&a[k]* when referring to the address of the *(k+1)*st element (k^{th} index) of *a[ ]*. This is done in order to illustrate some of the types of pointer errors that you can easily make when using pointer notation to manipulate arrays.These types of coding errors can often be avoided simply by using array indexing, which is a primary reason why using array indexing to manipulate arrays is the preferred programming practice. If this code had been written correctly, it would have generated the following (correct) output.

```
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]
The sequence of elements in the sorted array are: [ 0 1 2 4 5 6 8 9 13 17 ]
The average value of the array elements is: 6.500000
The frequencies of the elements in the sorted array are: [ 0.100000 0.100000 0.100000 0.100000 0.100000
0.100000 0.100000 0.100000 0.100000 0.100000 ]
The sum of the element frequencies is: 1.000000
```

2.7.4 Debugging the Sample Code

2.7.4.1 Debugging the Sample Code

We will demonstrate our process for debugging this sample code in the following sections.

2.7.4.2 Error 1: Uninitialized Pointer

We now compile the pointerBugs.c code and run it on Linux to see what results it generates. We compile it using gcc and include the '□g' option since we know that our program requires debugging.

```
$ gcc -g -o pointerBugs pointerBugs.c
pointerBugs.c: In function `main':
pointerBugs.c:48: warning: passing arg 3 of `sortArray' from incompatible pointer type
```

We find that the code does compile, but the compiler has issued a warning about an argument being passed to one of our functions, sortArray(), from an incompatible pointer type. For now, we ignore this warning and run the compiled code to see what happens.

```
$ ./pointerBugs
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]

Segmentation fault
```

The segmentation fault tells us that there is definitely a bug somewhere in our code. Based on the warning message from the compiler,we might guess that the bug is in the function sortArray(). To identify the exact nature of this bug, we rerun the code in GDB.

```
$ gdb pointerBugs
(gdb) run
Starting program: pointerBugs
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]

Program received signal SIGSEGV, Segmentation fault.
  0x080486d6 in sortArray(nelem_in_array=10, array=0x8049c80, order=0x8048734

  <ascending>) at pointerBugs.c:114
```

```
114           (!*order(sflag,(array + indx) , (array + (indx + 1))))
```

As we suspected, the bug occurred in the function sortArray() and appears to be some type of error related to the expression

```
(!*order(sflag, (array + indx) , (array + (indx + 1))))
```

at line 114.

The first thing we do is set a breakpoint at line 114 and rerun our code.

```
(gdb) b 114
Breakpoint 1 at 0x80486aa: file pointerBugs.c, line 114.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: pointerBugs
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]

Breakpoint 1, sortArray (nelem_in_array=10, array=0x8049c80, order=0x8048734 <ascending>)
  at pointerBugs.c:114
114           if (!*order(sflag,
      (array + indx) , (array + (indx + 1))))
```

Since the error appears to originate from the if statement at line 114 in sortArray(), we print the value of the expression in parentheses to see what it is.

```
(gdb) p *order(sflag, (array + indx) , (array + (indx + 1)))
Cannot access memory at address 0x1
```

This output is odd since the expression in the if statement should be a numerical value. If

```
order(sflag, (array + indx) , (array + (indx + 1)))
```

is a pointer to a number, then the *dereferenced* pointer

```
*order(sflag, (array + indx) , (array + (indx + 1)))
```

should be a number unless the pointer, *order()*, was improperly initialized. Consequently, this looks suspiciously like an uninitialized pointer error.

If we look at the code we can see from our call to sortArray() in *main()*

```
sortArray(nelem, array, ascending);
```

that that we are passing the address of the function *ascending* to sortArray() (recall that the address of a function is just the name of the function without parentheses). We also see from our definition of sortArray(),

```
void sortArray(const int nelem_in_array, int *array, int *order(int, int *, int *)),
```

that the function pointer *order()* is a pointer used to call the function *ascending()*

```
int ascending(int sflag, int *n, int *m)
```

that returns an integer value to the calling function sortArray(). Therefore, the argument 3 of sortArray(),

```
int *order(int, int *, int *)
```

is supposed to be a pointer to a function that returns an *int*. Note that this is exactly the argument that the compiler warned us about when we first compiled pointerBug.c.

If *order()* is supposed to be pointing to the function *ascending()*, then either *ascending()* must be located at the inaccessible memory address 0x1 or *order()* is pointing to the wrong location in memory. To identify the correct scenario, we print the address of the function *ascending()*. (Recall that the address of a function is just the name of the function without parentheses.)

```
(gdb) p ascending
```

```
 = {int (int, int, int)} 0x8048734 <ascending>
```

This is consistent with the information:

```
Breakpoint 1, sortArray (nelem_in_array=10, array=0x8049c80, order=0x8048734 <ascending>)
   at pointerBugs.c:114
114           if (!*order(sflag, array + indx) , (array + (indx + 1))))
```

Clearly, *order()* is not pointing to *ascending()* since order is pointing to the address **0x1**

```
(gdb) p order(sflag, (array + indx) , (array + (indx + 1)))
(gdb) p order(sflag, (array + indx) , (array + (indx + 1))) $2 = (int *) 0x1 = (int *) 0x1
```

whereas *ascending()* is located at the address '0x8048734'. Therefore, we know that the segmentation fault in our program is caused by a pointer that is not pointing where it is supposed to be pointing. In other words, we are using a pointer that has not been initialized correctly.

If we look more carefully at how the function sortArray() was defined, we see that the error is in the third argument that is being passed to the function:

```
void sortArray(const
      int nelem_in_array, int *array, int *order(int, int *, int *))
```

The argument that is being used in this function:

```
int *order(int, int *, int *)  =  int * (order(int, int *, int *))
```

is actually a function, order(int, int *, int *), that returns a pointer to an integer, *int*. It is not a pointer to a function that returns an integer. We have simpl defined our function pointer incorrectly. We can see what the correct form of this function pointer should be by looking at the address of the pointer that is actually pointing to the function ascending().

```
(gdb) p &ascending
 = (int (*)(int, int, int)) 0x8048734 <ascending>
```

We can see that to correctly define a pointer to a function, we must place parentheses around the pointer name as in:

```
int (*order)(int, int *, int *)
```

We will make this correction in each of three locations within our code:

- one correction in the function header for sortArray() prior to main() `void sortArray(int nelem_in_array, int *array, int (*fcn)(int sortFlag, int *n, int *m));`
- two corrections in the definition of the function sortArray() `void sortArray(const int nelem_in_array, int *array, int (*order)(int, int *, int *))`

```
{
 for (i = 0; i < nelem_in_array; i++)
 {
  for (indx = 0; indx < (nelem_in_array - 1); indx++)
  {
   sflag = 0;

  if (! (*order)(sflag, (array + indx) , (array + (indx + 1))))
  {
   exchange((array + indx), (array + (indx + 1)));
   }
  }
 }
}
```

2.7.4.3 Error 2: Another Uninitialized Pointer

Now that we have made these corrections to our code, we will recompile it and rerun it in GDB. We edit and recompile our code in a separate terminal window so we do not have to restart GDB each time we modify our code.

```
$ gcc -g -o pointerBugs pointerBugs.c
pointerBugs.c: In function `main':
```

As you can see, our corrections have eliminated the compiler warning regarding an incompatible pointer type.

```
(gdb) run
Starting program: pointerBugs
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]
The sequence of elements in the sorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]

The average value of the array elements is: 6.500000

The frequencies of the elements in the sorted array are: [ 0.100000 0.100000 0.100000 0.100000 0.100000
0.100000
      0.100000 0.100000 0.100000 0.000000 ]

Program received signal SIGSEGV, Segmentation fault.
0x080485d6 in main () at pointerBugs.c:76
76        *pfsum = *fqsum(nelem,freq);
```

Now that we have corrected the error in our function pointer in *sortArray( ),* we find that we have another error in our code. This error occurs at line 76 in the statement

```
*pfsum = *fqsum(nelem, freq);
```

We delete our previous breakpoint at line 114 and set a new breakpoint at line 76. Then we rerun our code in GDB.

```
(gdb) clear 114
Deleted breakpoint 1
(gdb) b 76
Breakpoint 2 at 0x80485c0: file pointerBugs.c, line 76.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: pointerBugs
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]
The sequence of elements in the sorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]
The average value of the array elements is: 6.500000
The frequencies of the elements in the sorted array are: [ 0.100000 0.100000 0.100000 0.100000 0.100000
0.100000
      0.100000 0.100000 0.100000 0.000000 ]
Breakpoint 2, main () at pointerBugs.c:76
76        *pfsum = *fqsum(nelem, freq);
```

The function fqsum() shown below calculates the sum of the frequencies of the individual elements in our array of ten integers.

```
float *fqsum(int nelem_in_array, float *freq)
{
 float *fsum = (float *)malloc(sizeof(float));
 *fsum = 0.0;

 for (indx = 0; indx < nelem_in_array; indx++)
 {
  *fsum += freq[indx];
 }

 return fsum;
}
```

We step into this function from line 76 and examine its output during during each of the nelem = 10 iterations of the for loop.

```
(gdb) step
fqsum (nelem_in_array=10, freq=0x8049cb0) at pointerBugs.c:192
192        float *fsum = (float *)malloc(sizeof(float));
(gdb) n
193       *fsum = 0.0;
(gdb) n
```

```
195       for (indx = 0; indx < nelem_in_array; indx++)
(gdb) disp *fsum
2: *fsum = 0
(gdb) n
197         *fsum += freq[indx];
2: *fsum = 0
(gdb) n
195       for (indx = 0; indx < nelem_in_array; indx++)
2: *fsum = 0.100000001
(gdb) n
197         *fsum += freq[indx];
2: *fsum = 0.100000001
(gdb) n
195       for (indx = 0; indx < nelem_in_array; indx++)
2: *fsum = 0.200000003
(gdb) n
197         *fsum += freq[indx];
2: *fsum = 0.200000003
(gdb) n
195       for (indx = 0; indx < nelem_in_array; indx++)
2: *fsum = 0.300000012
(gdb) n
197         *fsum += freq[indx];
2: *fsum = 0.300000012
(gdb) n
195       for (indx = 0; indx < nelem_in_array; indx++)
2: *fsum = 0.400000006
(gdb) n
197         *fsum += freq[indx];
2: *fsum = 0.400000006
(gdb) n
195       for (indx = 0; indx < nelem_in_array; indx++)
2: *fsum = 0.5
(gdb) n
197         *fsum += freq[indx];
2: *fsum = 0.5
(gdb) n
195       for (indx = 0; indx < nelem_in_array; indx++)
2: *fsum = 0.600000024
(gdb) n
197         *fsum += freq[indx];
2: *fsum = 0.600000024
(gdb) n
195       for (indx = 0; indx < nelem_in_array; indx++)
2: *fsum = 0.700000048
(gdb) n
197         *fsum += freq[indx];
2: *fsum = 0.700000048
(gdb) n
195       for (indx = 0; indx < nelem_in_array; indx++)
2: *fsum = 0.800000072
(gdb) n
197         *fsum += freq[indx];
2: *fsum = 0.800000072
(gdb) n
195       for (indx = 0; indx < nelem_in_array; indx++)
2: *fsum = 0.900000095
(gdb) n
197         *fsum += freq[indx];
2: *fsum = 0.900000095
(gdb) n
195       for (indx = 0; indx < nelem_in_array; indx++)
2: *fsum = 0.900000095
(gdb) n
200       return fsum;
2: *fsum = 0.900000095
(gdb) n
201     }
2: *fsum = 0.900000095
```

```
(gdb) n

Program received signal SIGSEGV, Segmentation fault.
  0x080485d6 in main () at pointerBugs.c:76
76        *pfsum = *fqsum(nelem,
      freq);
```

Clearly, the segmentation fault does not occur during any of the ten separate computations carried out by fqsum( ). Instead, the segmentation fault appears to occur immediately upon exit from this function.

At line 76, we print both the value and the address of fqsum( ).

```
(gdb) p *fqsum(nelem, freq)
 = 0.900000095
(gdb) p fqsum(nelem, freq)
 = (float *) 0x8049d30
```

So far, everything looks perfectly fine. Now, we print the value and the address of the pointer *pfsum*.

```
(gdb) p *pfsum
Cannot access memory at address 0x0
(gdb) p pfsum
 = (float *) 0x0
```

This should look vaguely familiar. It appears that, once again, we are attempting to use an uninitialized pointer, *pfsum*.

In our code, we have declared the pointer *pfsum*

float *pfsum;

and then we have set its value equal to the value returned by the function fqsum( ),

*pfsum = *fqsum(nelem, freq);

However, since we failed to initialize *pfsum* anywhere in our code, we now have a pointer that points to an unknown memory address.Consequently, when we attempt to dereference this pointer in our printf statement,

```
printf("The sum of the element frequencies is: %f\n\n", *pfsum);
```

we obtain a segmentation fault since the address is unknown and is therefore inaccessible.

We can correct our code in either of two ways:

1. redefine the pointer *pfsum* to be a float value, *fsum*, `float fsum;` set `fsum = *fqsum(nelem, freq);` and change the printf statement to read: `printf("The sum of the element frequencies is: %f\n\n", fsum);`
2. initialize the pointer, *pfsum*, to point to the address of the function fqsum( ), `fsum = fqsum(nelem, freq);`

Since our topic is the correct use of pointers, we use the latter correction to our code.

```
/* Compute the sum of the frequencies of the array elements  */
 pfsum = fqsum(nelem, freq);

/* Print the sum of the frequencies of the array elements  */

printf("The sum of the element frequencies is: %f\n\n", *pfsum);
```

2.7.4.4 Error 3: Passing in Pointers to Integers

Now that we have corrected our code, we recompile it and rerun our program:

```
$ gcc -g -o pointerBugs pointerBugs.c
pointerBugs.c: In function `main':
```

```
(gdb) run
Starting program: pointerBugs
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]

The sequence of elements in the sorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]

The average value of the array elements is: 6.500000

The frequencies of the elements in the sorted array are: [ 0.100000 0.100000 0.100000 0.100000 0.100000
0.100000
      0.100000 0.100000 0.100000 0.000000 ]

Breakpoint 2, main () at pointerBugs.c:76
76        pfsum = fqsum(nelem, freq);
(gdb) c
Continuing.
The sum of the element frequencies is: 0.900000

Program exited normally.
```

The program completes successfully but there appears to be several problems with the results. The sorted array is *not* sorted and the frequencies do not sum to 1.0.

Since we have eliminated the segmentation fault at line 76, we delete our line 76 breakpoint and set a new breakpoint at main().

```
(gdb) clear 76
Deleted breakpoint 2
(gdb) b main
Breakpoint 3 at 0x80483a0: file pointerBugs.c, line 19.
(gdb) run
Starting program: pointerBugs

Breakpoint 3, main () at pointerBugs.c:19
19        const int nelem = 10;
```

Now we list the lines in our source code to determine where we want to set our next breakpoint.

```
(gdb) l
  14      float *computeFreq(int
      nelem_in_array, int *array, float *freq);
  15      float *fqsum(int nelem_in_array,
      float *freq);
  16
  17      int main(void)
  18      {
  19        const int nelem =
      10;
  20
  21        int *array;
  22        float *freq;
  23        float *pfsum = 0;
  (gdb)
  24
  25        array = (int *)malloc(nelem*sizeof(int));
  26        freq = (float *)malloc(nelem*sizeof(float));
  27
  28      /*  Initialize the arrays  */
  29        for (indx = 0; indx < nelem;
      indx++)
  30        {
  31          array[indx] = 0;
  32          freq[indx] = 0.0;
  33        }
  (gdb)
  34
  35      /* Replace initial values
      of array elements  */
  36        populateArray(nelem,
```

```
    array);
 37
 38      /* Print the elements
    of the unsorted array  */
 39        printf("The sequence
    of elements in the unsorted array are: [ ");
 40
 41        for (indx = 0; indx < nelem;
    indx++)
 42        {
 43          printf("%d ",
    array[indx]);
 (gdb)
 44        }
 45        printf("]\n\n");
 46
 47      /* Sort the array of
    integers  */
 48        sortArray(nelem,
      array, ascending);
 49
 50      /* Print the elements
    of the sorted array  */
 51        printf("The sequence
    of elements in the sorted array are: [ ");
 52        for (indx = 0; indx < nelem;
    indx++)
 53        {
```

Since our unsorted array is not being sorted when the function sortArray() is called, we set a breakpoint at line 48 and step into this function:

```
(gdb) clear main
  Deleted breakpoint 3
  (gdb) b 48
  Breakpoint 4 at 0x8048493: file
      pointerBugs.c, line 48.
  (gdb) run
  The program being debugged has
      been started already.
  Start it from the beginning?
      (y or n) y
  Starting program: pointerBugs
  The sequence of elements in
      the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]
  Breakpoint 4, main () at pointerBugs.c:48
  48        sortArray(nelem, array,
      ascending);
  (gdb) step
  sortArray (nelem_in_array=10,
      array=0x8049c80, order=0x804872b <ascending>)
      at pointerBugs.c:108
  108       for (i = 0; i < nelem_in_array;
      i++)
  (gdb) step
  110         for (indx = 0; indx < (nelem_in_array
      - 1); indx++)
  (gdb) step
  112           sflag = 0;
  (gdb) step
  114           if (!(*order)(sflag,
      (array + indx) , (array + (indx + 1))))
  (gdb) step
  ascending (sflag=0, n=0x8049c80,
      m=0x8049c84) at pointerBugs.c:133
  133       sflag = 0;
```

Now that we have reached the point in sortArray() where the function ascending() in line 133 is called, we step into this function and look at its output each time it is called during the first (nelem 1) = 9 iterations of the for loop in line 110:

```
for (indx = 0; indx < (nelem_in_array - 1); indx++)
```

We know from our previous look at the function sortArray() that ascending() returns a value of zero if the value of a given array element is less than the value of the preceding element, indicating that sorting is necessary:

```
(*order)(sflag, (array + indx) , (array + (indx + 1))) = 0
```

This makes the if statement in line 114 *true*:

```
if (!(*order)(sflag, (array + indx) , (array + (indx + 1))))
```

then calls the function exchange( ) to exchange the values of the two sequential array elements. Since no exchange of elements is taking place, then either the unsorted array is already in ascending order, the function exchange( ) is simply never called due to some coding error, or exchange() is called but fails to exchange the values of the sequential array elements.

Obviously, the integers in our unsorted array:

```
array  = [ 1  0  5  2  9  4  13  6  17  8 ]
```

are not already in ascending order. Therefore, either the function exchange() is never called or it does not work properly when it is called.

We see from looking at our unsorted array that exchange() should be called five separate times during the first pass through the array (the first nine iterations of the inner for loop in our sortArray() function).

To see if exchange() is being called and, if not, whether the problem is in the if statement that calls this function, we look for a call to exchange() and also print the values of both *sflag* and *(*order)(sflag,(array + indx) , (array + (indx + 1)))* as we step through each of the first nine iterations of inner for loop of sortArray(),

```
for indx = 0; indx < (nelem_in_array - 1); indx++ )
```

We also print out the value of *indx* so that we know when we have completed nine iterations of this for loop.

We can use the 'display' (or 'disp') command to print out the values for the output every time the program pauses. Note that only the output for iterations:

- 1 (indx = 0)
- 2 (indx = 1)
- 9 (indx = 8)

is shown below since each of the six iterations produces identical output.

```
(gdb) disp indx
1: indx = 0
(gdb) disp sflag
2: sflag = 0
(gdb) disp (*order)(sflag,
      (array + indx) , (array + (indx + 1)))
3: (*order) (sflag, array +
    indx, array + (indx + 1)) = 1
(gdb) step
ascending (sflag=0, n=0x8049c80,
    m=0x8049c84) at pointerBugs.c:133
133       sflag = 0;
2: sflag = 0
1: indx = 0
(gdb) step
135       if (m > n)
2: sflag = 0
1: indx = 0
(gdb) step
137        sflag = 1;
2: sflag = 0
1: indx = 0
(gdb) step
139       return sflag;
2: sflag = 0
```

```
1: indx = 0
(gdb) step
140     }
2: sflag = 0
1: indx = 0
(gdb) step
sortArray (nelem_in_array=10,
    array=0x8049c80, order=0x804872b <ascending>)
   at pointerBugs.c:110
110         for (indx = 0; indx < (nelem_in_array
    - 1); indx++)
3: (*order) (sflag, array +
    indx, array + (indx + 1)) = 1
2: sflag = 0
1: indx = 0
(gdb) step
112           sflag = 0;
3: (*order) (sflag, array +
    indx, array + (indx + 1)) = 1
2: sflag = 0
1: indx = 1
(gdb) step
114           if (!(*order)(sflag,
    (array + indx) , (array + (indx + 1))))
3: (*order) (sflag, array +
    indx, array + (indx + 1)) = 1
2: sflag = 0
1: indx = 1
(gdb) step
ascending (sflag=0, n=0x8049c84,
    m=0x8049c88) at pointerBugs.c:133
133       sflag = 0;
2: sflag = 0
1: indx = 1
(gdb) step
135       if (m > n)
2: sflag = 0
1: indx = 1
(gdb) step
137        sflag = 1;
2: sflag = 0
1: indx = 1
(gdb) step
139       return sflag;
2: sflag = 0
1: indx = 1
(gdb) step
140     }
2: sflag = 0
1: indx = 1
(gdb) step
sortArray (nelem_in_array=10,
    array=0x8049c80, order=0x804872b <ascending>)
   at pointerBugs.c:110
110         for (indx = 0; indx < (nelem_in_array
    - 1); indx++)
3: (*order) (sflag, array +
    indx, array + (indx + 1)) = 1
2: sflag = 0
1: indx = 1
(gdb) step
112           sflag = 0;
3: (*order) (sflag, array +
    indx, array + (indx + 1)) = 1
2: sflag = 0
1: indx = 2
.     .     .     .     .     .
.     .     .     .     .     .
(gdb) step
```

```
112           sflag = 0;
3: (*order) (sflag, array +
    indx, array + (indx + 1)) = 1
2: sflag = 0
1: indx = 8
(gdb) step
114           if (!(*order)(sflag,
    (array + indx) , (array + (indx + 1))))
3: (*order) (sflag, array +
    indx, array + (indx + 1)) = 1
2: sflag = 0
1: indx = 8
(gdb) step
ascending (sflag=0, n=0x8049ca0,
    m=0x8049ca4) at pointerBugs.c:133
133       sflag = 0;
2: sflag = 0
1: indx = 8
(gdb) step
135       if (m > n)
2: sflag = 0
1: indx = 8
(gdb) step
137        sflag = 1;
2: sflag = 0
1: indx = 8
(gdb) step
139       return sflag;
2: sflag = 0
1: indx = 8
(gdb) step
140     }
2: sflag = 0
1: indx = 8
(gdb) step
sortArray (nelem_in_array=10,
    array=0x8049c80, order=0x804872b <ascending>)
   at pointerBugs.c:110
110         for (indx = 0; indx < (nelem_in_array
    - 1); indx++)
3: (*order) (sflag, array +
    indx, array + (indx + 1)) = 1
2: sflag = 0
1: indx = 8
(gdb) step
```

We can clearly see that our function exchange() is never called during any of these nine iterations of the for loop in line 110. We can also see that for each of the nine individual iterations, the value returned from the function ascending() is always 1:

```
(*order) (sflag, array + indx, array + (indx + 1)) = 1
```

Consequently, we can see why exchange( ) is never called. Since ascending() always returns the integer 1, then the if statement in line 114:

```
if (!(*order)(sflag, (array + indx) , (array + (indx + 1))))
```

is always *false,* which tells us that the if statement simply never triggers a call to exchange().

To see if we can find out why ascending() always returns the value 1 and never returns the value 0 for an array that is not already in ascending order, we set a new breakpoint at the beginning of this function in line 133 and rerun our code.

```
(gdb) undisp
 Delete all auto-display expressions?(y or n) y
(gdb) clear 48
Deleted breakpoint 4
(gdb) b 133
Breakpoint 5 at 0x804872e: file pointerBugs.c, line 133.
(gdb) run
```

```
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: pointerBugs
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]
Breakpoint 5, ascending (sflag=0, n=0x8049c80, m=0x8049c84) at pointerBugs.c:133
133       sflag = 0;
```

As we see from the definition of our function ascending(), this function should return an integer of 0 when

```
array[j+1] > array[j]
```

and should return an integer of 1 (one) when

```
array[j+1] < array[j]
```

Note here that we are using array indexing instead of pointer notation.

To see if we can figure out why ascending() only returns the integer 1 and never returns 0, we move one line at a time through this function and compare the actual values used in the inequality expression, $m > n$.

```
(gdb) n
  135       if (m > n)
  (gdb) p m
   = (int *) 0x8049c84
  (gdb) p n
  (gdb) n 135 if (m > n) (gdb) p m $1 = (int *) 0x8049c84 (gdb) p n $2 = (int *) 0x8049c80 (gdb) n 137 sflag = 1;
(gdb) n 139 return sflag; (gdb) n 140 } (gdb) n sortArray (nelem_in_array=10, array=0x8049c80, order=0x804872b ) at
pointerBugs.c:110 110 for (indx = 0; indx < (nelem_in_array - 1); indx++) (gdb) n 112 sflag = 0; (gdb) n 114 if
(!(*order)(sflag, (array + indx) , (array + (indx + 1)))) (gdb) n Breakpoint 5, ascending (sflag=0, n=0x8049c84,
m=0x8049c88) at pointerBugs.c:133 133 sflag = 0; (gdb) n 135 if (m > n) (gdb) p m $3 = (int *) 0x8049c88 (gdb) p n
$4 = (int *) 0x8049c84 = (int *) 0x8049c80
  (gdb) n
  137        sflag = 1;
  (gdb) n
  139       return sflag;
  (gdb) n
  140     }
  (gdb) n
  sortArray (nelem_in_array=10, array=0x8049c80, order=0x804872b
)
    at pointerBugs.c:110
110         for (indx = 0; indx (gdb) n
112           sflag = 0;
(gdb) n
114           if (!(*order)(sflag, (array + indx) , (array + (indx + 1))))
(gdb) n
Breakpoint 5, ascending (sflag=0, n=0x8049c84, m=0x8049c88) at pointerBugs.c:133
133       sflag = 0;
(gdb) n
135       if (m > n)
 (gdb) p m
 = (int *) 0x8049c88
(gdb) p n
 = (int *) 0x8049c84
```

We can now see exactly what the problem is in the function ascending(). Our definition for ascending() outside of main(),

```
int ascending(int sflag, int n, int m)
{
   sflag = 0;

   if ( m > n )
   {
     sflag = 1;
   }

   return sflag;
}
```

tells us that we are passing in three integer arguments; ascending() then sets the value of the first integer argument to 0 and then compares the numerical values of the remaining two integer arguments. However, it is clear from our printout of the values for the arguments *m* and *n* that they are not integer values at all but, instead, are memory addresses.

```
(gdb) p m
 = (int *) 0x8049c84
(gdb) p n
(gdb) p m $1 = (int *) 0x8049c84 (gdb) p n $2 = (int *) 0x8049c80 . . . (gdb) p m $3 = (int *) 0x8049c88 (gdb) p n
$4 = (int *) 0x8049c84 = (int *) 0x8049c80
.   .   .

(gdb) p m
 = (int *) 0x8049c88
(gdb) p n
 = (int *) 0x8049c84
```

Actually, we can obtain this same information without even printing out the values of *n* and *m*. If we look carefully at the function ascending() each time the program pauses at breakpoint 5 on line 133, we see that ascending() tells us that it is passing in one parameter with an integer value,

```
Breakpoint 5, ascending (sflag=0, n=0x8049c80, m=0x8049c84) at pointerBugs.c:133  ,
Breakpoint 5, ascending (sflag=0, n=0x8049c84, m=0x8049c88) at pointerBugs.c:133
```

and two parameters whose values are memory addresses,

```
Breakpoint 5, ascending (sflag=0, n=0x8049c80, m=0x8049c84) at pointerBugs.c:133
Breakpoint 5, ascending (sflag=0, n=0x8049c84, m=0x8049c88) at pointerBugs.c:133
```

These are exactly the same results we obtained when we asked GDB to print out the values of *n* and *m*.

As you can see *n* and *m* are always separated by 4 bytes (= *sizeof(int)*), which is the separation of adjacent elements in *array[ ]*:

```
Breakpoint 5, ascending (sflag=0, n=0x8049c80,     m=0x8049c84) at pointerBugs.c:133  ,
                                                 &array[ j ]       &array[ j+1 ]

Breakpoint 5, ascending (sflag=0, n=0x8049c84,      m=0x8049c88) at pointerBugs.c:133
                                                 &array[ j+1 ]     &array[ j+2 ]
```

Consequently, ascending() is actually comparing the value of the two addresses for *n* and *m*. Obviously, instead of passing in two integers as arguments 2 and 3 of ascending(), we are mistakenly passing in pointers to integers, **n* and **m*. Therefore, *n* actually represents the address of the integer array element *j*, *&array[j]*, whereas *m* represents the address of the integer array element *(j + 1)*, *&array[j+1]*. In this case,

```
m > n
```

will always be true because element *array[j+1]* always follows *array[j]* within the block of contiguous memory allocated for *array[ ]* (i.e., *&array[j+1]* will always be located downstream from *&array[j]*). If we look at the function prototype for both ascending() and descending() prior to main():

```
int ascending( int,  int *n,  int *m);
int descending( int,  int *n,  int *m);
```

and at the way in which these two functions are defined outside of main():

```
int ascending(int sflag, int *n, int *m)
int descending(int sflag, int *n, int *m)
```

To correct this code, we need to replace the two pointer-to-integer arguments with integer arguments:

- in the function prototypes of ascending() and descending() prior to main()
  ```
  int ascending( int,  int n,  int m);
  int descending( int,  int n,  int m);
  ```
- in the definitions of both ascending() and descending() following main()
  ```
  int ascending(int sflag, int n, int m)
  {
      .  .  .
  }
  ```

```
int descending(int sflag, int n, int m)
{
    .  .  .
}
```

- in the function pointer that points to the function ascending() that is passed as the third argument to our function sortArray() both in the prototype for sortArray( ) and in the definition of this function
```
void sortArray(int nelem_in_array, int *array, int (*fcn)(int sortFlag, int n, int m));

void sortArray(const int nelem_in_array, int *array, int (*order)(int, int, int))
{
    .  .  .
}
```
- in the call to ascending() in sortArray() through the pointer **order( )* in the if statement in line 114 inside main() `if (!*order(sflag, *(array + indx) , *(array + (indx + 1))))` Note that in this last correction we are changing the address of the array elements, *(array + indx) = &array[indx]* to the value of array element at that same index, **(array + indx) = array[indx]*.

2.7.4.5 Error 4: Different Pointers Pointing to the Same Address

After making these corrections, we will recompile our code and rerun it in GDB.

```
(gdb) clear 133
Deleted breakpoint 5
 (gdb) run
Starting program: pointerBugs
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]

The sequence of elements in the sorted array are: [ 0 0 2 2 4 4 6 6 8 8 ]

The average value of the array elements is: 4.000000

The frequencies of the elements in the sorted array are: [ 0.300000 0.300000 0.300000 0.300000 0.100000
0.000000 0.000000 0.000000 0.000000 0.000000 ]

The sum of the element frequencies is: 1.300000

Program exited normally.
```

We can see that our sortArray() function is now working - at least it is doing something - but it appears to be doing something very strange. In other words, sortArray() is working, but it is not working correctly. The elements are certainly in ascending order, but the value at each odd-numbered index is being repeated in pairs:

```
1, 0   ®   0, 0
5, 2   ®   2, 2
    .
    .
    .
17, 8   ®   8, 8
```

It looks suspiciously like sortArray() is attempting to sort adjacent elements through the function ascending(), but that sorting is never quite completed with the result that the value of every odd-index element is getting copied to the previous even-index elements.

Let us go back to the beginning of sortArray() at line 110 and list the lines of code in this function to see where we might want to set our next breakpoint.

```
(gdb) b 110
Breakpoint 6 at 0x8048680: file pointerBugs.c, line 110.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: pointerBugs
```

```
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]

Breakpoint 6, sortArray (nelem_in_array=10, array=0x8049c80, order=0x8048732 )
    at pointerBugs.c:110
110         for (indx = 0; indx < (nelem_in_array - 1); indx++)
(gdb) l
105
106     void sortArray(const int nelem_in_array, int *array, int (*order)(int, int, int))
107     {
108       for (i = 0; i < nelem_in_array; i++)
109       {
110         for (indx = 0; indx < (nelem_in_array - 1); indx++)
111         {
112           sflag = 0;
113
114           if (!(*order)(sflag, *(array + indx) , *(array + (indx + 1))))
(gdb)
115           {
116             exchange((array + indx), (array + (indx + 1)));
117           }
118         }
119       }
120     }
121
122     void exchange(int *elem1, int *elem2)
123     {
124       int *tmp;
(gdb)
125
126       tmp = elem1;
127       *elem1 = *elem2;
128       elem2 = tmp;
129     }
130
```

Since it appears that sortArray() is making some type of aberrant attempt to exchange the order of the integers in our array, we check to see if exchange() is being called when

```
if (!(*order)(sflag, *(array + indx) , *(array + (indx + 1))))
```

is true. We set a new breakpoint at line 114, step into the function ascending(), and then see if we are able to step into the function exchange(). If we are able to step into ascending(), then we will continue to step through this function and examine its output to see if we can determine where our error is occurring.

```
(gdb) clear 110
Deleted breakpoint 6
(gdb) b 114
Breakpoint 7 at 0x80486a2: file pointerBugs.c, line 114.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: pointerBugs
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]

Breakpoint 7, sortArray (nelem_in_array=10, array=0x8049c80,
    order=0x8048732 ) at pointerBugs.c:114
114           if (!(*order)(sflag, *(array + indx) , *(array + (indx + 1))))
(gdb) p (*order)(sflag, *(array + indx) , *(array + (indx + 1)))
 = 0
```

Note that after making our previous code correction, (*order)(sflag, *(array + indx) , *(array + (indx + 1))) = 0 as it should since *array[1] < array[0]*.

```
(gdb) step
ascending (sflag=0, n=1, m=0) at pointerBugs.c:133
133       sflag = 0;
(gdb) step
135       if (m > n)
```

```
(gdb) p m
 = 0
(gdb) p n
 = 1
```

Note also that after making our previous code correction, the parameters whose values are being compared are integers and not addresses (pointers).

```
(gdb) step
139       return sflag;
(gdb) step
140     }
(gdb) step
sortArray (nelem_in_array=10, array=0x8049c80, order=0x8048732 )
    at pointerBugs.c:116
116             exchange((array + indx), (array + (indx + 1)));
```

As we can see after making our previous code correction, exchange() is now being called from within sortArray(). Since everything appears to be working properly up to this point, we might guess that the bug is in the function exchange() itself.

```
(gdb) step
exchange (elem1=0x8049c80, elem2=0x8049c84) at pointerBugs.c:126
126       tmp = elem1;
(gdb) step
127       *elem1 = *elem2;
(gdb) step
128       elem2 = tmp;
(gdb) step
129     }
```

Let's print the values of each of these variables to see if we can determine exactly what is going wrong within this function.

```
(gdb) p tmp
 = (int *) 0x8049c80
(gdb) p elem1
 = (int *) 0x8049c80
(gdb) p *elem1
 = 0
(gdb) p *elem2
 = 0
(gdb) p elem2
 = (int *) 0x8049c80
```

Now it is clear what the problem is. In line 126 we initialize the pointer to point to the address of one of the elements in our array. In this case *tmp* is set to point to the first element, *array[0]* whose address is *&array[0],*

```
(gdb) p tmp
 = (int *) 0x8049c80
(gdb) p elem1
 = (int *) 0x8049c80
```

Then in line 127 we set the value of first array element, *array[0]*, equal to the value of the second array element , *array[1]* which gives us array[0] = array[1] = 0 (since 0 is the value of the second element in our array),

```
(gdb) p *elem1
 = 0
(gdb) p *elem2
 = 0
```

Finally, for some unknown reason, in line 128 we set the pointer to the second array element to point to the first array element. The variable *elem2* points to the address pointed to by *tmp*, which is the same as the address of *elem1*, 0x8049c80,

```
(gdb) p elem2
 = (int *) 0x8049c80
```

Now we have two different pointers pointing to the same address array element and the data value at that address is 0. In essence, we have simply overwritten the value of the first array element with the value of the second array element.It is

quite likely that we have also really messed up our array since we are changing the addresses of contiguous elements within a single block of memory. On some systems this type of error might well generate a segmentation fault instead of just generating a really odd looking sequence of array elements. Nevertheless, when we print the values of the elements in our sorted array, we simply get pairs of repeated values. The basis of our problem is that we are not just exchanging the values of our array elements; we are also exchanging addresses which we absolutely do not want to be doing. Clearly, if we want to exchange values instead of addresses, we should be using integer values instead of addresses within the entire body of the function exchange(). To correct this error, we need to replace all pointer variables with integer variables in the definition of the function exchange() following main(),

```
void exchange(int elem1, int elem2)
{
  int tmp;       /*  declare an integer (not a pointer) to temporarily hold the 'value' (*elem1)  */

  tmp = *elem1;              /*    set the value of tmp equal to the integer array[k]                */
  *elem1 = *elem2;        /*    set the 'value' at  address &array[k ] equal to array[k + 1]   */
  *elem2 = tmp;             /*    set the 'value' at address &array[k + 1] equal to array[k]     */
}
```

2.7.4.6 Error 5: Missing Assignment Statement

Now that we have made the correction to our code from Error 4, we recompile and rerun our program.

```
$ gcc -g -o pointerBugs pointerBugs.c
pointerBugs.c: In function `main':
(gdb) run
Starting program: pointerBugs
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]

The sequence of elements in the sorted array are: [ 0 1 2 4 5 6 8 9 13 17 ]

The average value of the array elements is: 6.500000

The frequencies of the elements in the sorted array are: [ 0.100000 0.100000 0.100000 0.100000 0.100000
0.100000 0.100000 0.100000 0.100000 0.000000 ]

The sum of the element frequencies is: 0.900000

Program exited normally.
```

Now we see that the array has been sorted in ascending order and that it contains each of the elements from the unsorted array with no duplications or deletions. However, we still have the problem of the frequencies of the individual array elements not summing to 1.0. If we look at our frequency array, we see why. All of the first nine elements are 0.1 and these nine frequencies sum to 0.9. The tenth frequency is 0.0, which explains why the total of the ten frequencies is 0.9. The tenth element in the frequency array is simply missing.

We might guess that our next error is somewhere in the function that computes the frequency of occurrence for each different integer value in our array. Therefore, we take a look at each of the functions involved in this computation. To see where we want to place a new breakpoint, we list the lines in our source code that follow the call to sortArray() on line 48. (We start at line 48 because we already listed the lines above 48 and found that there were no functions relating to frequency in that part of the code.)

```
(gdb) clear 114
Deleted breakpoint 7
(gdb) b 48
Breakpoint 8 at 0x8048493: file pointerBugs.c, line 48.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: pointerBugs
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]


Breakpoint 8, main () at pointerBugs.c:48
48        sortArray(nelem, array, ascending);
(gdb) l
43          printf("%d ", array[indx]);
44        }
```

```
45        printf("]\n\n");
46
47      /* Sort the array of integers  */
48        sortArray(nelem, array, ascending);
49
50      /* Print the elements of the sorted array  */
51        printf("The sequence of elements in the sorted array are: [ ");
52        for (indx = 0; indx < nelem; indx++)
(gdb) l
53        {
54          printf("%d ", array[indx]);
55        }
56        printf("]\n\n");
57
58      /*  Compute average value of elements in array  */
59        getAverage(nelem, array);
60
61      /*  Print average */
62        printf("The average value of the array elements is: %f\n\n ", getAverage(nelem, array));
(gdb) l
63
64      /* Compute the frequency of each element in the array  */
65        computeFreq(nelem, array, freq);
66
67      /* Print frequencies */
68        printf("The frequencies of the elements in the sorted array are: [ ");
69        for (indx = 0; indx 70        {
71          printf("%f ", freq[indx]);
72        }
(gdb)
73        printf("]\n\n");
74
75      /* Compute the sum of the frequencies of the array elements  */
76        pfsum = fqsum(nelem, freq);
77
78      /* Print the sum of the frequencies of the array elements  */
79        printf("The sum of the element frequencies is: %f\n\n", *pfsum);
```

We see from this list that the first call to a function relating to a frequency calculation is the call to computeFreq() on line 65. Therefore, we delete our current breakpoint at line 48 and set a new breakpoint at line 65.

```
(gdb) clear 48
Deleted breakpoint 8
(gdb) b 65
Breakpoint 9 at 0x8048544: file pointerBugs.c, line 65.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: pointerBugs
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]

The sequence of elements in the sorted array are: [ 0 1 2 4 5 6 8 9 13 17 ]

The average value of the array elements is: 6.500000


Breakpoint 6, main () at pointerBugs.c:65
65        computeFreq(nelem, array, freq);
```

Now we step into this function to see if we can determine why the last element in our frequency array is not being computed.

```
(gdb) step
computeFreq (nelem_in_array=10, array=0x8049ca0, freq=0x8049cd0)
    at pointerBugs.c:166
```

Now we'll move down through this function one line at a time.

```
166       int k = 0;
```

```
(gdb) n
168       numfreq = (float *)malloc(nelem_in_array*sizeof(float));
(gdb) n
170       *numfreq = 1.0;
(gdb) n
172       for (indx = 0; indx < (nelem_in_array - 1); indx++)
(gdb) n
174         *(freq + k) += (float)(*(numfreq + k))/nelem_in_array;
```

Here on line 174 we find the expression that assigns the value for the individual elements in our frequency array, **(freq + k)*. Consequently, we might be able to get a handle on our coding error if we print out both the left- and right-hand sides of this expression. In addition, we go ahead and print out the values for both of the indices, *k* and *indx*, that are being incremented within the for loop on line 172. In order not to have to type four separate print statements each time we want to look at these values, we can define a single print variable in GDB that will allow us to print all four variable using one print command. To define our print variable, we can use the GDB command 'define' and then simply write a GDB printf statement defining what it is that we want to print out when we type our print variable.

```
(gdb) define pvar
Type commands for definition of "pvar".
End with a line saying just "end".
>printf "k = %d, indx = %d, rhs = %f, lhs = %f\n", k, indx, (float)(*(numfreq + k))/nelem_in_array,
*(freq + k)
>end
```

Now that we have defined the print variable *pvar*, let's use it to see what the current values are for the two indices and for the left- and right-hand sides of the assignment statement on line 174.

```
(gdb) pvar
k = 0, indx = 0, rhs = 0.100000, lhs = 0.000000
(gdb) n
176         if (*(array + (indx + 1)) == *(array + indx))
 (gdb) pvar
k = 0, indx = 0, rhs = 0.100000, lhs = 0.100000
```

As we can see, the left-hand side of our assignment statement, which is the value of the first element in our frequency array, *(freq + 0) = freq[0], assigned following the *if( )* statement on line 176. Since our problem appears to be that the final (10th) element in this array is not being assigned a value or is being assigned a value incorrectly, *(freq + 9) = freq[9] = 0.0, we might guess that it would not be very instructive to examine every iteration of the for loop on line 172. Therefore, our debugging effort would probably be more efficient if we could simply skip over most of the earlier iterations and jump right to the final one or two iterations prior to completion of this for loop. We can do this by setting the indices of the for loop to a value close to 9 (using the gdb command set), since this is the value of *indx* for which the loop would be exited. To be conservative, let's set the value of both *k* and *indx* to 7, move through the remaining iterations of the for loop, and print out the values defined by out print variable, *pvar*.

```
(gdb) set k = 7
(gdb) set indx = 7
(gdb) n
182           k++;
(gdb) n
183           *(numfreq + k) = 1;
(gdb) n
172       for (indx = 0; indx (gdb) n
174         *(freq + k) += (float)(*(numfreq + k))/nelem_in_array;
(gdb) pvar
k = 8, indx = 8, rhs = 0.100000, lhs = 0.000000
(gdb) n
176         if (*(array + (indx + 1)) == *(array + indx))
(gdb) pvar
k = 8, indx = 8, rhs = 0.100000, lhs = 0.100000
(gdb) n
182           k++;
(gdb) n
183           *(numfreq + k) = 1;
(gdb) n
172       for (indx = 0; indx (gdb) n
187       return freq;
(gdb) pvar
k = 9, indx = 9, rhs = 0.100000, lhs = 0.000000
```

```
(gdb) n
188     }
(gdb) pvar
k = 9, indx = 9, rhs = 0.100000, lhs = 0.000000
(gdb) n
main () at pointerBugs.c:68
68        printf("The frequencies of the elements in the sorted array are: [ ");
```

Now, the problem is quite clear. The left-hand side of the expression:

```
*(freq + k) += (float)(*(numfreq + k))/nelem_in_array;
```

is never assigned a value by the right-hand side of this expression. As we can see from the last six lines of the code listed above,

```
172       for (indx = 0; indx (gdb) n
187       return freq;
(gdb) pvar
k = 9, indx = 9, rhs = 0.100000, lhs = 0.000000
(gdb) n
188     }
(gdb) pvar
k = 9, indx = 9, rhs = 0.100000, lhs = 0.000000
(gdb) n
main () at pointerBugs.c:68
68        printf("The frequencies of the elements in the sorted array are: [ ");
```

freq = &freq[0] is returned (line 187) and the for loop is exited (line 188) before the value of the final array element (k = 9) is assigned a value. Consequently, the final element will simply have the default value it was assigned when the array was initialized at the beginning of the program,

```
/* Initialize the arrays */
for (indx = 0; indx
```

That is, *(freq + 9) = freq[9] = 0.0 .

Clearly, the reason that *freq[9]* is not assigned a value during the for loop is because there is no assignment statement,

```
*(freq + k) += (float)(*(numfreq + k))/nelem_in_array;
```

present at the end of the for loop before *freq* is returned and the loop is exited. The only assignment statement in this for loop is the one immediately after the loop is entered in line 174 (it is local to the for loop). Since this statement is inside the for loop and the for loop,

```
for (indx = 0; indx
```

exits when *indx = 8*; *(nelem_in_array (1) = (10 (1)) = 9*, which means the for loop looks like

```
for (indx = 0; indx
```

not

```
for (indx = 0; indx
```

Therefore, once the value of *indx* reaches 8, the assignment statement in line 174 that is local to the for loop cannot assign a value to *freq[9]* since there was never a ninth iteration of this loop and, therefore, never a ninth iteration of the assignment statement.

To correct this error, we need to add an additional assignment statement immediately following the for loop just before freq = &freq[0] is returned. You may have noticed that this was not an actual "pointer" bug. It was, however a bug relating to an array whose name is a pointer.

```
float *computeFreq(int nelem_in_array, int *array, float *freq)
{
  int k = 0;
  float *p, *numfreq;
  numfreq = (float *)malloc(nelem_in_array*sizeof(float));
```

```
  *numfreq = 1.0;

  for (indx = 0; indx   {
    *(freq + k) += (float)(*(numfreq + k))/nelem_in_array;

    if (*(array + (indx + 1)) == *(array + indx))
    {
      *(numfreq + k) += 1;
    }
    else
    {
      k++;
      *(numfreq + k) = 1;
    }
  }

  *(freq + k) += (float)(*(numfreq + k))/nelem_in_array;

  return freq;
}
```

2.7.4.7 Debugged Code

Now that we have made the corrections to our code, we recompile and rerun our program.

```
$ gcc -g -o pointerBugs pointerBugs.c
pointerBugs.c: In function `main':

(gdb) clear 65
Deleted breakpoint 9
(gdb) run
Starting program: /home/rmshee0/debugging/C_files/chapterExample/fifthCorrection
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]

The sequence of elements in the sorted array are: [ 0 1 2 4 5 6 8 9 13 17 ]

The average value of the array elements is: 6.500000

 The frequencies of the elements in the sorted array are: [ 0.100000 0.100000 0.100000 0.100000 0.100000
0.100000 0.100000 0.100000 0.100000 0.100000 ]

The sum of the element frequencies is: 1.000000

Program exited normally.
(gdb)
```

This is exactly the correct output that we wanted, so it looks like we have successfully debugged our code.

The final version of our fully debugged source code is shown below. Since it has been debugged, we rename it noPointerBugs.c.

noPointerBugs.c

```
#include <stdio.h>
#include <stdlib.h>

int i, j, indx;
int sflag;
int sum;

void populateArray(int nelem_in_array, int *intArray);
void sortArray(int nelem_in_array, int *array, int (*fcn)(int sortFlag, int n, int m));
void exchange(int *elem1, int *elem2);
int ascending(int sortFlag, int n, int m);
int descending(int sortFlag, int n, int m);
float getAverage(int nelem_in_array, int *array);
float *computeFreq(int nelem_in_array, int *array, float *freq);
float *fqsum(int nelem_in_array, float *freq);
```

```
int main(void)
{
  const int nelem = 10;

  int *array;
  float *freq;
  float *pfsum = 0;

  array = (int *)malloc(nelem*sizeof(int));
  freq = (float *)malloc(nelem*sizeof(float));

/*  Initialize the arrays  */
  for (indx = 0; indx < nelem; indx++)
  {
    array[indx] = 0;
    freq[indx] = 0.0;
  }

/* Replace initial values of array elements  */
  populateArray(nelem, array);

/* Print the elements of the unsorted array  */
  printf("The sequence of elements in the unsorted array are: [ ");

  for (indx = 0; indx < nelem; indx++)
  {
    printf("%d ", array[indx]);
  }
  printf("]\n\n");

/* Sort the array of integers  */
  sortArray(nelem, array, ascending);

/* Print the elements of the sorted array  */
  printf("The sequence of elements in the sorted array are: [ ");
  for (indx = 0; indx < nelem; indx++)
  {
    printf("%d ", array[indx]);
  }
  printf("]\n\n");

/*  Compute average value of elements in array  */
  getAverage(nelem, array);

/*  Print average */
  printf("The average value of the array elements is: %f\n\n ", getAverage(nelem, array));

/* Compute the frequency of each element in the array  */
  computeFreq(nelem, array, freq);

/* Print frequencies */
  printf("The frequencies of the elements in the sorted array are: [ ");
  for (indx = 0; indx < nelem; indx++)
  {
    printf("%f ", freq[indx]);
  }
  printf("]\n\n");

/* Compute the sum of the frequencies of the array elements  */
  pfsum = fqsum(nelem, freq);

/* Print the sum of the frequencies of the array elements  */
  printf("The sum of the element frequencies is: %f\n\n", *pfsum);

  free(array);
  free(freq);

  return 0;
}
```

```
void populateArray(const int nelem_in_array, int *intArray)
{
  for (j = 0; j < nelem_in_array; j++)
  {
    if ((j % 2) == 0)
    {
      *intArray = (2*j + 1);
    }
    else
    {
      *intArray = (j - 1);
    }

  intArray += 1;
  }
}

void sortArray(const int nelem_in_array, int *array, int (*order)(int, int, int))
{
  for (i = 0; i < nelem_in_array; i++)
  {
    for (indx = 0; indx < (nelem_in_array - 1); indx++)
    {
      sflag = 0;

      if (!(*order)(sflag, *(array + indx) , *(array + (indx + 1))))
      {
        exchange((array + indx), (array + (indx + 1)));
      }
    }
  }
}

void exchange(int *elem1, int *elem2)
{
  int tmp;

  tmp = *elem1;
  *elem1 = *elem2;
  *elem2 = tmp;
}

int ascending(int sflag, int n, int m)
{
  sflag = 0;

  if (m > n)
  {
   sflag = 1;
  }
  return sflag;
}

int decending(int sflag, int n, int m)
{
  sflag = 0;

  if (n > m)
  {
    sflag = 1;
  }
  return sflag;
}

float getAverage(int nelem_in_array, int *array)
{
  sum = 0;

  for (indx = 0; indx < nelem_in_array; indx++)
```

```
  {
    sum += *(array + indx);
  }
  return (float)sum/nelem_in_array;
}

float *computeFreq(int nelem_in_array, int *array, float *freq)
{
  int k = 0;
  float *p, *numfreq;
  numfreq = (float *)malloc(nelem_in_array*sizeof(float));

  *numfreq = 1.0;

  for (indx = 0; indx < (nelem_in_array - 1); indx++)
  {
    *(freq + k) += (float)(*(numfreq + k))/nelem_in_array;

    if (*(array + (indx + 1)) == *(array + indx))
    {
      *(numfreq + k) += 1;
    }
    else
    {
      k++;
      *(numfreq + k) = 1;
    }
  }

  *(freq + k) += (float)(*(numfreq + k))/nelem_in_array;

  return freq;
}

float *fqsum(int nelem_in_array, float *freq)
{
  float *fsum = (float *)malloc(sizeof(float));
  *fsum = 0.0;

  for (indx = 0; indx < nelem_in_array; indx++)
  {
    *fsum += freq[indx];
  }

  return fsum;
}
```

2.7.5 Debugging Exercises

Now that you have worked along with the sample debugging sessions, we give you the opportunity to debug some code with pointer errors on your own. The following five problems provide you with C (and C++) codes that contain one or more pointer bugs as a result of programming errors. See if you can identify the coding error(s) using a debugger of your choice and correct them so that the code runs successfully.

Problem 1. pointerError1.c

```
#include <stdio.h>
#include <stdlib.h>

int i, j, indx;
int sflag;
int sum;

void populateArray(int nelem_in_array, int *array);
void sortArray(int nelem_in_array, int *array, int (*fcn)(int sortFlag, int n, int m));
void exchange(int *elem1, int *elem2);
int ascending(int sortFlag, int n, int m);
int descending(int sortFlag, int n, int m);
int sumElements(int nelem_in_array, int *array);
```

```
float getAverage(int nelem_in_array, int *array);

int main(void)
{
  const int nelem = 10;

  int *array;
  array = (int *)malloc(nelem*sizeof(int));

/*  Initialize the arrays  */
  for (indx = 0; indx < nelem; indx++)
  {
    array[indx] = 0;
  }

/* Replace initial values of array elements  */
  populateArray(nelem, array);

/* Print the elements of the unsorted array  */
  printf("The sequence of elements in the unsorted array are: [ ");

  for (indx = 0; indx < nelem; indx++)
  {
    printf("%d ", array[indx]);
  }
  printf("]\n\n");

/* Sort the array of integers  */
  sortArray(nelem, array, ascending);

/* Print the elements of the sorted array  */
  printf("The sequence of elements in the sorted array are: [ ");
  for (indx = 0; indx < nelem; indx++)
  {
    printf("%d ", array[indx]);
  }
  printf("]\n\n");

/*  Compute average value of elements in array  */
  getAverage(nelem, array);

/* Print sum of elements in array  */
  printf("The sum of the %d elements in the array is: %d\n\n ", nelem, sum);

/*  Print average */
/*  printf("The average value of the array elements is: %f\n\n ", getAverage(nelem, array));  */

  free(array);

  return 0;
}

void populateArray(const int nelem_in_array, int *intArray)
{
  for (j = 0; j < nelem_in_array; j++)
  {
    if ((j % 2) == 0)
    {
      *intArray = 10;
    }
    else
    {
      *intArray = 100;
    }

    intArray += 1;
  }
}
```

```
void sortArray(const int nelem_in_array, int *array, int (*order)(int, int, int))
{
  for (i = 0; i < nelem_in_array; i++)
  {
    for (indx = 0; indx < (nelem_in_array - 1); indx++)
    {
      sflag = 0;

      if (!(*order)(sflag, *(array + indx) , *(array + (indx + 1))))
      {
        exchange((array + indx), (array + (indx + 1)));
      }
    }
  }
}

void exchange(int *elem1, int *elem2)
{
  int tmp;

  tmp = *elem1;
  *elem1 = *elem2;
  *elem2 = tmp;
}

int ascending(int sflag, int n, int m)
{
  sflag = 0;

  if (m > n)
  {
   sflag = 1;
  }
  return sflag;
}

int decending(int sflag, int n, int m)
{
  sflag = 0;

  if (n > m)
  {
    sflag = 1;
  }
  return sflag;
}

float getAverage(int nelem_in_array, int *array)
{
  sumElements(nelem_in_array, array);

  return (float)sum/nelem_in_array;
}


int sumElements(int nelem_in_array, int *array)
{
  sum = 0;

  for (indx = 0; indx < nelem_in_array; indx++)
  {
    sum += *array + indx;
  }
  return sum;
}
```

Problem 2. pointerError2.c

```
#include <stdio.h>
```

```
#include <stdlib.h>

int i, j, indx;
int sflag;
int sum;

void populateArray(int nelem_in_array, int array);
void sortArray(int nelem_in_array, int *array, int (*fcn)(int sortFlag, int n, int m));
void exchange(int *elem1, int *elem2);
int ascending(int sortFlag, int n, int m);
int descending(int sortFlag, int n, int m);
float getAverage(int nelem_in_array, int *array);

int main(void)
{
  const int nelem = 10;

  int *array;

  array = (int *)malloc(nelem*sizeof(int));

/*  Initialize the arrays  */
  for (indx = 0; indx < nelem; indx++)
  {
    array[indx] = 0;
  }

/* Replace initial values of array elements  */
  populateArray(nelem, *array);

/* Print the elements of the unsorted array  */
  printf("The sequence of elements in the unsorted array are: [ ");

  for (indx = 0; indx < nelem; indx++)
  {
    printf("%d ", array[indx]);
  }
  printf("]\n\n");

/* Sort the array of integers  */
  sortArray(nelem, array, ascending);

/* Print the elements of the sorted array  */
  printf("The sequence of elements in the sorted array are: [ ");
  for (indx = 0; indx < nelem; indx++)
  {
    printf("%d ", array[indx]);
  }
  printf("]\n\n");

/*  Compute average value of elements in array  */
  getAverage(nelem, array);

/* Print sum of elements in array  */
  printf("The sum of the %d elements in the array is: %d\n\n ", nelem, sum);

/*  Print average */
  printf("The average value of the array elements is: %f\n\n ", getAverage(nelem, array));


  free(array);

  return 0;
}

void populateArray(const int nelem_in_array, int intArray)
{
  for (j = 0; j < nelem_in_array; j++)
  {
```

```
    if ((j % 2) == 0)
    {
      intArray = (2*j + 1);
    }
    else
    {
      intArray = (j - 1);
    }

    intArray += 1;
  }
}

void sortArray(const int nelem_in_array, int *array, int (*order)(int, int, int))
{
  for (i = 0; i < nelem_in_array; i++)
  {
    for (indx = 0; indx < (nelem_in_array - 1); indx++)
    {
      sflag = 0;

      if (!(*order)(sflag, *(array + indx) , *(array + (indx + 1))))
      {
        exchange((array + indx), (array + (indx + 1)));
      }
    }
  }
}

void exchange(int *elem1, int *elem2)
{
  int tmp;

  tmp = *elem1;
  *elem1 = *elem2;
  *elem2 = tmp;
}

int ascending(int sflag, int n, int m)
{
  sflag = 0;

  if (m > n)
  {
   sflag = 1;
  }
  return sflag;
}

int decending(int sflag, int n, int m)
{
  sflag = 0;

  if (n > m)
  {
    sflag = 1;
  }
  return sflag;
}

float getAverage(int nelem_in_array, int *array)
{
  sum = 0;

  for (indx = 0; indx < nelem_in_array; indx++)
  {
    sum += *(array + indx);
  }
  return (float)sum/nelem_in_array;
```

```
}
```

Problem 3. pointerError3.c

```
#include <stdio.h>
#include <stdlib.h>

int i, j, indx;
int sflag;
int *sum;
float average;

void populateArray(int nelem_in_array, int *array);
void sortArray(int nelem_in_array, int *array, int (*fcn)(int sortFlag, int n, int m));
void exchange(int *elem1, int *elem2);
int ascending(int sortFlag, int n, int m);
int descending(int sortFlag, int n, int m);
int *sumElements(int nelem_in_array, int *array);

int main(void)
{
  const int nelem = 10;

  int *array;
  array = (int *)malloc(nelem*sizeof(int));

/*  Initialize the arrays  */
  for (indx = 0; indx < nelem; indx++)
  {
    array[indx] = 0;
  }

/* Replace initial values of array elements  */
  populateArray(nelem, array);

/* Print the elements of the unsorted array  */
  printf("The sequence of elements in the unsorted array are: [ ");

  for (indx = 0; indx < nelem; indx++)
  {
    printf("%d ", array[indx]);
  }
  printf("]\n\n");

/* Sort the array of integers  */
  sortArray(nelem, array, ascending);

/* Print the elements of the sorted array  */
  printf("The sequence of elements in the sorted array are: [ ");
  for (indx = 0; indx < nelem; indx++)
  {
    printf("%d ", array[indx]);
  }
  printf("]\n\n");

/*  Compute the sum of the elements in the array  */
  sumElements(nelem, array);

/* Print sum of elements in the array  */
  printf("The sum of the %d elements in the array is: %d\n\n ", nelem, sum);

  free(array);

  return 0;
}

void populateArray(const int nelem_in_array, int *intArray)
{
  for (j = 0; j < nelem_in_array; j++)
```

```
  {
    if ((j % 2) == 0)
    {
      *intArray = 10;
    }
    else
    {
      *intArray = 100;
    }

    intArray += 1;
  }
}

void sortArray(const int nelem_in_array, int *array, int (*order)(int, int, int))
{
  for (i = 0; i < nelem_in_array; i++)
  {
    for (indx = 0; indx < (nelem_in_array - 1); indx++)
    {
      sflag = 0;

      if (!(*order)(sflag, *(array + indx) , *(array + (indx + 1))))
      {
        exchange((array + indx), (array + (indx + 1)));
      }
    }
  }
}

void exchange(int *elem1, int *elem2)
{
  int tmp;

  tmp = *elem1;
  *elem1 = *elem2;
  *elem2 = tmp;
}

int ascending(int sflag, int n, int m)
{
  sflag = 0;

  if (m > n)
  {
   sflag = 1;
  }
  return sflag;
}

int decending(int sflag, int n, int m)
{
  sflag = 0;

  if (n > m)
  {
    sflag = 1;
  }
  return sflag;
}

int *sumElements(int nelem_in_array, int *array)
{
  sum = 0;

  for (indx = 0; indx < nelem_in_array; indx++)
  {
    sum += array[indx];
  }
```

```
  return sum;
}
```

Problem 4. sneakyError.c

Consider the following scenario. You have written a program that compiles successfully with no warning messages of any kind. The compiled code also runs successfully and gives results that are absolutely correct. Obviously, it is perfectly safe to assume that your code is bug-free. Or is it?

Some programs may run over an extensive period of time with no indication of any problem at all. Then one day, quite unexpectedly, your program gives you results that are completely incorrect. Or, perhaps, your program fails to run at all. Unfortunately, even programs that appear to be bug-free may, in fact, harbor a hidden or dormant bug just waiting to emerge when you least expect it. An example of such a hidden bug is contained in the example code "noPointerBugs.c" that we (supposedly) debugged successfully earlier in this chapter. It turns out that one of the functions in this debugged code is defined incorrectly. However, this bug remained hidden because of the particular data we used to populate the 10-element array. To see this bug emerge, make the following changes in the definition of the function populateArray() in noPointerBugs.c:

```
void populateArray(const int nelem_in_array, int *intArray)
    {
      for (j = 0; j < nelem_in_array; j++)
      {
        if ((j % 2) == 0)
        {
          intArray[j] = 10;
        }
        else
        {
          intArray[j] = 1000;
        }
      }
    }
```

Then recompile it, run the executable, and look at the results. Debug this code in your debugger of choice and see if you can correct the programming error(s).

Problem 5. Pointer Bugs in C++

Since the name of an array is simply an address (specifically, the address of the first element in the array), then we can always point to the beginning of an array of elements using a pointer. The statement

ptr = arr;

initializes a pointer, *ptr*, to point to *&arr[0]*, where *arr* is the name of some array.

If we want to write a function that returns the address of the beginning of the array *arr*, one way to do this is to have the function return the name of the array, *arr*. An alternative way would be to initialize a pointer, *ptr*, using the statement shown above and have the function return the pointer, *ptr*.

The following C++ program uses this concept to copy the elements from one array to another array. Run this program to see if it completes successfully and, if not, debug the program in the debugger of your choice, identify the bug(s) in the code, and see if you can correct the error(s) so that the program runs successfully.

Note that in the code below we are using the ANSI-C++ standard header files.

anotherPointerBug.cc

```
#include <cstdio>
using namespace std;
#include <cstdlib>
using namespace std;
#include <iostream>
using namespace std;

class arrayCopier
{
```

```
  public:

    double *copyArray(int nelem_in_array, double *array2Bcopied, double *copied2Array)
    {
      int index;
      double *ptr;

      ptr = copied2Array;

      for (index = 0; index < nelem_in_array; index++)
      {
        *ptr = array2Bcopied[index];

        if (index < (nelem_in_array - 1))
        {
          ptr++;
        }
      }

      return ptr;
    }

    void printArray(int nelem_in_array, double *array2Bprinted)
    {
      int index;

      cout << '\n';
      cout << "The elements copied from the original array to the new array are [  ";

      for (index = 0; index < nelem_in_array; index++)
      {
        cout << *array2Bprinted << "  ";
        array2Bprinted++;
      }

      cout << "]" << '\n';
      cout << '\n';
    }
};


int main(int argc, char *argv[])
{
  int nelem, index;
  double *origArray, *newArray;

// Instantiate a pointer to an arrayCopier object
  arrayCopier *cpy = new arrayCopier;

// Set number of elements in array
  nelem = 5;

// Dynamically allocate memory for each array
  origArray = new double[nelem];
  newArray = new double[nelem];

// Initialize array that is to be copied
  for (index = 0; index < nelem; index++)
  {
    origArray[index] = (index + 1);
  }

// Copy the original array to the new array
  newArray = (cpy -> copyArray(nelem, origArray, newArray));

// Print the elements of the new array
  cpy -> printArray(nelem, newArray);

  delete[] origArray;
```

```
  delete[] newArray;

    return 0;
}
```

2.7.6 Solutions to Exercises

Problem 1. pointerError1.c

Running pointerError1.c in GDB generates the following output:

```
$ gcc g o pointerError1 pointerError1.c

$ gdb pointerError1

(gdb) run
Starting program: pointerError1
The sequence of elements in the unsorted array are: [ 10 100 10 100 10 100 10 100 10 100 ]

The sequence of elements in the sorted array are: [ 10 10 10 10 10 100 100 100 100 100 ]

The sum of the 10 elements in the array is: 145

Program exited normally.
```

Clearly, something has gone wrong in this run. The unsorted array was sorted correctly, but the sum of the elements in the array is obviously incorrect. All of the elements in the array are positive and five of the ten elements have the value 100, which tells us that the sum of the array elements should be greater than 500. Yet, the output from our program gives the sum of all ten elements as only 145. Consequently, our initial guess might be that the function that computes the sum of the array elements is coded incorrectly; i.e., it contains a bug.

If we look at our code for pointerError1.c we see that the function sumElements() computes the sum of the array elements.

```
int sumElements(int nelem_in_array, int *array)
{
  sum = 0;

  for (indx = 0; indx   {
    sum += *array + indx;
  }
  return sum;
}
```

We also see that this function is called from within the function getAverage(), which computes the average value of the elements in the array.

```
float getAverage(int nelem_in_array, int *array)
{
  sumElements(nelem_in_array, array);

  return (float)sum/nelem_in_array;
}
```

If we list the individual lines of code in our program, we find that the function getAverage() is on line 53.

```
(gdb) list

.   .   .

41      /* Sort the array of integers  */
42        sortArray(nelem, array, ascending);
43
44      /* Print the elements of the sorted array  */
45        printf("The sequence of elements in the sorted array are: [ ");
46        for (indx = 0; indx 47        {
48          printf("%d ", array[indx]);
(gdb)
49        }
50        printf("]\n\n");
```

```
51
52      /*  Compute average value of elements in array  */
53        getAverage(nelem, array);
54
55      /* Print sum of elements in array  */
56        printf("The sum of the %d elements in the array is: %d\n\n ", nelem, sum);
57
58      /*  Print average */
(gdb)
```

Therefore, we set a breakpoint at line 53 and step through the function getAverage(), which steps us into the function sumElements(). We then step through the function sumElements() and print out the values of **array, *array + indx,* and sum after each iteration of the summation loop. We also print the value of *indx* so that we can monitor our progress through the array. Since we want to print the values of four separate variable, we use the GDB 'define' command to construct a single print statement that we can use to print all four values with one print command.

```
(gdb) b 53
Breakpoint 1 at 0x80484d7: file pointerError1.c, line 53.
(gdb) run
Starting program: pointerError1
The sequence of elements in the unsorted array are: [ 10 100 10 100 10 100 10 100 10 100 ]

The sequence of elements in the sorted array are: [ 10 10 10 10 10 100 100 100 100 100 ]

Breakpoint 1, main () at pointerError1.c:53
53        getAverage(nelem, array);
(gdb) s
getAverage (nelem_in_array=10, array=0x80499c8) at pointerError1.c:132
132       sumElements(nelem_in_array, array);
(gdb) s
sumElements (nelem_in_array=10, array=0x80499c8) at pointerError1.c:140
140       sum = 0;
(gdb) define pvar
Type commands for definition of "pvar".
End with a line saying just "end".
>printf "indx = %d\n *array = %d\n *array + indx = %d\n sum = %d\n", indx, *array, *array+indx,
sum
>end
(gdb) pvar
indx = 0
 *array = 10
 *array + indx = 10
 sum = 10
(gdb) s
144         sum += *array + indx;
(gdb)
142       for (indx = 0; indx < nelem_in_array; indx++)
(gdb) pvar
indx = 1
 *array = 10
 *array + indx = 11
 sum = 21
(gdb) s
144         sum += *array + indx;
(gdb) s
142       for (indx = 0; indx < nelem_in_array; indx++)
(gdb) pvar
indx = 2
 *array = 10
 *array + indx = 12
 sum = 33
(gdb) s
144         sum += *array + indx;
(gdb) s
142       for (indx = 0; indx < nelem_in_array; indx++)
(gdb) pvar
indx = 3
 *array = 10
 *array + indx = 13
```

```
 sum = 46
(gdb) s
144         sum += *array + indx;
(gdb) s
142       for (indx = 0; indx < nelem_in_array; indx++)
(gdb) pvar
indx = 4
 *array = 10
 *array + indx = 14
 sum = 60
(gdb) s
144         sum += *array + indx;
(gdb) s
142       for (indx = 0; indx < nelem_in_array; indx++)
(gdb) pvar
indx = 5
 *array = 10
 *array + indx = 15
 sum = 75
```

We can see that during the first five iterations everything looks alright. However, when we get to the sixth iteration (*indx = 5*), things start to go wrong. The sixth element in our sorted array is 100. But the printed output gives the value of the array element after the sixth iteration as 10, not 100. Clearly, the correct array element is not being used in the expression

```
sum += *array + indx;
```

To see which element is being used to compute this sum during the sixth iteration of the summation loop, we print out the address of this element.

```
(gdb) p array
 = (int *) 0x80499c8
```

This certainly does not look right because as we can see from the values of the arguments of sumElements() reported when we initially stepped into this function at line 140,

```
sumElements (nelem_in_array=10, array=0x80499c8) at Problem1.c:140 ,
```

the address of the element being used in the sixth iteration of the summation loop,

```
(gdb) p array
 = (int *) 0x80499c8
```

is the same as the address of the beginning of the array, 0x80499c8.

Moreover, if we move to the final iterative step in the summation loop and look at the address of the array element being used in the expression

```
sum += *array + indx;
```

we find that we are at the address of the beginning of the array,

```
(gdb) set indx = ( nelem_in_array □ 1 )
(gdb) n
146       return sum;
(gdb) n
147     }
(gdb) p array
 = (int *) 0x80499c8
(gdb)
```

Consequently, instead of moving downstream through the array and computing the cumulative sum of sequential elements in the array, we are just repeatedly adding the iterations number to the first element of the array and computing its cumulative sum over the ten iterations of the summation loop. That is, we are actually computing

```
(nelem) * (*array)  + [ (nelem □ 1) * (nelem) ] / 2  =  10*10  +  (9*10)/2  = 100 + 45 = 145
```

It is now exactly clear what our programming error is — we are using pointer arithmetic to traverse our array, but we are using it incorrectly. The pointer array points to the first element of our array, so the dereferenced pointer, **array*, will

always equal 10. Consequently, we are not traversing our array during the ten iterations of the summation loop in the function sumElements(). The reason is that we failed to place parenthesis around the expression *array + indx*.

The address of the *(k+1)*st element in an array, *arr*, is given by *(arr + k)*. For example, the address of the last (10^{th}) element of the array in Problem 1 is 0x80499ec.

```
(gdb) p (array+( nelem_in_array □ 1 ) )
 = (int *) 0x80499ec
```

which is 36 bytes downstream from the address of the first element in the array

```
(gdb) p ( ( ( array+( nelem_in_array □ 1 ) ) ? array ) * sizeof( int ) )
 = 36
```

Since the address of the *(indx + 1)*st element in array is *(array + indx)*, then the value of the *(indx + 1)*st element would be **(array + indx)*, not **array + indx*. Therefore, to fix our bug, we need to make the following correction to our program code in the function sumElements():

```
int sumElements(int nelem_in_array, int *array)
{
  sum = 0;

  for (indx = 0; indx   {
    sum += *(array + indx);
  }
  return sum;
}
```

After this correction is made, our program gives us the correct results.

```
$ gcc □g □o pointerError1 pointerError1.c

$ gdb pointerError1

(gdb) run
Starting program: pointerError1
The sequence of elements in the unsorted array are: [ 10 100 10 100 10 100 10 100 10 100 ]

The sequence of elements in the sorted array are: [ 10 10 10 10 10 100 100 100 100 100 ]

The sum of the 10 elements in the array is: 550

 Program exited normally.
```

It is probably quite obvious that this bug could have been avoided altogether if we had used array indexing instead of pointer arithmetic to sum the array elements in the function sumElements():

```
int sumElements(int nelem_in_array, int *array)
{
  sum = 0;

  for (indx = 0; indx < nelem_in_array; indx++)
  {
    sum += array[indx];
  }
  return sum;
}
```

Problem 2. pointerError2.c

Running pointerError2.c in GDB generates the following output:

```
$ gcc □g □o pointerError2 pointerError2.c

$ gdb pointerError2

(gdb) run
Starting program: pointerError2
The sequence of elements in the unsorted array are: [ 0 0 0 0 0 0 0 0 0 0 ]
```

```
The sequence of elements in the sorted array are: [ 0 0 0 0 0 0 0 0 0 0 ]

The sum of the 10 elements in the array is: 0

The average value of the array elements is: 0.000000

Program exited normally.
```

Something definitely appears to have gone wrong in this run. Every result in the output is zero, which is not likely to be the results you would expect. Of course it is not surprising that the sum and average of the elements in the array are since every element in the unsorted array is zero. Consequently, our initial guess might be that there is a bug in the section that assigns values to the elements that make up the unsorted array.

We can see in PointerError2.c that when the unsorted array is initialized, all elements are given a value of zero. Subsequently, these initial values are replaced by a call to a function named populateArray(). Since the values of all array elements in the output are zero (the initialized values), it appears that the program is simply failing to replace these initial values with nonzero values. Consequently, it is quite likely that the bug is occurring somewhere within the function populateArray().

If we list the lines of code, we see that the function populateArray() is on line 30.

```
$ gdb pointerError2

(gdb) list
8       void populateArray(int nelem_in_array, int array);
9       void sortArray(int nelem_in_array, int *array, int (*fcn)(int sortFlag, int n, int m));
10      void exchange(int *elem1, int *elem2);
11      int ascending(int sortFlag, int n, int m);
12      int descending(int sortFlag, int n, int m);
13      float getAverage(int nelem_in_array, int *array);
14
15      int main(void)
16      {
17        const int nelem = 10;
(gdb)
18
19        int *array;
20
21        array = (int *)malloc(nelem*sizeof(int));
22
23      /*  Initialize the arrays  */
24        for (indx = 0; indx 25        {
26          array[indx] = 0;
27        }
(gdb)
28
29      /* Replace initial values of array elements  */
30        populateArray(nelem, *array);
31
```

Therefore, we can set a breakpoint at line 30 and step into and through the function populateArray() to see if we can diagnose our coding error(s).

```
(gdb) b 30
Breakpoint 1 at 0x80483f0: file pointerError2.c, line 30.
(gdb) run
Starting program: pointerError2

Breakpoint 1, main () at pointerError2.c:30
30        populateArray(nelem, *array);
(gdb) s
populateArray (nelem_in_array=10, intArray=0) at pointerError2.c:69
69        for (j = 0; j
```

Note that as soon as we step into the function populateArray() we find a potential error. If we look closely at the value of each argument being passed to he function, we see that the first argument, *nelem_in_array*, has a value of 10 and the second argument, *intArray*, has a value of 0,

```
populateArray (nelem_in_array=10, intArray=0) at pointerError2.c:69
```

This tells us that both arguments are being passed using the method known as call-by-value.

The whole reason that we are including populateArray() in our program is to modify the values of each element in the array. When we invoke any function using call-by-value, what gets passed to the function is only a copy of the argument's value. As we see on line 30, because of the way we are invoking populateArray(), we are passing a copy of the value of the first element in our array, **array*, to this function,

```
Breakpoint 1, main () at pointerError2.c:30
30        populateArray(nelem, *array);
```

Once this copy of **array* is passed to populateArray(), the function modifies the value of this copy. Unfortunately, any modification to this copied value does not affect the original value of the argument **array*. That is, when an argument is passed call-by- value to a function, whatever takes place inside the function has no effect on the actual value of the argument used in the call. Therefore, the original values of the elements in array, [0 0 0 0 0 0 0 0 0 0], are not affected by the call to populateArray().

We can confirm that it is only a copy of the original value of the first element of array that is being passed to populateArray() by comparing the address of the first element of *array* and the address of the value being passed to populateArray().

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: pointerError2

Breakpoint 1, main () at pointerError2.c:30
30        populateArray(nelem, *array);
(gdb) p array
 = (int *) 0x8049a28
(gdb) s
populateArray (nelem_in_array=10, intArray=0) at pointerError2.c:69
69        for (j = 0; j (gdb) p &intArray
(gdb) run The program being debugged has been started already. Start it from the beginning? (y or n) y Starting
program: pointerError2 Breakpoint 1, main () at pointerError2.c:30 30 populateArray(nelem, *array); (gdb) p array $1
= (int *) 0x8049a28 (gdb) s populateArray (nelem_in_array=10, intArray=0) at pointerError2.c:69 69 for (j = 0; j <
nelem_in_array; j++) (gdb) p &intArray $2 = (int *) 0xbfffea84 = (int *) 0xbfffea84
```

Clearly, the address of the value of the first element of *array*, 0x8049a28, and the address of the value of the argument being passed to populateArray(), 0xbfffea84, are not the same. This confirms that it is only a copy of the first element of *array* that is being modified by our function.

Obviously, modifying a copy of the value of the first element in *array* is not what we want our program to do. Instead, we want our function populateArray() to modify the actual value of each of the elements in *array* when it is invoked.

To pass the actual value of an argument to a function, instead of a copy of the value of the argument, we need to pass the address of the original value of the argument to the function. This method of invoking a function is known as "call-by-reference". When an argument is passed call-by-reference, it is a copy of the address of the argument that is passed (not a copy of the argument's value). In C (and C++), an argument can be passed to a function call-by-reference by passing a pointer to the argument.

Therefore, if we are going to modify the initial zero values of each of the elements in *array*, we need to access the actual data contained in *array*. We do this by passing a pointer to the first element of *array*; i.e., we pass the name of the array, *array*, to the function populateArray(),

```
populateArray(nelem, array);
```

Since the name of an array is a pointer, we next have to modify the way we have defined our function populateArray(), as well as the header for this function (in line 8), in order to tell the function that we are passing a pointer as the second argument. We also must modify the body of our function definition so that our function modifies the value of the appropriate element of *array* during each iteration of the for loop. Therefore, to fix our bug, we need to make the following corrections to populateArray() in our program code:

```
void populateArray(int nelem_in_array, int array);        /*  function header  */

.   .   .
```

```
void populateArray(const int nelem_in_array, int *intArray)    /*  function definition  */
{
  for (j = 0; j   {
    if ((j % 2) == 0)
    {
      *intArray = (2*j + 1);
    }
    else
    {
      *intArray = (j - 1);
    }

    intArray += 1;     /*  move 1 element downstream in intArray at the end of each iteration  */
  }
}
```

After this correction is made, our program gives us the correct results.

```
$ gcc g o pointerError2 pointerError2.c

$ gdb pointerError2

(gdb) run
Starting program: pointerError2
The sequence of elements in the unsorted array are: [ 1 0 5 2 9 4 13 6 17 8 ]

The sequence of elements in the sorted array are: [ 0 1 2 4 5 6 8 9 13 17 ]

The sum of the 10 elements in the array is: 65

The average value of the array elements is: 6.500000

 Program exited normally.
```

Problem 3. pointerError3.c

Running pointerError3.c in GDB generates the following output:

```
$ gcc g o pointerError3 pointerError3.c

$ gdb pointerError3

(gdb) run
Starting program: pointerError3
The sequence of elements in the unsorted array are: [ 10 100 10 100 10 100 10 100 10 100 ]

The sequence of elements in the sorted array are: [ 10 10 10 10 10 100 100 100 100 100 ]

The sum of the 10 elements in the array is: 2200

Program exited normally.
```

Our output shows that the unsorted array was sorted correctly. However, the sum of the elements in the sorted array is clearly in error. Since all ten array elements have values no greater than 100, the sum of the elements cannot be greater than 10*100 = 1000. But our program is telling us that the sum is 2200, which is certainly greater than 1000.

Based on our output, a good initial guess would be that there is a coding error (or errors)in the function that computes the sum of the elements in our array,

```
int *sumElements(int nelem_in_array, int *array)
{
  sum = 0;

  for (indx = 0; indx   {
    sum += array[indx];
  }
  return sum;
}
```

If we list the lines of our code in pointerError3, we see that the function sumElements()is called in line 53.

```
(gdb) list
9       void populateArray(int nelem_in_array, int *array);
10      void sortArray(int nelem_in_array, int *array, int (*fcn)(int sortFlag, int n, int m));
11      void exchange(int *elem1, int *elem2);
12      int ascending(int sortFlag, int n, int m);
13      int descending(int sortFlag, int n, int m);
14      int *sumElements(int nelem_in_array, int *array);
15
16      int main(void)
17      {
18        const int nelem = 10;
(gdb)
19
20        int *array;
21        array = (int *)malloc(nelem*sizeof(int));
.  .  .

44      /* Print the elements of the sorted array  */
45        printf("The sequence of elements in the sorted array are: [ ");
46        for (indx = 0; indx < nelem; indx++)
47        {
48          printf("%d ", array[indx]);
(gdb)
49        }
50        printf("]\n\n");
51
52      /*  Compute the sum of the elements in the array  */
53        sumElements(nelem, array);
54
55      /* Print sum of elements in the array  */
56        printf("The sum of the %d elements in the array is: %d\n\n ", nelem, sum);
57
58        free(array);
```

Therefore, we will set a breakpoint at line 53 and step into and through the function sumElements() in an attempt to find the bug in our program. Since this function computes the cumulative sum of all the elements in *array*, which is the result that is in error in our current program, we print out the value of both the left- and right-hand sides of the expression,

```
sum += array[indx];
```

to see exactly what values are being computed within this function.

```
(gdb) b 53
Breakpoint 1 at 0x80484d7: file pointerError3.c, line 53.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: pointerError3
The sequence of elements in the unsorted array are: [ 10 100 10 100 10 100 10 100 10 100 ]

The sequence of elements in the sorted array are: [ 10 10 10 10 10 100 100 100 100 100 ]

Breakpoint 1, main () at pointerError3.c:53
53        sumElements(nelem, array);
(gdb) s
sumElements (nelem_in_array=10, array=0x80499d0) at pointerError3.c:129
129       sum = 0;
(gdb) s
131       for (indx = 0; indx (gdb) s
133         sum += array[indx];
(gdb) p indx
 = 0
(gdb) p array[indx]
 = 10
(gdb) p sum
 = (int *) 0x0
```

The values of *indx* and *array[indx]* certainly look correct. However, when we print out the integer value *sum*, we immediately discover an error. The value of *sum* is not an integer value at all; instead, it is an address. If we attempt to print the value of the data at this address,

```
(gdb) p *sum
Cannot access memory at address 0x0
```

we get an error message that looks suspiciously like an uninitialized pointer error. If we again look at our code for this function, we see that *sum* is actually a null pointer and that the above error occurs when we attempt to dereference this null pointer, **sum*. In line 6 of our code, we have declared the pointer *sum*,

```
#include
#include

int i, j, indx;
int sflag;
int *sum;
float average;
```

Then in our function, sumElements(), we initialize this pointer to point to nothing:

```
int *sumElements(int nelem_in_array, int *array)
{
  sum = 0;

  for (indx = 0; indx   {
    sum += array[indx];
  }
  return sum;
}
```

As we can see, our function is supposed to return a pointer to an integer,

```
int *sumElements( )
```

and, in fact, it does,

```
for (indx = 0; indx < nelem_in_array; indx++)
{
  sum += array[indx];
}
return sum;
```

since sum is a pointer to *int*

,

```
int *sum;
```

However, if we look carefully at our function, we see that the pointer being returned by our function does not make sense, we do not know where it is pointing and we do not know to what it is pointing.

There are two basic problems with the way we have coded the function sumElements(). First, the function returns a pointer to *int*, but the function never assigns any value to the data to which it is being pointed. It only sequentially increments the address of the null pointer *sum* and then returns the final address of this pointer. Any actual data associated with this address would be meaningless since that data would have no relationship at all to the cumulative sum of the elements in *array*.

Second, even if the data at the final address of *sum* were meaningful, it would be inaccessible because the sumElements() never assigns a valid address to this null pointer. During each iteration of the for loop in sumElements(), the function simply increments to the address of a pointer that points to nothing but *array[indx]*sizeof(int)* bytes that simply gives us a pointer with a new invalid address.

What we intended sumElements() to do was to sequentially increment the value (not the address) of an integer variable by an amount *array[indx]* during each iteration of the for loop. We then want this function to return either the final value of this integer or a valid pointer to this integer; i.e., a pointer that can be dereferenced to access the value of the cumulative sum of the elements in *array*.

This gives us two alternative ways of correcting the pointer errors in our code. One way is to:

- redefine sum as an integer (instead of a pointer to an integer), then
- redefine sumElements() as a function returning an integer (instead of returning a pointer to an integer)

```
#include <stdio.h>
#include <stdlib.h>

int i, j, indx, sflag;
int sum;
float average;

void populateArray(int nelem_in_array, int *array);
void sortArray(int nelem_in_array, int *array, int (*fcn)(int sortFlag, int n, int m));
void exchange(int *elem1, int *elem2);
int ascending(int sortFlag, int n, int m);
int descending(int sortFlag, int n, int m);
int sumElements(int nelem_in_array, int *array);

int main(void)
{
  const int nelem = 10;

  int *array;
  array = (int *)malloc(nelem*sizeof(int));

  .   .   .

  .   .   .

/*  Compute the sum of the elements in the array  */
  sumElements(nelem, array);

/* Print sum of elements in the array  */
  printf("The sum of the %d elements in the array is: %d\n\n ", nelem, sum);

  free(array);

  return 0;
}

void populateArray(const int nelem_in_array, int *intArray)
{
  for (j = 0; j   {
    if ((j % 2) == 0)
    {
      *intArray = 10;
    }
    else
    {
      *intArray = 100;
    }

    intArray += 1;
  }
}

  .   .   .

  .   .   .

int sumElements(int nelem_in_array, int *array)
{
  sum = 0;

  for (indx = 0; indx   {
    sum += array[indx];
  }
```

```
  return sum;
}
```

The second way to correct the errors in our code is to:

- initialize the pointer *sum* to point to a valid integer address,
- set the initial value of the integer pointed to by *sum* equal to zero,
- recode the for loop to increment the value pointed to by *sum,* and
- recode the printf statement on line 56 of our code to access the integer value, **sum,* pointed to by *sum*.

```
#include <stdio.h>
#include <stdlib.h>

int i, j, indx, sflag;
int *sum;
float average;

void populateArray(int nelem_in_array, int *array);
void sortArray(int nelem_in_array, int *array, int (*fcn)(int sortFlag, int n, int m));
void exchange(int *elem1, int *elem2);
int ascending(int sortFlag, int n, int m);
int descending(int sortFlag, int n, int m);
int *sumElements(int nelem_in_array, int *array);

int main(void)
{
  const int nelem = 10;

  int *array;
  array = (int *)malloc(nelem*sizeof(int));

  .   .   .

  .   .   .

/*  Compute the sum of the elements in the array  */
  sumElements(nelem, array);

/* Print sum of elements in the array  */
  printf("The sum of the %d elements in the array is: %d\n\n ", nelem, *sum);

  free(array);

  return 0;
}

void populateArray(const int nelem_in_array, int *intArray)
{
  for (j = 0; j < nelem_in_array; j++)
  {
    if ((j % 2) == 0)
    {
      *intArray = 10;
    }
    else
    {
      *intArray = 100;
    }

    intArray += 1;
  }
}

  .   .   .

  .   .   .

int *sumElements(int nelem_in_array, int *array)
{
```

```
   int k = 0;

   sum = &k;

  for (indx = 0; indx < nelem_in_array; indx++)
  {
    *sum += array[indx];
  }

  return sum;
}
```

After making the corrections according to either of the above two alternatives, our program gives us the correct results.

```
$ gcc -g -o pointerError3 pointerError3.c

$ gdb pointerError3

(gdb) run
Starting program: pointerError3
The sequence of elements in the unsorted array are: [ 10 100 10 100 10 100 10 100 10 100 ]

The sequence of elements in the sorted array are: [ 10 10 10 10 10 100 100 100 100 100 ]

The sum of the 10 elements in the array is: 550

Program exited normally.
```

Problem 4. sneakyError.c

In this lesson's sample program, we debugged a code that contained a number of different pointer errors. By the end of the lesson, it certainly looked as though we had successfully debugged the entire program because it gave the correct answers when it was run. Unfortunately, when it comes to pointer bugs, looks can definitely be deceiving. Running sneakyError.c in GDB produces the following output:

```
$ gcc -g -o sneakyError sneakyError.c

$ gdb sneakyError

(gdb) run
Starting program: sneakyError
The sequence of elements in the unsorted array are: [ 10 1000 10 1000 10 1000 10 1000 10 1000 ]

The sequence of elements in the sorted array are: [ 10 10 10 10 10 1000 1000 1000 1000 1000 ]

The average value of the array elements is: 505.000000

The frequencies of the elements in the sorted array are:
[ 1.500000 1.500000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ]

The sum of the element frequencies is: 3.000000

Program exited normally.
```

Looking at the output, we see that it is correct until we get to the results of the frequency analysis, which seem to be way off track. Not only is the sum of the individual frequencies greater than 1.0, but the individual frequencies for the different element values (10 and 1000) are both greater than 1.0. Although the result for the sum of the element frequencies (2.60) is rather absurd (since it is greater than 1.0), the actual calculation is correct given the incorrect values for the individual element frequencies (1.10 and 1.50). Therefore, the error does not appear to be located in the function that computes this sum,

```
float *fqsum(int nelem_in_array, float *freq)
```

So a good initial guess would be that there is a coding error (or errors) in the function that computes the frequency of each distinguishable value in the sorted array,

```
float *computeFreq(int nelem_in_array, int *array, float *freq)
```

If we list the lines of our code in sneakyError.c, we see that the function computeFreq() is called in line 65.

```
(gdb) list
12      int descending(int sortFlag, int n, int m);
13      float getAverage(int nelem_in_array, int *array);
14      float *computeFreq(int nelem_in_array, int *array, float *freq);
15      float *fqsum(int nelem_in_array, float *freq);
16
17      int main(void)
18      {
19        const int nelem = 10;
(gdb)
20
21        int *array;
22        float *freq;.  .  .

.  .  .

61      /*  Print average */
62        printf("The average value of the array elements is: %f\n\n ", getAverage(nelem, array));
63
64      /* Compute the frequency of each element in the array  */
65        computeFreq(nelem, array, freq);
66
67      /* Print frequencies */
```

Therefore, we will set a breakpoint at line 65 of our code and step into and through the function computeFreq() to see if we can find our coding errors.

```
(gdb) b 65
Breakpoint 1 at 0x8048544: file sneakyError.c, line 65.
(gdb) run
Starting program: sneakyError
The sequence of elements in the unsorted array are: [ 10 1000 10 1000 10 1000 10 1000 10
1000 ]

The sequence of elements in the sorted array are: [ 10 10 10 10 10 1000 1000 1000 1000 1000 ]

The average value of the array elements is: 505.000000

Breakpoint 1, main () at sneakyError.c:65
65        computeFreq(nelem, array, freq);
(gdb) s
computeFreq (nelem_in_array=10, array=0x8049ce0, freq=0x8049d10) at sneakyError.c:162
162       int k = 0;
(gdb) s
164       numfreq = (float *)malloc(nelem_in_array*sizeof(float));
(gdb) s
165       *numfreq = 1.0;
(gdb) s
167       *(freq + k) += (float)(*(numfreq + k))/nelem_in_array;
(gdb) s
169       for (indx = 0; indx (gdb) s
173           *(numfreq + k) += 1;
 (gdb) define pvar
Type commands for definition of "pvar".
End with a line saying just "end".
>printf "indx = %d\n k = %d\n *(array + indx) = %d\n *(array + (indx + 1)) = %d\n *(freq + k) =
%f\n *(numfreq + k) = %f\n", indx, k, *(array + indx), *(array + (indx + 1)), *(freq + k), *(numfreq +
k)
>end
(gdb) pvar
indx = 0
 k = 0
 *(array + indx) = 10
 *(array + (indx + 1)) = 10
 *(freq + k) = 0.100000
 *(numfreq + k) = 1.000000
(gdb) s
```

```
175         }
(gdb) s
176         else
(gdb) s
169       for (indx = 0; indx (gdb) s
173           *(numfreq + k) += 1;
(gdb) pvar
indx = 1
 k = 0
 *(array + indx) = 10
 *(array + (indx + 1)) = 10
 *(freq + k) = 0.300000
 *(numfreq + k) = 2.000000
(gdb) s
175         }
(gdb) s
176         else
(gdb) s
169       for (indx = 0; indx (gdb) s
173           *(numfreq + k) += 1;
(gdb) pvar
indx = 2
 k = 0
 *(array + indx) = 10
 *(array + (indx + 1)) = 10
 *(freq + k) = 0.600000
 *(numfreq + k) = 3.000000
(gdb) s
175         }
(gdb) s
176         else
(gdb) s
169       for (indx = 0; indx (gdb) s
173           *(numfreq + k) += 1;
(gdb) pvar
indx = 3
 k = 0
 *(array + indx) = 10
 *(array + (indx + 1)) = 10
 *(freq + k) = 1.000000
 *(numfreq + k) = 4.000000
(gdb) s
175         }
(gdb) s
176         else
(gdb) s
169       for (indx = 0; indx (gdb) s
173           *(numfreq + k) += 1;
(gdb) pvar
indx = 4
 k = 0
 *(array + indx) = 10
 *(array + (indx + 1)) = 1000
 *(freq + k) = 1.500000
 *(numfreq + k) = 5.000000
```

Looking carefully at the results generated for the frequency of occurrence of each numerical value in *array*, **(freq + k)*, clearly tells us what the coding error is. During the first five iterations of the for loop in computeFreq(), *indx = 0, 1, 2, 3, 4*, we see that the numerical value of the corresponding elements in *array* is 10,

```
*(array + indx) = 10
```

after which the value of the array element changes to 1000,

```
*(array + (indx + 1)) = 1000 ,  for indx = 4
```

That is, the value 10 is repeated for the first five elements of array. If a given element value has occurred n times within an array containing N total elements, the frequency of occurrence, f, will be $f = n / N$. If the same value then occurs again, the change in frequency of occurrence will be

```
( f * N + 1) / N ) ? f  =  1 / N
```

In the current program where we have an array containing a total of 10 elements (*N = 10*), then during the first five iterations of the for() loop in computeFreq(), the frequency of occurrence should change by

```
1 / N = 0.1
```

each time the element value 10 is repeated. However, what we find instead is that the frequency of occurrence of the element value 10 changes by (0.1 − 0.0) = 0.1, (0.3 − 0.1) = 0.2, (0.6 − 0.3) = 0.3, (1.0 − 0.6) = 0.4, and (1.5 − 1.0) = 0.5 for *indx*= 0, 1, 2, 3, and 4, respectively. Obviously, we have included some factor or term in our code for computing the frequency that should not be there. If we take a look at our code for computing frequency (line 70),

```
*(freq + k) += (float)(*(numfreq + k))/nelem_in_array;
```

we can see that, whenever an element value is repeated, this code will give us a change in frequency of

```
(float)(*(numfreq + k))/nelem_in_array  = *(numfreq + k) / 10  =  [ *(numfreq + k) ] * 0.1
```

Consequently, we see that the extra factor that we have included in our code (the factor that should not be there) is

```
*(numfreq + k)
```

Instead, this factor should be 1.0. Therefore, we can correct the bug in our code by eliminating **(numfreq + k)* in our function computeFreq()

```
float *computeFreq(int nelem_in_array, int *array, float *freq)
{
  int k = 0;
  *freq += 1.0/nelem_in_array;

  for (indx = 0; indx   {

    if (*(array + (indx + 1)) == *(array + indx))
    {
      *(freq + k) += 1.0/nelem_in_array;
    }
    else
    {
      k++;
      *(freq + k) += 1.0/nelem_in_array;
    }
  }
  return freq;
}
```

After making this correction in our code, our program gives us the correct results.

```
$ gcc −g −o sneakyError sneakyrError.c

$ gdb sneakyError

(gdb) run
Starting program: sneakyError
The sequence of elements in the unsorted array are: [ 10 1000 10 1000 10 1000 10 1000 10 1000 ]

The sequence of elements in the sorted array are: [ 10 10 10 10 10 1000 1000 1000 1000 1000 ]

The average value of the array elements is: 505.000000

The frequencies of the elements in the sorted array are:
[ 0.500000 0.500000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ]

The sum of the element frequencies is: 1.000000

Program exited normally.
```

Problem 5. arrayCopyBug.cc

Running arrayCopyBug.cc in GDB generates the following output:

```
$ g++ -g -o arrayCopyBug arrayCopyBug.cc

$ gdb arrayCopyBug

(gdb) run
Starting program: arrayCopyBug

The elements in the original array that are being copied to the new array are:

     [  1  2  3  4  5  ]

The elements copied in the new array that were copied from the original array are:

     [  5  9.40681e-311  0  0  0  ]


Program received signal SIGSEGV, Segmentation fault.
0x420744b0 in _int_free () from /lib/tls/libc.so.6
```

Looking at these results, we see that something has clearly gone wrong while copying of the elements from the original array to the new array. Our initial guess might be that there is a bug somewhere in the function that copies array elements from one array to another. In arrayCopyBug.cc we are using a function named copyArray() to copy arrays. Let us take a look at this function to see exactly what it is doing.

```
double *copyArray(int nelem_in_array, double *array2Bcopied, double *copied2Array)
{
  int index;
  double *ptr;

  ptr = copied2Array;

  for (index = 0; index   {
    *ptr = array2Bcopied[index];

    if (index < (nelem_in_array - 1))
    {
      ptr++;
    }
  }
   return ptr;
}
```

The function copyArray(), returns a pointer to a double and takes an integer and two double arrays as arguments. The second argument is the array whose elements are being copied and the third argument is the array to which the elements are copied. In this function we first declare a pointer to a double and then initialize this pointer to point to the first element in the array to which we are copying, *copied2Array*.

```
double *ptr;
ptr = copied2Array;
```

We then set the value located at the address *ptr*, i.e., the value located at the address of the first element in *copied2Array*, equal to the value of the first element in *array2Bcopied*,

```
*ptr = array2Bcopied[0];
```

We then increment the pointer to point to the second element in *copied2Array*, which is sizeof(double) bytes downstream from the first element in this array,

```
ptr++;
```

We then repeat these steps until we have reached the end of *array2Bcopied* and then return the address of *copied2Array* to the calling function copyArray().

If we list the lines of code in our program we see that our function copyArray() is called on line 76.

```
(gdb) list
43      };
44
```

```
45
46      int main(int argc, char *argv[])
47      {
48        int nelem, index;
49        double *origArray, *newArray;
50
51      // Instantiate a pointer to an arrayCopier object
52        arrayCopier *cpy = new arrayCopier;
(gdb)
53
54      // Set number of elements in array
55        nelem = 5;
56
57      // Dynamically allocate memory for each array
58        origArray = new double[nelem];
59        newArray = new double[nelem];
60
61      // Initialize array that is to be copied
62        for (index = 0; index (gdb)
63        {
64          origArray[index] = (index + 1);
65        }
66
67      // Print the elements of the array being copied
68        cout << '\n';
69        cout << "The elements in the original array that are being copied to the new array are: "
<<'\n' << '\n';
70        cout << "     [  ";
71        cpy -> printArray(nelem, origArray);
72        cout << "]" << '\n';
(gdb)
73        cout << '\n';
74
75      // Copy the original array to the new array
76        newArray = (cpy -> copyArray(nelem, origArray, newArray));
77
```

If we set a breakpoint at line 76, we can step into and through this function to try to determine exactly what the coding error is in this function.

```
(gdb) b 76
Breakpoint 1 at 0x80487be: file arrayCopyBug.cc, line 76.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: arrayCopyBug

The elements in the original array that are being copied to the new array are:

     [  1  2  3  4  5  ]

Breakpoint 1, main (argc=1, argv=0xbfffce54) at arrayCopyBug.cc:76
76        newArray = (cpy -> copyArray(nelem, origArray, newArray));
(gdb) s
arrayCopier::copyArray(int, double*, double*) (this=0x8049ec0,
    nelem_in_array=5, array2Bcopied=0x8049ed0, copied2Array=0x8049f00)
    at arrayCopyBug.cc:18
18            ptr = copied2Array;
(gdb) s
20            for (index = 0; index
```

In this code we are using the pointer, *ptr*, to do two different things. First, we are using *ptr* to sequentially point to the elements in *copied2Array*. Then we are using *ptr* to assign a given element in *copied2Array* the same value as the element with the corresponding index in *array2Bcopied*. Therefore, we print out both the address being pointed to by *ptr* and the value we obtain when we dereference *ptr*. We also print out the actual value of each element copied to *copied2Array* to see if incorrect values are being copied to this array.

```
(gdb) define pvar
Type commands for definition of "pvar".
```

```
End with a line saying just "end".
>printf "index = %d\n ptr = %d\n *ptr = %d\n copied2Array[index] = %d\n", index, ptr, *ptr,
copied2Array[index]
>end
(gdb) s
22              *ptr = array2Bcopied[index];
(gdb) s
24              if (index (gdb) s
26                ptr++;
(gdb) pvar
index = 0
ptr = 134520576
*ptr = 1
copied2Array[index] = 1
```

For some reason the address pointed to by *ptr* is being printed as an integer. If we want to print it in hexadecimal format, we can use the GDB command.

```
p /x ptr
```

Although we can redefine our print macro *pvar* to print *ptr* in hex format, here we simply go ahead and print the hex address of *ptr* on a separate line.

```
(gdb) p /x ptr
(gdb) p /x ptr $2 = 0x8049f00 (gdb) s 20 for (index = 0; index < nelem_in_array; index++) (gdb) s 22 *ptr =
array2Bcopied[index]; (gdb) s 24 if (index < (nelem_in_array - 1)) (gdb) s 26 ptr++; (gdb) pvar index = 1 ptr =
134520584 *ptr = 2 copied2Array[index] = 2 (gdb) p /x ptr $3 = 0x8049f08 (gdb) s 20 for (index = 0; index <
nelem_in_array; index++) (gdb) s 22 *ptr = array2Bcopied[index]; (gdb) s 24 if (index < (nelem_in_array - 1)) (gdb)
s 26 ptr++; (gdb) pvar index = 2 ptr = 134520592 *ptr = 3 copied2Array[index] = 3 (gdb) p /x ptr $4 = 0x8049f10 (gdb)
s 20 for (index = 0; index < nelem_in_array; index++) (gdb) s 22 *ptr = array2Bcopied[index]; (gdb) s 24 if (index <
(nelem_in_array - 1)) (gdb) s 26 ptr++; (gdb) pvar index = 3 ptr = 134520600 *ptr = 4 copied2Array[index] = 4 (gdb)
p /x ptr $5 = 0x8049f18 (gdb) s 20 for (index = 0; index < nelem_in_array; index++) (gdb) s 22 *ptr =
array2Bcopied[index]; (gdb) s 24 if (index < (nelem_in_array - 1)) (gdb) s 20 for (index = 0; index < nelem_in_array;
index++) (gdb) pvar index = 4 ptr = 134520608 *ptr = 5 copied2Array[index] = 5 (gdb) p /x ptr $6 = 0x8049f20 =
0x8049f00
(gdb) s
20            for (index = 0; index < nelem_in_array; index++)
(gdb) s
22              *ptr = array2Bcopied[index];
(gdb) s
24              if (index < (nelem_in_array - 1))
(gdb) s
26                ptr++;
(gdb) pvar
index = 1
ptr = 134520584
*ptr = 2
copied2Array[index] = 2
(gdb) p /x ptr
 = 0x8049f08
(gdb) s
20            for (index = 0; index (gdb) s
22              *ptr = array2Bcopied[index];
(gdb) s
24              if (index < (nelem_in_array - 1))
(gdb) s
26                ptr++;
(gdb) pvar
index = 2
ptr = 134520592
*ptr = 3
copied2Array[index] = 3
(gdb) p /x ptr
 = 0x8049f10
(gdb) s
20            for (index = 0; index < nelem_in_array; index++)
(gdb) s
22              *ptr = array2Bcopied[index];
(gdb) s
24              if (index < (nelem_in_array - 1))
```

```
(gdb) s
26                ptr++;
(gdb) pvar
index = 3
ptr = 134520600
*ptr = 4
copied2Array[index] = 4
(gdb) p /x ptr
 = 0x8049f18
(gdb) s
20            for (index = 0; index < nelem_in_array; index++)
(gdb) s
22              *ptr = array2Bcopied[index];
(gdb) s
24              if (index < (nelem_in_array - 1))
(gdb) s
20            for (index = 0; index < nelem_in_array; index++)
(gdb) pvar
index = 4
ptr = 134520608
*ptr = 5
copied2Array[index] = 5
(gdb) p /x ptr
 = 0x8049f20
```

As we see from the output for *pvar* for each of the 5 values of *index* in the for> loop, the array *copied2Array,*

```
[ 1  2  3  4  5 ]
```

is an exact copy of the array *array2Bcopied*. Consequently, we know that the error in our function copyArray() is not in the section of code that actually copied the individual elements of an array. The only other thing that this function does is return the array once it has been copied. Since the copied array itself is correct, there must be an error in the way the array is being returned to the calling function in line 76. Therefore, we continue stepping through the function copyArray() to see exactly what is being returned to the calling function.

```
(gdb) s
30            return ptr;
(gdb) pvar
index = 5
ptr = 134520608
*ptr = 5
copied2Array[index] = 0
(gdb) p /x ptr
 = 0x8049f20
(gdb) s
31          }
(gdb) pvar
index = 5
ptr = 134520608
*ptr = 5
copied2Array[index] = 0
(gdb) p /x ptr
 = 0x8049f20
```

As we can see, the function is returning a pointer that points to a value located at the address 0x8049f20. Since we want to return the entire array that we just copied, then we want this address to be the address of the first element in the five-element array being returned to the calling function in line 76. However, if we look at the address of the address of the first element of the array being passed to our function (the array into which the elements of *array2Bcopied* are being copied), we see that the address 0x8049f20 is not the address of the first element of the array that was passed to our function,

```
arrayCopier::copyArray(int, double*, double*) (this=0x8049ec0, nelem_in_array=5,
    array2Bcopied=0x8049ed0, copied2Array=0x8049f00) at arrayCopyBug.cc:18
```

Instead, the address of the first element in the array being copied to is 0x8049f00. We can confirm that this is the address of the beginning of the array that is returned from the function copyArray() by looking at the address of the first element in the array *newArray* that our program prints out.

```
(gdb) s
main (argc=1, argv=0xbfffe8d4) at arrayCopyBug.cc:79
79        cout << '\n';
(gdb) n

80        cout
```

2.8 Dynamic Memory Allocation Errors

2.8.1 Dynamic Memory Allocation Errors

Introduction

Static memory allocation has been the traditional method for assigning memory to an array for quite some time. In this method, memory is allocated at the beginning of the program and does not change for the duration of the program. One problem with this method is that the size of the array may not be known at the initial declaration. Sometimes the array size will be input from a file or user prompt or perhaps be calculated in the program. In these cases, memory cannot be statically allocated to the array.

Dynamic memory allocation is the ability to assign memory to an array at any point in a program. In dynamic memory allocation, the memory size can be determined from sources such as a file, user prompt, or calculation and memory of the appropriate amount is connected to the array name at the point in the code when it is needed. Unlike static memory allocation, this method does not tie up all the memory allocated to it throughout the duration of the program but rather just from the point at which it is allocated. In addition, the size declared is the exact size needed for the array and the memory can be released when the array is no longer needed.

Several high-level languages provide the ability to dynamically allocate memory. These languages typically provide this capability through built in routines that can be called when memory needs to be allocated. The languages most commonly used for dynamic memory allocation are C, C++, and Fortran 90.

As with any programming technique, when dynamic memory allocation is performed incorrectly it results in a variety of errors. Also, how the error manifests itself depends on which compiler, loader, and machine is used. In one situation, the compiler could catch the programming error and in another an executable could be made. In the latter case, when the buggy code is executed either a run-time error occurs or incorrect results are produced.

Two of the most common programming errors in dynamic memory allocation are trying try to allocate too much memory and not allocating the memory at all. Also common are allocating a zero size, the incorrect size, and for multi-dimensional arrays not allocating enough memory for all the dimensions. This is by no means an exhaustive list of the errors that can occur but it should give you an idea of the most common.

Objectives

In this lesson, you will learn how to identify and debug a dynamic memory allocation error in a serial Fortran 90 program. A sample program is provided that you can copy and debug along with the lesson. A programming exercise is included at the end of the lesson to test what you learned.

2.8.2 Sample Program with a Memory Allocation Error

We now give a sample program written in Fortran 90 to illustrate a memory allocation. This sample is a serial program; for a parallel program example written in C see the lesson titled "Dynamic Memory Allocation Errors" in the "Debugging Parallel Code" section of this tutorial.

```
PROGRAM dynamic
  REAL,DIMENSION(:),ALLOCATABLE::mobile
  REAL,DIMENSION(12)::stiff
  INTEGER::total,addem
  stiff=(/(i,i=1,12)/)
  mobile=2*stiff
  total=SUM(stiff)
  print *,"total=",total
  addem=SUM(mobile)
  print *,"addem=",addem
END PROGRAM dynamic
```

The program works with two arrays: one static array named *stiff* and one dynamic array named *mobile*. First, the static array is filled with values and the sum of those values is calculated and outputted. Then, each element of the dynamic array is assigned twice the value of the static array in the line:

mobile=2*stiff

Finally, the sum of the elements of the dynamic array is calculated and printed out.

Compiling and Executing

Our sample program was run on a Linux cluster of Intel P4 processor chips using the native Intel compiler named "ifort". The following output was produced:

```
piv$ ifort none3.f90
piv$ ./a.out
total=          78
addem=           0
```

Although this program has a *serious* allocation error, it compiles and loads and an executable is produced. However, by looking at the values printed we see that the sum for the static array *stiff* is correct but the sum of the elements of the dynamic array *mobile* in *addem* is incorrect. The sum printed out is 0 when it should be 156.

2.8.3 Debugging the Program

As we saw in the output from the sample program, the dynamic array *mobile* did not receive the values that we expected. So the first debugging step we plan to take is to set a breakpoint after *mobile* is assigned values so we can can see the values assigned. Since the elements in *mobile* are calculated from the values in the serial array *stiff*, we will also check those values by setting an earlier breakpoint. (Checking the values of *stiff* is being overly careful since the summation of the elements in *stiff* gave the correct result. But when it comes to debugging, being overly careful is not necessarily a bad trait.)

We use the GNU debugger gdb for our debugging session. First, we begin the session and then use the 'list' command to view the source code so we can decide where we should place breakpoints.

```
piv$ ifort -g -o none3 none3.f90
piv$ gdb none3
GNU gdb Red Hat Linux (6.1post-1.20040607.17rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".

(gdb) list
1       PROGRAM dynamic
2         REAL,DIMENSION(:),ALLOCATABLE::mobile
3         REAL,DIMENSION(12)::stiff
4         INTEGER::total,addem
5         stiff=(/(i,i=1,12)/)
6         mobile=2*stiff
7         total=SUM(stiff)
8         print *,"total=",total
9         addem=SUM(mobile)
10        print *,"addem=",addem
(gdb)
```

Since the value of *stiff* is set in line 5, to check if it was initialized correctly we set a breakpoint at line 6:

```
(gdb) break 6
Breakpoint 1 at 0x8049ddc: file none3.f90, line 6.
(gdb) run
Starting program: /a/dje/debug/alloc/none3
[Thread debugging using libthread_db enabled]
[New Thread 16384 (LWP 29113)]
[Switching to Thread 16384 (LWP 29113)]

Breakpoint 1, dynamic () at none3.f90:6
6         mobile=2*stiff
Current language:  auto; currently fortran
```

```
(gdb) print stiff
 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
(gdb)
```

As expected, this output shows that the values in *stiff* are correct. Similarly, we set a breakpoint at line 7 to see the values for *mobile*:

```
(gdb) break 7
Breakpoint 2 at 0x8049e39: file none3.f90, line 7.
(gdb) cont
Continuing.

Breakpoint 2, dynamic () at none3.f90:7
7         total=SUM(stiff)
(gdb) print mobile
(gdb) break 7 Breakpoint 2 at 0x8049e39: file none3.f90, line 7. (gdb) cont Continuing. Breakpoint 2, dynamic () at
none3.f90:7 7 total=SUM(stiff) (gdb) print mobile $2 = () (gdb) = ()
(gdb)
```

We can see from these results that the dynamic array *mobile* not only does not have the expected values, it has no values. This is because memory was not allocated for *mobile* during the array syntax calculation on line 6 and thus, it does not have a size. Memory must always be directly allocated to a dynamic array; it is not done automatically.

2.8.4 The Corrected Program

In Fortran 90, a dynamic array must be given the attribute ALLOCATABLE at its declaration. In the sample program that was done correctly in line 2. But this declaration, by itself, is not enough. You must also use the routine ALLOCATE(array-name(size))to assign actual memory to the dynamic array so the values can be put in to those new memory locations.

Below is the corrected sample program. The program is fixed with the addition of only one line that calls the **ALLOCATE** routine:

```
PROGRAM dynamic
  REAL,DIMENSION(:),ALLOCATABLE::mobile
  REAL,DIMENSION(12)::stiff
  INTEGER::total,addem
  stiff=(/(i,i=1,12)/)
  ALLOCATE(mobile(12))
  mobile=2*stiff
  total=SUM(stiff)
  print *,"total=",total
  addem=SUM(mobile)
  print *,"addem=",addem
END PROGRAM dynamic
```

Although this error is rather obvious, it occurs very often. So often that there is another Fortran 90 function, ALLOCATED(array-name), that returns true if the memory for the array has been allocated and false otherwise. Therefore, it must be a rather common error for such a function to be made available.

2.8.5 Programming Exercise

Write a Fortran 90 program in which a two-dimensional array is dynamically allocated. Prompt the user to enter in the number of rows and number of columns required. If the array was allocated correctly, use the F90 intrinsic functions LBOUND, UBOUND, and SIZE to determine the characteristics of the new array. Print out the values returned by LBOUND, UBOUND, and SIZE. Run your program several times inputting different values for the number of rows and the number of columns.

2.8.6 Programming Exercise

The following Fortran 90 program is the solution for the exercise:

```
PROGRAM dynamic
  REAL,DIMENSION(:,:),ALLOCATABLE::grid
  INTEGER::code,m,n
  PRINT *,"Enter the rows and columns of the array"
  READ *,m,n
```

```
  ALLOCATE(grid(m,n),STAT=code)
  IF (code > 0 ) THEN
    PRINT *,"Error code is ",code
    PRINT *,"ALLOCATED=",ALLOCATED(grid)
  ELSE
    PRINT *,"Good code is ",code
    PRINT *, 'Bounds and Size of grid = ',LBOUND(grid),UBOUND(grid),SIZE(grid)
    PRINT *,"ALLOCATED=",ALLOCATED(grid)
  END IF
END PROGRAM dynamic
```

Notice that we used a new argument, STAT, in the ALLOCATE routine. If the allocation occurs without any problems, the STAT variable returns 0. If there is a problem, the STAT variable returns a positive integer.

Below, we show a session run on a UNIX machine in which the above solution code is executed a number of times. Each time the code is run, different values are input for the number of rows and number of columns.

```
775:oscb$ ./dynamic
 Enter the rows and columns of the array
5 8
 Good code is  0
 Bounds and Size of grid =  2*1,  5,  8,  40
 ALLOCATED= T
776:oscb$ ./dynamic
 Enter the rows and columns of the array
40 3
 Good code is  0
 Bounds and Size of grid =  2*1,  40,  3,  120
 ALLOCATED= T
777:oscb$ ./dynamic
 Enter the rows and columns of the array
17 1
 Good code is  0
 Bounds and Size of grid =  2*1,  17,  1,  17
 ALLOCATED= T
778:oscb$ ./dynamic
 Enter the rows and columns of the array
1 54
 Good code is  0
 Bounds and Size of grid =  3*1,  2*54
 ALLOCATED= T
779:oscb$ ./dynamic
 Enter the rows and columns of the array
9 0
 Good code is  0
 Bounds and Size of grid =  2*1,  9,  2*0
 ALLOCATED= T
780:oscb$ ./dynamic
 Enter the rows and columns of the array
0 345
 Good code is  0
 Bounds and Size of grid =  2*1,  0,  345,  0
 ALLOCATED= T
781:oscb$ ./dynamic
 Enter the rows and columns of the array
6 -13
 Good code is  0
 Bounds and Size of grid =  2*1,  6,  2*0
 ALLOCATED= T
782:oscb$ ./dynamic
 Enter the rows and columns of the array
-5 8
 Good code is  0
 Bounds and Size of grid =  2*1,  0,  8,  0
 ALLOCATED= T
783:oscb$ ./dynamic
 Enter the rows and columns of the array
30000  40000
 Error code is  1205
```

```
 ALLOCATED= F
784:oscb$
```

You can see in the execution sequence that ALLOCATE is a very robust command. It can create one-dimensional arrays and can even handle zero or a negative input to make zero-sized arrays. Only in the last run did memory allocation fail. This failure was due to the values for the number of rows and the number of columns being too high and not enough memory could be found for the large array requested.

2.9 Debugging Integer Divide Errors

2.9.1 Debugging Integer Divide Errors

Introduction

An integer-divide error occurs when an integer-divide is performed where a floating-point divide is required. Although this bug is rather simple, it can wreak havoc with the expected results. This error is simple to fix, but it can sometimes be difficult to track down. Depending upon the exact nature of the error, it has the potential to create problems throughout the remainder of the code as we show in the sample program.

Objectives

In this lesson you will learn what an integer-divide bug is and the effects one can have on your code. A sample program is provided and the dbx debugger is used to show how to locate the problem.

2.9.2 Sample Code with an Integer-Divide Error

The following Fortran code integrates the function cos(x) over the interval zero to pi using the trapezoidal rule. The result should be identically equal to zero. For demonstration purposes, only 10 intervals (11 endpoints) are used.

```
program trapezoid

! integrate sin(x) over the interval 0 to pi
! using trapezoid rule

implicit none

integer :: i, n = 11, nm
real :: pi, f1, f2, x1, x2, xint

write(*, "('  i        x       f(x)    int(f(x))'/&

          &'  1      0.00      1.00      0.00' )" )

pi = acos(-1.0)
nm = n-1
xint = 0.0
x1 = 0.0

do i = 2, n
   x2 = ((i-1)/nm)*pi
   f1 = cos(x1)
   f2 = cos(x2)
   xint = xint + 0.5*(f1+f2)*(x2-x1)
   x1 = x2

   write(*, '(i3, 3(f10.2))') i, x2, f2, xint

enddo

end program trapezoid
```

The sample code was compiled and run, and gave the following output:

```
  i        x       f(x)    int(f(x))

  1      0.00      1.00      0.00

  2      0.00      1.00      0.00
```

```
  3      0.00      1.00      0.00

  4      0.00      1.00      0.00

  5      0.00      1.00      0.00

  6      0.00      1.00      0.00

  7      0.00      1.00      0.00

  8      0.00      1.00      0.00

  9      0.00      1.00      0.00

 10      0.00      1.00      0.00

 11      3.14     -1.00      0.00
```

The four columns contain the endpoint number, the independent variable *x*, the dependent variable *cos(x)*, and the integral of the function from the left endpoint to the current point. The good news is that the final value of the integral is exactly zero, as expected. The bad news is that most of the other values in the output file are clearly incorrect. Next we examine the code using dbx.

2.9.3 Debugging the Integer-Divide Error

Using dbx, our first step is to re-compile the code with the '-g' flag, which creates a symbol table. Under AIX, the compile command is

```
% xlf90 -o trapezoid -g -qsuffix=f=f90 trapezoid.f90
```

The first row of output data is not computed, but simply the result of a write statement. The first incorrect computed number is the independent variable value *x* at *i=2*, so we concentrate on that value.

The first step is to start dbx in the usual way:

```
% dbx trapezoid
Type 'help' for help.
reading symbolic information ...
(dbx)
```

Then we list the code to get the line numbers. Since the code is so short, we can list all of it as follows:

```
(dbx) l 1,27
    1   program trapezoid
    2
    3   ! integrate sin(x) over the interval 0 to pi
    4   ! using trapezoid rule
    5
    6   implicit none
    7
    8   integer :: i, n = 11, nm
    9   real :: pi, f1, f2, x1, x2, xint
   10
   11   write(*, "('  i        x       f(x)    int(f(x))'/&
   12             &'  1      0.00      1.00      0.00' )" )
   13
   14   pi = acos(-1.0)
   15   nm = n-1
   16   xint = 0.0
   17   x1 = 0.0
   18   do i = 2, n
   19      x2 = ((i-1)/nm)*pi
   20      f1 = cos(x1)
   21      f2 = cos(x2)
   22      xint = xint + 0.5*(f1+f2)*(x2-x1)
   23      x1 = x2
   24      write(*, '(i3, 3(f10.2))') i, x2, f2, xint
   25   enddo
```

```
   26
   27   end program trapezoid
(dbx)
```

We want to examine the value of *x2*, which we have seen to be incorrect the first time through the loop. First, set a breakpoint at line 20, after *x2* has been computed, and run the code to that point.

```
(dbx) stop at 20
[1] stop at 20

(dbx) run
  i        x       f(x)    int(f(x))
  1     0.00      1.00      0.00
[1] stopped in trapezoid at line 20
   20      f1 = cos(x1)
(dbx)
```

The expected value of *x2* is 0.1*pi=0.31. Examine the actual value:

```
(dbx) p x2
0.0
```

Something is clearly incorrect here. Print the values used to calculate *x2*:

```
(dbx) p i,nm,pi
2 10 3.14159274
(dbx)
```

All these numbers are correct. Examining the source line where *x2* is computed,

```
(dbx) l 19
   19      x2 = ((i-1)/nm)*pi
(dbx)
```

shows the problem. All the variables on the right-hand side have the correct values, but the resulting calculation is incorrect. This leads us to suspect some kind of type mismatch, and we indeed see that we are performing integer arithmetic rather than the required floating-point arithmetic. This is why the output showed a value of zero for *x2* for every point up to point 11, where the correct value of pi was shown. Changing line 19 to

```
x2 = (real(i-1)/real(nm))*pi
```

and recompiling and running yields the correct result:

```
% trapezoid
  i        x       f(x)    int(f(x))
  1     0.00      1.00      0.00
  2     0.31      0.95      0.31
  3     0.63      0.81      0.58
  4     0.94      0.59      0.80
  5     1.26      0.31      0.94
  6     1.57      0.00      0.99
  7     1.88     -0.31      0.94
  8     2.20     -0.59      0.80
  9     2.51     -0.81      0.58
 10     2.83     -0.95      0.31
 11     3.14     -1.00      0.00
```

There is an additional caveat here: The debugger will not necessarily perform the same type conversions as the compiler. Going back to the version of the code with the bug, suppose we had chosen to print the value of the expression used to compute *x2*:

```
(dbx) p i,nm,pi
2 10 3.14159274
(dbx) p (i-1)/nm
0.10000000000000001
(dbx) p ((i-1)/nm)*pi
0.31415927410125732
```

These results are correct, even though the code computes an incorrect result for the same operations! You must exercise

caution when computing expressions within the debugger, since the debugger may not handle variable typing in the same way as the source code.

2.9.4 Integer Divide Debugging Exercise

In this exercise, we use a version of "Conway's game of life." An array is declared, and each element is set to zero or one ("dead" or "alive"). Each element is examined, and its state is evaluated based on the states of its neighbors. If exactly three neighbors are alive, the element's value is set to one. If two neighbors are alive, the element retains its previous value. If any other number of neighbors are alive, the element's value is set to zero. The process is then iterated for as many steps as desired. Over time, patterns can evolve, and it's interesting to observe the dynamics of the system. In an actual game of life, the array would be rendered graphically after each step. For our simple example, the number of live elements is simply printed after the specified number of iterations.

The Fortran code is shown below.

```
!---------------------------------------
program life
!---------------------------------------
! Conway game of life

! CONTAINS BUG FOR DEBUGGING EXAMPLE!
!---------------------------------------

implicit none
integer, parameter :: ni=200, nj=200, nsteps = 100
integer :: i, j, n, im, ip, jm, jp, nsum, isum
integer, dimension(0:ni,0:nj) :: old, new
real :: arand, nim2, njm2

!  initialize elements of "old" to 0 or 1

do j = 1, nj
   do i = 1, ni
      call random_number(arand)
      old(i,j) = nint(arand)
   enddo
enddo

nim2 = ni - 2
njm2 = nj - 2

! time iteration

time_iteration: &
do n = 1, nsteps

   do j = 0, nj
      do i = 0, ni

         ! periodic boundaries
         im = 1 + (i+nim2) - ((i+nim2)/ni)*ni
         jm = 1 + (j+njm2) - ((j+njm2)/nj)*nj
         ip = 1 + i - (i/ni)*ni
         jp = 1 + j - (j/nj)*nj

         ! for each point, add surrounding values
         nsum =  old(im,jp) + old(i,jp) + old(ip,jp) &
               + old(im,j )             + old(ip,j ) &
               + old(im,jm) + old(i,jm) + old(ip,jm)

         ! set new value based on number of "live" neighbors
         select case (nsum)
         case (3)
            new(i,j) = 1
         case (2)
            new(i,j) = old(i,j)
         case default
            new(i,j) = 0
```

```
         end select

      enddo
   enddo

   ! copy new state into old state
   old = new

enddo time_iteration

! write number of live points
print*, 'number of live points = ', sum(new)

end program life
```

When this code is compiled and run, we find that no elements have survived our iterations. Although in general this is possible to occur, it is known from experience that this system should not have died out so quickly. Use a debugger to find the problem.

2.9.5 Integer Divide Exercise Solution

In the Integer Divide lesson, an example was given where integer division by zero was encountered. In the current example, the problem is with unintentional mixed float and integer arithmetic. The variables *nim2* and *njm2* were unintentionally declared as reals rather then integers. The error was in the index calculation for the neighboring elements, causing the incorrect elementsto be interrogated. Correcting the declarations results in the code working properly. Below is the corrected code.

```
!----------------------
program life
!----------------------
! Conway game of life
!----------------------

implicit none
integer, parameter :: ni=200, nj=200, nsteps = 100
integer :: i, j, n, im, ip, jm, jp, nsum, isum, nim2, njm2
integer, dimension(1:ni,1:nj) :: old, new
real :: arand

!  initialize elements of "old" to 0 or 1

do j = 1, nj
   do i = 1, ni
      call random_number(arand)
      old(i,j) = nint(arand)
   enddo
enddo

nim2 = ni - 2
njm2 = nj - 2

! time iteration

time_iteration: &
do n = 1, nsteps

   do j = 1, nj
      do i = 1, ni

         ! periodic boundaries
         im = 1 + (i+nim2) - ((i+nim2)/ni)*ni
         jm = 1 + (j+njm2) - ((j+njm2)/nj)*nj
         ip = 1 + i - (i/ni)*ni
         jp = 1 + j - (j/nj)*nj

         ! for each point, add surrounding values
         nsum =  old(im,jp) + old(i,jp) + old(ip,jp) &
```

```
               + old(im,j )             + old(ip,j ) &
               + old(im,jm) + old(i,jm) + old(ip,jm)

         ! set new value based on number of "live" neighbors
         select case (nsum)
         case (3)
            new(i,j) = 1
         case (2)
            new(i,j) = old(i,j)
         case default
            new(i,j) = 0
         end select

      enddo
   enddo

   ! copy new state into old state
   old = new

enddo time_iteration

! write number of live points
print*, 'number of live points = ', sum(new)

end program life
```

3 Debugging Parallel Code

3.1 Debugging Parallel Code

In the following sections, you will learn several methods for debugging parallel code.

3.2 Checking Variable Values for Processes

3.2.1 Checking Variable Values for Processes

Introduction

The first step you are most likely to take to find a bug in your parallel program is the same step you are most likely to take for a serial program - examine variable values while executing your code. However, the method is slightly different for parallel programs since you need to examine variable values for multiple processes. In this lesson, we show how to do this using the pdbx debugger.

Objectives

In this lesson, you will learn how to examine variable values for multiple processes in a parallel program using the pdbx debugger. A sample program is provided that you can copy and debug along with the lesson. A programming exercise is included at the end of the lesson to test what you learned.

3.2.2 Sample Code with Multiple Processes

The following sample program, send_recv_example.c, is written for two processes: process 0 sends the content of an array to process 1 while concurrently, process 1 probes process 0 for a message, determines its size, dynamically allocates the memory, and then receives the message. There is no bug in this program, but we use it to demonstrate how to check for variable values in multiple processes.

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>

void main(int argc, char* argv[])
{
      double sdata[55];
      double rdata[55];
      int p, i, count, myid, n;
      MPI_Status status;

      MPI_Init(&argc, &argv);               /* starts MPI */
```

```
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get current process id */
      MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */

      if (myid == 0) {
        for(i=0;i
```

3.2.3 Checking Variable Values in the Sample Code

We start by compiling the program using the '-g' option and then running it using the pdbx debugger on an IBM pSeries 690 computer system running AIX:

```
twister:debugger/ANALYZE % pdbx send_recv_example -procs 2
pdbx Version 4, Release 1 -- Jul  7 2004 17:26:19

   0:Core file "
   0:" is not a valid core file (ignored)
   1:Core file "
   1:" is not a valid core file (ignored)
   0:reading symbolic information ...
   1:reading symbolic information ...
   1:[1] stopped in main at line 14 ($t1)
   1:   14         MPI_Init(&argc, &argv);               /* starts MPI */
   0:[1] stopped in main at line 14 ($t1)
   0:   14         MPI_Init(&argc, &argv);               /* starts MPI */
0031-504  Partition loaded ...

pdbx(all)
```

Note in the screen output above that the debugger automatically stops at the statement that invokes MPI_INIT. This is effected by pdbx's default initial breakpoint, which is set to be the first executable statement in the main program and not necessarily an MPI function call. This behavior can be reset to another arbitrary executable statement via the environment variable MP_DEBUG_INITIAL_STOP. Please consult the pdbx manpage for details.

Next, for convenience, we use the 'alias' command to label *m* to mean the "master" (or process 0) and *w* the "worker" (or process 1). We then partially list the program source on both processes. The program is then run and the processes stopped at their respective break points (line 21 for process 0 and line 24 for process 1).

```
pdbx(all) alias m on 0

pdbx(all) alias w on 1

pdbx(all) m

pdbx(0) list
   0:   15         MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get current process id */
   0:   16         MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */
   0:   17
   0:   18         if (myid == 0) {
   0:   19           for(i=0;i<50;++i) { sdata[i]=(double)i; }
   0:   20           MPI_Send(sdata,50,MPI_DOUBLE,1,123,MPI_COMM_WORLD);
   0:   21         } else {
   0:   22           MPI_Recv(rdata,55,MPI_DOUBLE,0,MPI_ANY_TAG,
   0:   23                                 MPI_COMM_WORLD, &status);
   0:   24           MPI_Get_count(&status,MPI_DOUBLE,&count);


pdbx(0) stop at 21
0:[0] stop at "send_recv_example.c":21

pdbx(0) w

pdbx(1) stop at 24
1:[0] stop at "send_recv_example.c":24

pdbx(1) on all

pdbx(all) cont
   1:[6] stopped in main at line 24 ($t1)
   1:   24           MPI_Get_count(&status,MPI_DOUBLE,&count);
```

```
   0:[6] stopped in main at line 21 ($t1)
   0:   21         } else {
```

We proceed to step through the program using the debugger's 'print' command to see the contents of variables. First, we focus on process 0 to verify that the array it sends to process 1 is both correct and that it has been initiated by allowing the execution to advance exclusively on process 0 until just before the call to MPI_FINALIZE:

```
pdbx(all) m

pdbx(0) where
   0:main(argc = 1, argv = 0x2ff2276c), line 21 in "send_recv_example.c"

pdbx(0) print i
   0:50

pdbx(0) print sdata
   0:(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0,
18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0, 28.0, 29.0, 30.0, 31.0, 32.0, 33.0, 34.0,
35.0, 36.0, 37.0, 38.0, 39.0, 40.0, 41.0, 42.0, 43.0, 44.0, 45.0, 46.0, 47.0, 48.0, 49.0,
9.7937096270247017e-78, -1.0229443777064821e+78, 1.9936034456461534e-314, 0.0, 0.0)

pdbx(0) n
   0:stopped in main at line 21 ($t1)
   0:   21         } else {

pdbx(0) n
   0:stopped in main at line 27 ($t1)
   0:   27         MPI_Finalize();                       /* let MPI finish up ... */
```

Now that the sender (process 0) has essentially completed its task, we are ready to switch the execution to the receiver (process 1). The input arguments such as size of receive array for the MPI function MPI_RECV is printed to ensure its correctness. Then, the execution on process 1 is allowed to proceed. Since there are no bugs in this program we can just step through the MPI_RECV call and pause there to verify that the array *rdata* has indeed received the array *sdata* sent by MPI_SEND.

```
pdbx(0) w

pdbx(1) where
   1:main(argc = 1, argv = 0x2ff22764), line 24 in "send_recv_example.c"

pdbx(1) n
   1:stopped in main at line 25 ($t1)
   1:   25           do_work(rdata, count);

pdbx(1) print count
   1:50

pdbx(1) print rdata[0]
   1:0.0

pdbx(1) print rdata[49]
   1:49.0

pdbx(1) n
   1:This function performs computations with rdata
   1:stopped in main at line 27 ($t1)
   1:   27         MPI_Finalize();                       /* let MPI finish up ... */

pdbx(1) on all

pdbx(all) cont

pdbx(all)    0:
   0:execution completed
   1:
   1:execution completed
0029-2131 All tasks have exited.
Issue quit then restart the debugger if you wish to continue debugging.
```

```
pdbx(all) quit
```

3.2.4 Exercises

1. Use your favorite debugger to test the send_recv_example.c. Print both *sdata* and *rdata* where appropriate. Do you know, for instance, how to print a partial array within pdbx? Insert a print statement before the MPI_INIT statement to verify that pdbx has an automatic breakpoint that is positioned at the first executable statement of the function main.
2. If you try to print user-defined constants (thru C preprocessor #define)or MPI predefined constants in a debugger environment it will fail. Do you know why it fails?

3.2.5 Solutions

1. Use your favorite debugger to test the send_recv_example.c. pdbx(0) print sdata
   pdbx(1) print rdataTo print part of, say, *sdata*, enter`pdbx(0) print sdata(3..6)` prints the 4th through the 7th element of array *sdata*.
2. If you try to print user-defined constants (thru C preprocessor #define) or MPI predefined constants in a debugger environment it will fail. Do you know why it fails? Constant variables defined through #define are replaced during program compilation with their respective defined values. Hence, at runtime the variables no longer exist in the object file and therefore are considered by the debugger as "undefined" if you try to reference them (e.g. in a 'print' command). In a debugger like IBM's PDBX, attempting to print MPI predefined constants such as MPI_ANY_TAG in a C program will fail. However, it works fine in a Fortran program because they are not defined using #define).

3.3 Finding Logical Errors

3.3.1 Finding Logical Errors

In this lesson, you will learn about one of the most frequently encountered types of bugs - logical errors. A sample program is given to illustrate the types of problems caused by these errors and an exercise given to let you practice what you learned.

Logical errors are programming errors that involve branching such as in a logical IF-block. These branching errors occur quite frequently in serial programming. They happen even more frequently in parallel programming in general, and particularly in the *Single Program Multiple Data (SPMD)* programming paradigm. SPMD is commonly used in distributed memory message-passing MPI or PVM parallel programming.

There are literally tens, if not hundreds, of circumstances wherein a logical error occurs. Because there are so many ways that logical errors can happen, we can not cover them all, let alone demonstrate them. Therefore, we give a simple example to illustrate the general nature of these errors.

3.3.2 Sample Code with Logical Errors

Our sample program, logical_errors_example.c, (shown below) is written for two processes: by design, process 0 defines an array *sdata*, then sends it to process 1. However, due to an error this array is defined on process 1 instead. Subsequently, process 1 receives the message from process 0, determines its size and then passes the information on to the function do_work. The executable runs to completion. In actual code, the job may not be able to run to completion; or it would most likely generate the wrong answer.

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
```

```
void main(int argc, char* argv[])
{
      double sdata[55];
      double rdata[55];
      int p, i, count, myid, n;
      MPI_Status status;

      MPI_Init(&argc, &argv);                /* starts MPI */
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get current process id */
      MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */

      if (myid == 1) {
        for(i=0;i<50;++i) { sdata[i]=(double)i; }
      }

      if (myid == 0) {
        MPI_Send(sdata,50,MPI_DOUBLE,1,123,MPI_COMM_WORLD);
      } else {
        MPI_Recv(rdata,55,MPI_DOUBLE,0,MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        MPI_Get_count(&status, MPI_DOUBLE, &count);
        do_work(rdata, count);
      }
      MPI_Finalize();                       /* let MPI finish up ... */
}
do_work(double *rdata, int count)
{
printf("This function performs computation with rdata\n"););
}
```

3.3.3 Debugging the Sample Code

We compile our sample program with the '-g' option and then proceed to step through the code in a debugger. This debugging session was done using the pdbx debugger on an IBM pSeries 690 computer system running AIX.

```
twister:debugger/LOGICAL-ERRORS % pdbx logical_errors_example -procs 2
pdbx Version 4, Release 1 -- Jul  7 2004 17:26:19

   0:Core file "
   0:" is not a valid core file (ignored)
   1:Core file "
   1:" is not a valid core file (ignored)
   0:reading symbolic information ...
   1:reading symbolic information ...
   1:[1] stopped in main at line 14 ($t1)
   1:   14         MPI_Init(&argc, &argv);               /* starts MPI */
   0:[1] stopped in main at line 14 ($t1)
   0:   14         MPI_Init(&argc, &argv);               /* starts MPI */
0031-504  Partition loaded ...

pdbx(all)
```

For convenience, we first label *m* to mean the "master" (or process 0)and *w* the "worker" (or process 1). We then list (partially) the program source on both processes. The program is allowed to run and stopped at the processes' respective breakpoints (line 21 for both processes).

```
pdbx(all) alias m on 0

pdbx(all) alias w on 1

pdbx(all) list
   0:   15         MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get current process id */
   0:   16         MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */
   0:   17
   0:   18         if (myid == 1) {
   0:   19           for(i=0;i<50;++i) { sdata[i]=(double)i; }
   0:   20         }
   0:   21         if (myid == 0) {
   0:   22           MPI_Send(sdata,50,MPI_DOUBLE,1,123,MPI_COMM_WORLD);
   0:   23         } else {
```

```
   0:   24           MPI_Recv(rdata,55,MPI_DOUBLE,0,MPI_ANY_TAG,
   1:   15         MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get current process id */
   1:   16         MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */
   1:   17
   1:   18         if (myid == 1) {
   1:   19           for(i=0;i<50;++i) { sdata[i]=(double)i; }
   1:   20         }
   1:   21         if (myid == 0) {
   1:   22           MPI_Send(sdata,50,MPI_DOUBLE,1,123,MPI_COMM_WORLD);
   1:   23         } else {
   1:   24           MPI_Recv(rdata,55,MPI_DOUBLE,0,MPI_ANY_TAG,

pdbx(all) stop at 21
all:[0] stop at "logical_error_example.c":21

pdbx(all) cont

   0:[6] stopped in main at line 21 ($t1)
   0:   21         if (myid == 0) {
   1:[6] stopped in main at line 21 ($t1)
   1:   21         if (myid == 0) {
```

Next, we focus on process 0. We verify that the array it sends to process 1 is both correct and that it has been initiated by allowing the execution to advance exclusively on process 0 until just before MPI_FINALIZE.

```
pdbx(all) m

pdbx(0) stop at 29
0:[0] stop at "logical_error_example.c":29

pdbx(0) cont
   0:[7] stopped in main at line 29 ($t1)
   0:   29         MPI_Finalize();                       /* let MPI finish up ... */

pdbx(0) print sdata[5]
   0:2.1219958023287822e-314

pdbx(0) print sdata[7]
   0:9.7981735890506941e-78
```

It is obvious that *sdata* is not defined on process 0 and hence we expect that process 1 would receive bad, as well as incorrect data. We continue with the debugger to see what we receive on process 1.

```
pdbx(0) w

pdbx(1) where
   1:main(argc = 1, argv = 0x2ff227dc), line 21 in "logical_error_example.c"

pdbx(1) stop at 29
1:[0] stop at "logical_error_example.c":29

pdbx(0) print rdata[5]
   0:2.1219958023287822e-314

pdbx(0) print rdata[7]
   0:9.7981735890506941e-78
```

The above *rdata* values confirmed that process 1 received the (undefined) data sent by process 0 correctly - as expected.

```
pdbx(all) quit
```

3.3.4 Exercises

1. Use your favorite debugger to test the following logical errors program, logical_errors_example.c.

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>

void main(int argc, char* argv[])
```

```
{
      double sdata[55];
      double rdata[55];
      int p, i, count, myid, n;
      MPI_Status status;

/* Starts MPI processes ... */

      MPI_Init(&argc, &argv);                /* starts MPI */
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get current process id */
      MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */

      if (myid == 1) {
        for(i=0;i<50;++i) { sdata[i]=(double)i; }
      }
      if (myid == 0) {
        MPI_Send(sdata,50,MPI_DOUBLE,1,123,MPI_COMM_WORLD);
      } else {
        MPI_Recv(rdata,55,MPI_DOUBLE,0,MPI_ANY_TAG,
                            MPI_COMM_WORLD, &status);
        MPI_Get_count(&status,MPI_DOUBLE,&count);
        do_work(rdata, count);
      }
      MPI_Finalize();                       /* let MPI finish up ... */
}
do_work(double *rdata, int count)
{
  printf("This function performs computations with rdata\n");
}
```

3.3.5 Solutions

1. Use your favorite debugger to test the logical_errors_example.c. There is no actual "solution" for this exercise. If you need help please see the section "Logical Errors Example".

3.4 Debugging Process Count Errors

3.4.1 Debugging Process Count Errors

Introduction

In MPI programming, a common reason for a program to hang is a faulty assumption about the number or configuration of the MPI processes spawned by the parallel program. This is especially common when using MPI's *blocking* point-to-point communication routines (eg. MPI_SEND() and MPI_RECV()).

Objectives

In this lesson, you will learn how to diagnose programming errors involving incorrect process spawning assumptions and debug them using the Etnus Totalview® debugger.

3.4.2 Sample Program with Process Count Errors

Consider this binary tree example written in Fortran 77, prefix-mpi-buggy.f, using MPI:

```
 program prefix

 include 'mpif.h'

 integer TCONT,TSTOP
 parameter(TCONT=999)
 parameter(TSTOP=13)

 integer child0,child1,parent
 logical is_leaf
 logical is_down
 integer op
 integer x,y,here
 integer values(0:2)
 integer hstatus(MPI_STATUS_SIZE)
```

```
 integer xstatus(MPI_STATUS_SIZE),ystatus(MPI_STATUS_SIZE)
 integer reqid

 data values/3,4,5/

 call MPI_Init(ierr)
 call MPI_Comm_size(MPI_COMM_WORLD,npes,istat)
 call MPI_Comm_rank(MPI_COMM_WORLD,mype,istat)


 if(is_leaf(mype)) here=values(mod(mype,3))

c Enter leaf region of code

 if(is_leaf(mype)) then

   print *, 'This is leaf ',mype,' my starting value is ',here

c Leaf first sends input values to its parent

   call MPI_Issend(here,1,MPI_INTEGER,parent(mype),TCONT,
+       MPI_COMM_WORLD,reqid,istat)

c Leaf then receives messages from parent and then
c performs the operation on input and its stored value.
c Last message tagged with STOP id

   if (mype.ne.npes/2) then
     msgtype=TCONT
     do while (msgtype.ne.TSTOP)
        call MPI_Recv(x,1,MPI_INTEGER,parent(mype),MPI_ANY_TAG,
+            MPI_COMM_WORLD,xstatus,istat)
        here=op(x,here)
        msgtype=xstatus(MPI_TAG)
     end do
   end if

c Leaf is completely finished, prints out its output value

   print *, 'This is leaf ',mype,' my final value is ',here


c This is the code for what a normal node does

 else

c First thing: get input values from its children

    call MPI_Recv(x,1,MPI_INTEGER,child0(mype),TCONT,
+        MPI_COMM_WORLD,xstatus,istat)

    call MPI_Recv(y,1,MPI_INTEGER,child1(mype),TCONT,
+        MPI_COMM_WORLD,ystatus,istat)


c Perform the operation on the inputs

   here=op(x,y)


c Send result up to parent (if not the root node)

   if (mype.ne.1) then
      call MPI_Issend(here,1,MPI_INTEGER,parent(mype),TCONT,
+          MPI_COMM_WORLD,reqid,istat)
   end if

c Send input value recieved from left child to right child.
c Nodes which will receive nothing from above send the STOP code
```

```
   if(.not. is_down(mype)) then
     msgtype=TSTOP
   else
     msgtype=TCONT
   endif
   call MPI_Issend(x,1,MPI_INTEGER,child1(mype),msgtype,
+       MPI_COMM_WORLD,reqid,istat)


c if node receiving from above, takes in messages from parent and passes
c value down to both children.
c ceases when STOP message id arrives

   if (is_down(mype)) then
     msgtype=TCONT
     do while (msgtype.ne.TSTOP)
        call MPI_Recv(here,1,MPI_INTEGER,parent(mype),MPI_ANY_TAG,
+            MPI_COMM_WORLD,hstatus,istat)
        msgtype=hstatus(MPI_TAG)
        if (msgtype.ne.TCONT.and.msgtype.ne.TSTOP) then
           write(*,*) 'PE ',mype,' -- bad msgtype ',msgtype,
+            ' received from rank ',parent(mype)
           call MPI_Abort(MPI_COMM_WORLD,-msgtype)
        endif

        call MPI_Issend(here,1,MPI_INTEGER,child0(mype),msgtype,
+            MPI_COMM_WORLD,reqid,istat)

        call MPI_Issend(here,1,MPI_INTEGER,child1(mype),msgtype,
+            MPI_COMM_WORLD,reqid,istat)
     end do
   end if

 end if

 call MPI_Barrier(MPI_COMM_WORLD,istat)
 call MPI_Finalize(istat)
 stop
 end


c Function to return the zeroth child of a node

 integer function child0(me)
   child0=2*me
   return
 end


c Function to return the first child of a node

 integer function child1(me)
   child1=2*me+1
   return
 end

c Function to return the parent of a node

 integer function parent(me)
   parent=me/2
   return
 end

c Function to test if a node is a leaf

 logical function is_leaf(me)
   include 'mpif.h'
   call MPI_Comm_size(MPI_COMM_WORLD,npes,istat)
   if (me.ge.(npes/2).and.me.le.(npes-1)) then
```

```
      is_leaf=.true.
    else
      is_leaf=.false.
    end if
    return
  end

c Function to test if a node will pass messages down

  logical function is_down(me)
    include 'mpif.h'
    integer me,idwn
    call MPI_Comm_size(MPI_COMM_WORLD,npes,istat)
    is_down=.true.
    idwn=1
    do while (idwn.lt.npes/2)
      if (me.eq.idwn) is_down=.false.
      idwn=2*idwn
    end do
    return
  end

c Function that performs the associative operation

  integer function op(i,j)
    if(j.eq.3) op=3
    if(j.eq.4) op=i
    if(j.eq.5) op=5
    return
  end
```

This program constructs a binary tree topology among the MPI processes, which is used to pass around a series of tokens used to execute an arbitrary associative operation op(). The functions parent(), child0(), and child1() are used to determine the ranks of a process's neighbors in the tree structure. The program uses MPI_ISSEND() and MPI_RECV() to transport the operands for the associative operation, using different message tag values to indicate if a process should continue processing data or stop execution after the current operand.

The program also has two LOGICAL functions, is_leaf() and is_down(), used to determine a given MPI process's relative location in the tree and whether or not it has children with which to communicate.

As you would suspect, this program has a bug in it — when run, it hangs. For example, it displays the following behavior when running on 8 processors:

```
This is leaf            5  my starting value is            5
This is leaf            4  my starting value is            4
This is leaf            6  my starting value is            3
This is leaf            7  my starting value is            4
This is leaf            4  my final value is            4
This is leaf            5  my final value is            5
This is leaf            7  my final value is            3
This is leaf            6  my final value is            3
```

Note the lack of response from MPI processes 0 through 3. Worse, this behavior is independent of the number of MPI processes used.

3.4.3 Debugging the Sample the Program

To find the bug in our sample program, we run Totalview® on it and then trace the program's flow to find the bug's location. Once we find it, we will fix the code to eliminate the bug.

Running Totalview® on the Program

Before running Totalview® on our program, we want to make sure that it is compiled with debugging symbols enabled. On many systems, this can be be done by issuing the following command:

```
mpif77 -g prefix-mpi-buggy.f -o prefix
```

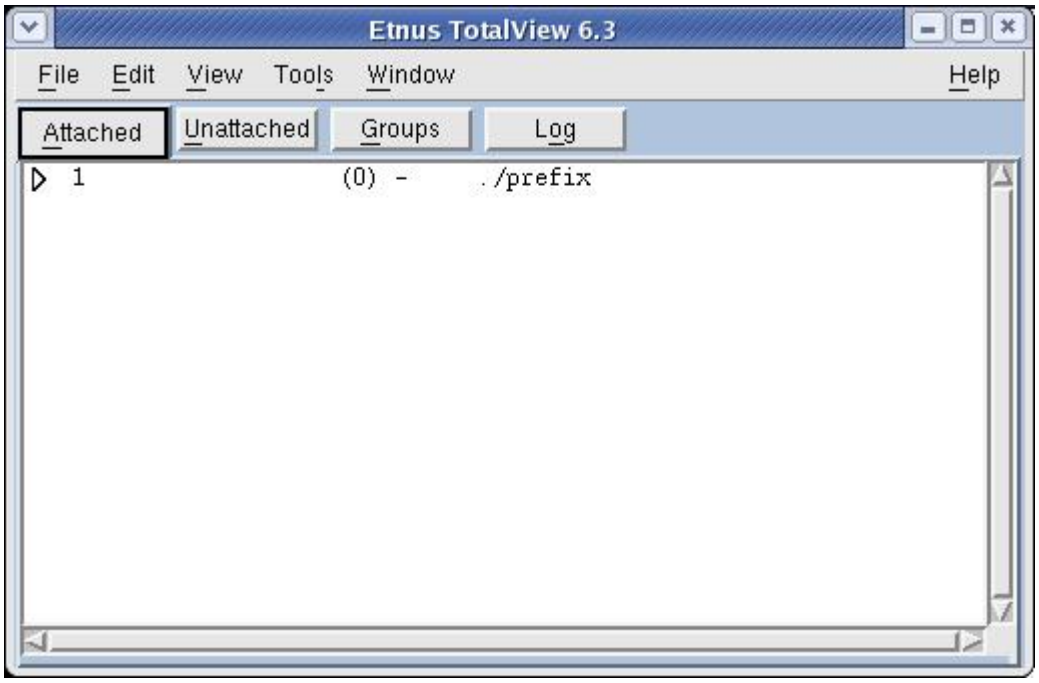
Now we try debugging the program in Totalview® with eight MPI processes. The exact command you need to use varies

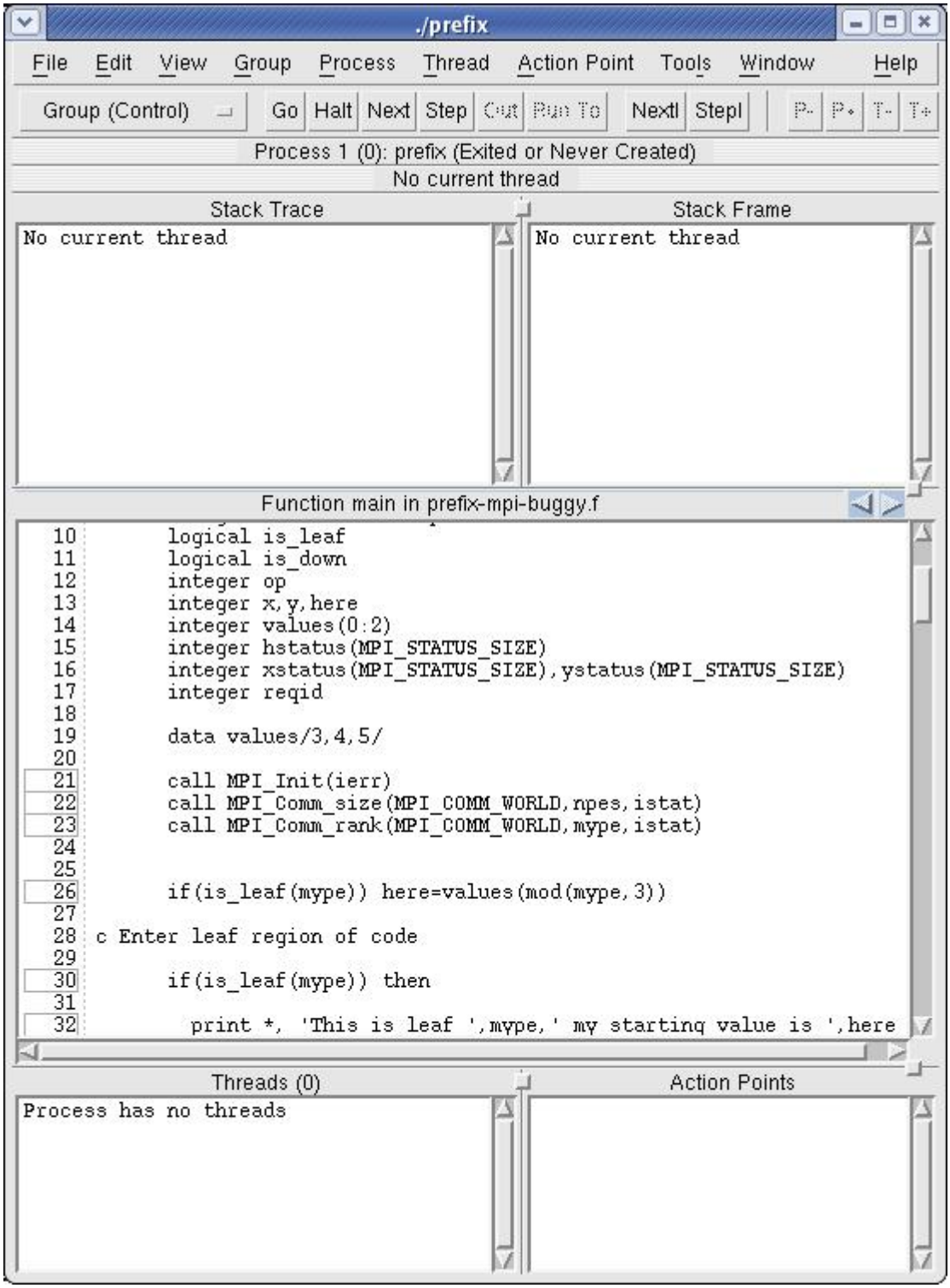
from system to system, but commonly used ones are:

```
mpirun -tv -np 8 ./prefix
mpiexec -tv -n 8 ./prefix
```

Note that the '-tv' option in these commands stands for "Totalview".

Assuming you have a working X11 display, two windows should be displayed; a control window and a main window. The Totalview® control window and the Totalview® main window in their initial states are shown in the following images, respectively:

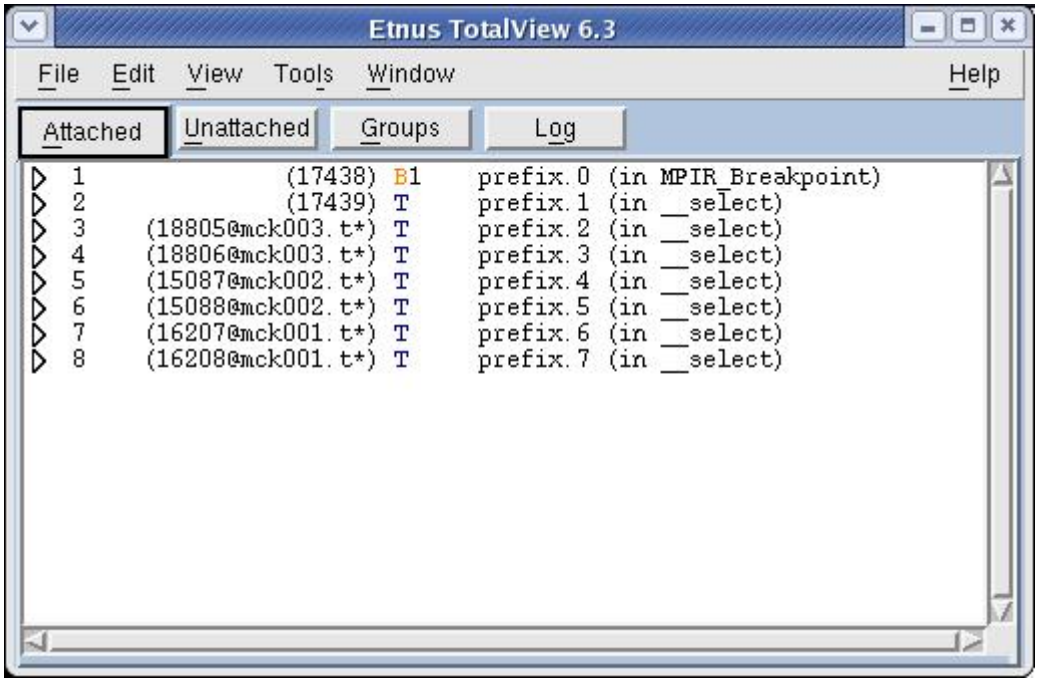




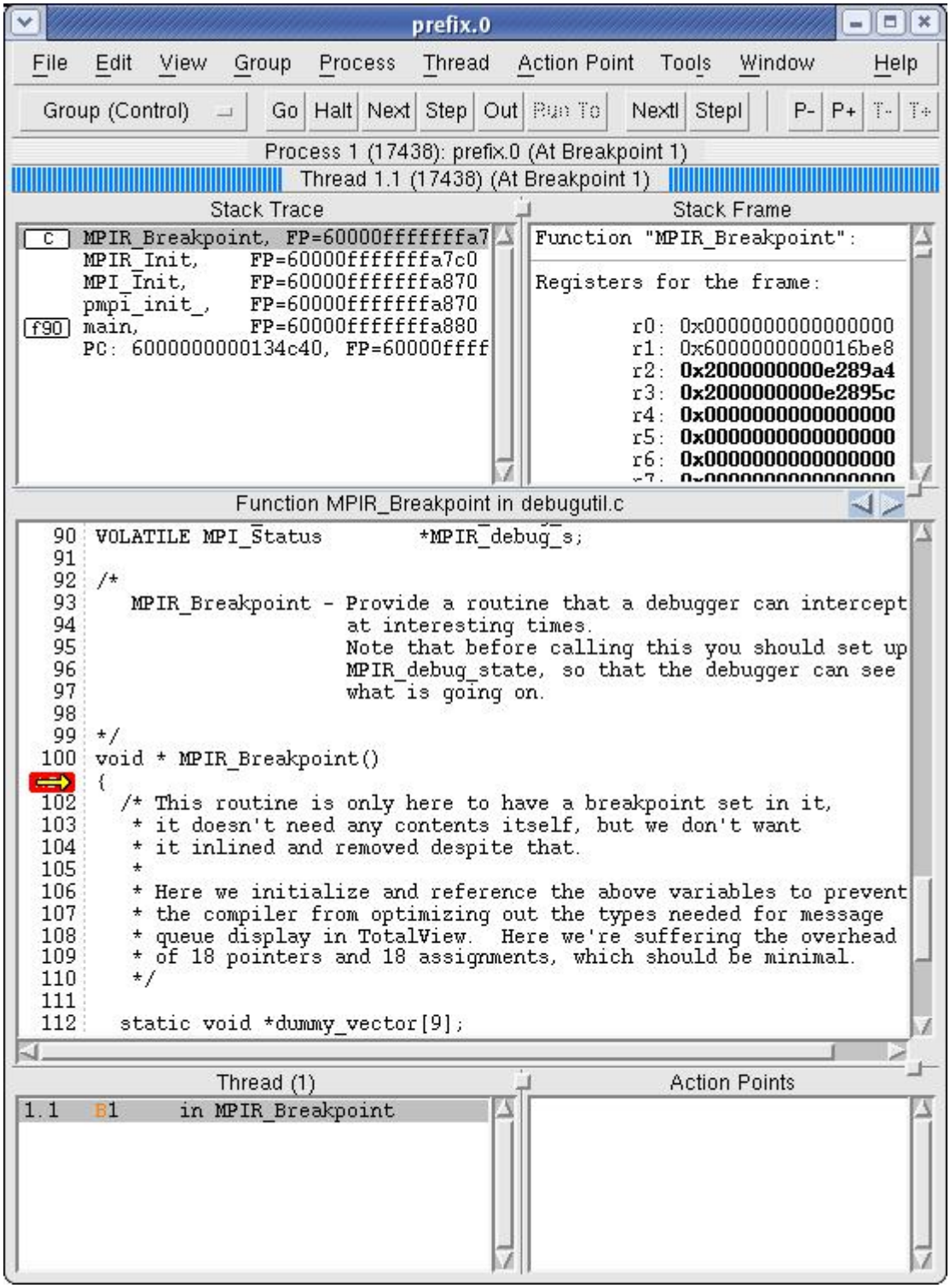
Notice that the control window lists only one process. Before you can do any debugging, you need to start the parallel program by clicking on the 'Go' button in the main window.

Tracing the Program's Flow in Totalview®

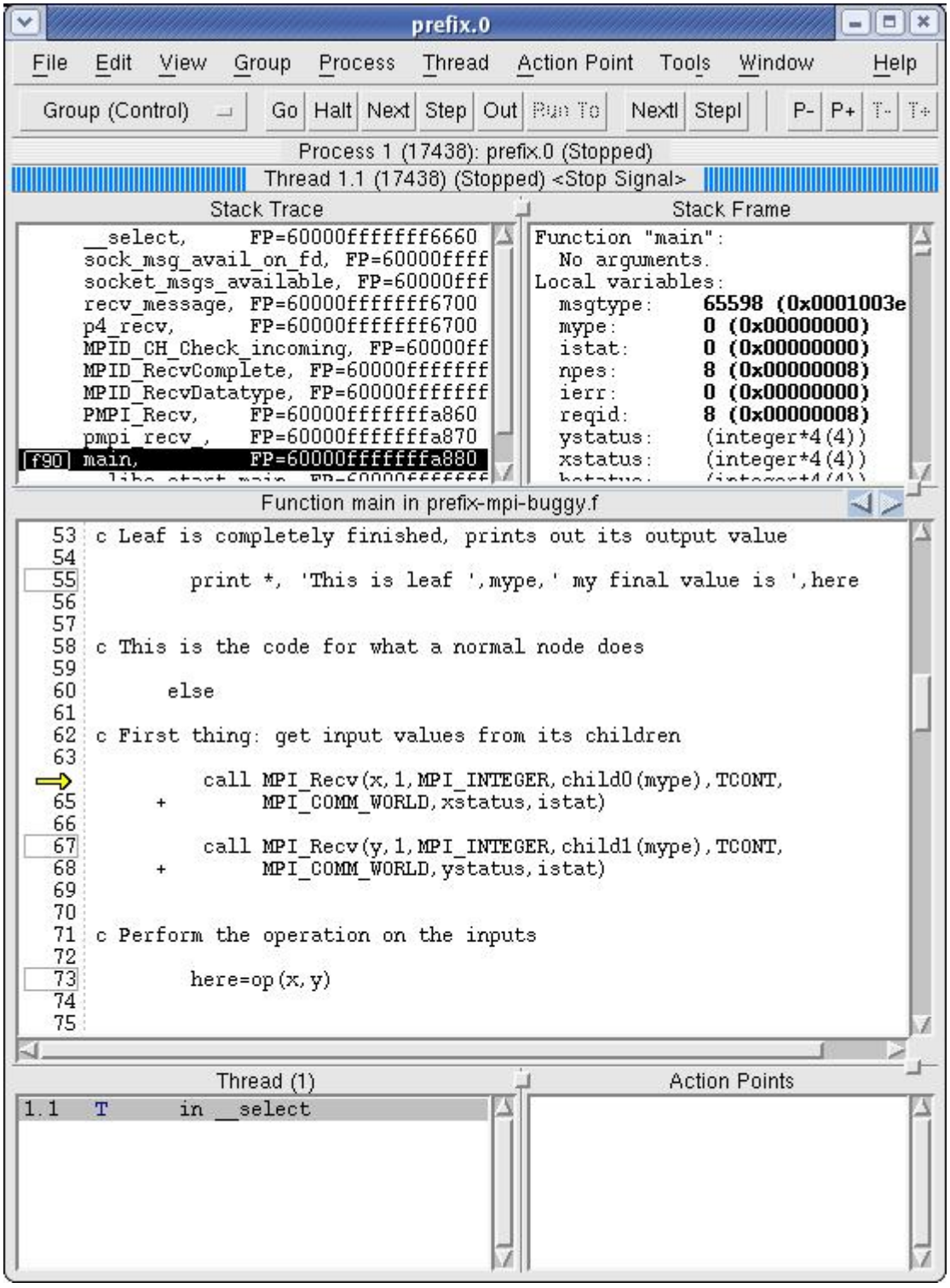
After clicking the 'Go' button in the main window, you are prompted to confirm that you want to start the parallel program. You may also get warnings if your MPI implementation was not compiled with message queue debugging information.Once you have responded to the prompts, your parallel program is started and waiting inside MPI_INIT(), as seen in the stack trace pane in the following figure:



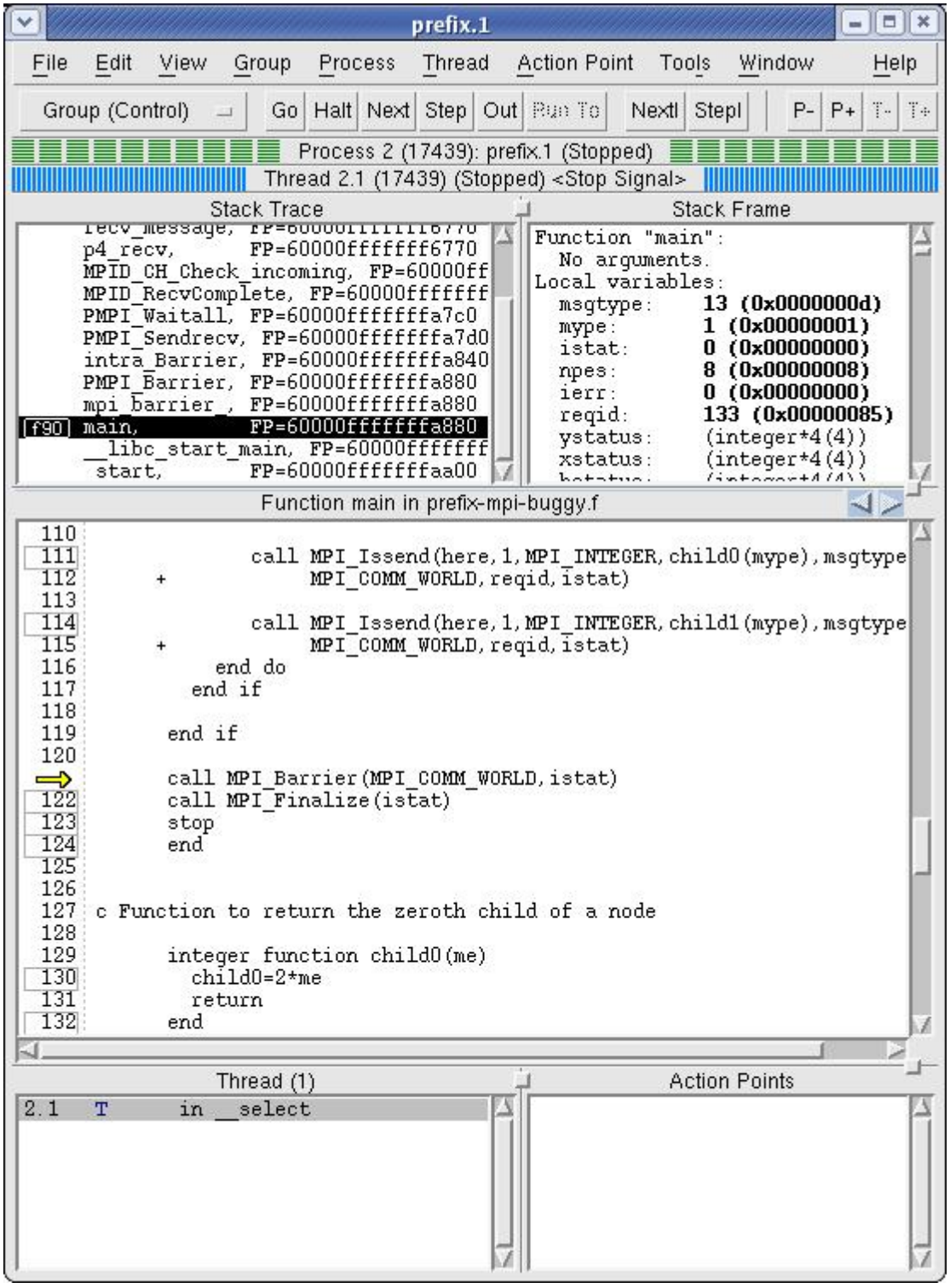
The Totalview® main window after program startup should look like this:



At this point, click the 'Go' button again and the parallel program will start running in earnest. After the program is allowed to run for a few minutes, it reaches its final hung state. To stop the program so you are able to examine its state, click on the 'Halt' button. Once you have done that, you can use the 'P+' and 'P-' buttons to move between processes in the parallel program. Click on the 'main' line in the stack trace pane to see where the program is currently located in that process. You should find that process 1 (corresponding to MPI rank 0) is stopped inside MPI_RECV(), as shown in this stack trace window:



Meanwhile, all seven other processes are stopped in an MPI_BARRIER() at the end of the program:



Finding and Fixing the Bug

We found that MPI rank 0 was not behaving in the same way as the other MPI processes. Whereas the other processes completed the program successfully, rank 0 was hung in an MPI_RECV() fairly early on in the program. Something in the construction of either our communication scheme or our binary tree configuration is flawed.

If we take another look at values returned by the functions parent(), child0(), and child1(), as shown in the following table, we find the answer:

mype	parent(mype)	child0(mype)	child1(mype)
i	i/2	2*i	2*i+1
0	0	0	1
1	0	2	3

2	1	4	5
3	1	6	7

Rank 0 thinks it is both its own parent **and** its own child, which causes it to try to receive a message from itself that it never has a chance to send! If we reexamine the tree functions (including is_leaf() and is_down()), they are all based on the idea of the process ranks starting at 1, but in MPI the ranks always start at 0. This problem can be easily fixed, by replacing the code:

```
c This is the code for what a normal node does
      else
```

with:

```
c This is the code for what a normal node does
      else if (mype.ne.0) then
```

The corrected source code, prefix-mpi-fixed.f, is as follows:

```
 program prefix

 include 'mpif.h'

 integer TCONT,TSTOP
 parameter(TCONT=999)
 parameter(TSTOP=13)

 integer child0,child1,parent
 logical is_leaf
 logical is_down
 integer op
 integer x,y,here
 integer values(0:2)
 integer hstatus(MPI_STATUS_SIZE)
 integer xstatus(MPI_STATUS_SIZE),ystatus(MPI_STATUS_SIZE)
 integer reqid

 data values/3,4,5/

 call MPI_Init(ierr)
 call MPI_Comm_size(MPI_COMM_WORLD,npes,istat)
 call MPI_Comm_rank(MPI_COMM_WORLD,mype,istat)


 if(is_leaf(mype)) here=values(mod(mype,3))

c Enter leaf region of code

 if(is_leaf(mype)) then

   print *, 'This is leaf ',mype,' my starting value is ',here

c Leaf first sends input values to its parent

   call MPI_Issend(here,1,MPI_INTEGER,parent(mype),TCONT,
+       MPI_COMM_WORLD,reqid,istat)

c Leaf then receives messages from parent and then
c performs the operation on input and its stored value.
c Last message tagged with STOP id

   if (mype.ne.npes/2) then
     msgtype=TCONT
     do while (msgtype.ne.TSTOP)
        call MPI_Recv(x,1,MPI_INTEGER,parent(mype),MPI_ANY_TAG,
+            MPI_COMM_WORLD,xstatus,istat)
        here=op(x,here)
```

```
        msgtype=xstatus(MPI_TAG)
      end do
   end if

c Leaf is completely finished, prints out its output value

   print *, 'This is leaf ',mype,' my final value is ',here


c This is the code for what a normal node does

 else if (mype.ne.0) then

c First thing: get input values from its children

    call MPI_Recv(x,1,MPI_INTEGER,child0(mype),TCONT,
+        MPI_COMM_WORLD,xstatus,istat)

    call MPI_Recv(y,1,MPI_INTEGER,child1(mype),TCONT,
+        MPI_COMM_WORLD,ystatus,istat)


c Perform the operation on the inputs

   here=op(x,y)


c Send result up to parent (if not the root node)

   if (mype.ne.1) then
      call MPI_Issend(here,1,MPI_INTEGER,parent(mype),TCONT,
+          MPI_COMM_WORLD,reqid,istat)
   end if

c Send input value received from left child to right child.
c Nodes which will receive nothing from above send the STOP code

   if(.not. is_down(mype)) then
     msgtype=TSTOP
   else
     msgtype=TCONT
   endif
   call MPI_Issend(x,1,MPI_INTEGER,child1(mype),msgtype,
+       MPI_COMM_WORLD,reqid,istat)


c if node receiving from above, takes in messages from parent and passes
c value down to both children.
c ceases when STOP message id arrives

   if (is_down(mype)) then
     msgtype=TCONT
     do while (msgtype.ne.TSTOP)
        call MPI_Recv(here,1,MPI_INTEGER,parent(mype),MPI_ANY_TAG,
+            MPI_COMM_WORLD,hstatus,istat)
        msgtype=hstatus(MPI_TAG)
        if (msgtype.ne.TCONT.and.msgtype.ne.TSTOP) then
           write(*,*) 'PE ',mype,' -- bad msgtype ',msgtype,
+            ' received from rank ',parent(mype)
           call MPI_Abort(MPI_COMM_WORLD,-msgtype)
        endif

        call MPI_Issend(here,1,MPI_INTEGER,child0(mype),msgtype,
+            MPI_COMM_WORLD,reqid,istat)

        call MPI_Issend(here,1,MPI_INTEGER,child1(mype),msgtype,
+            MPI_COMM_WORLD,reqid,istat)
     end do
   end if
```

```
 end if

 call MPI_Finalize(istat)
 stop
 end


c Function to return the zeroth child of a node

 integer function child0(me)
   child0=2*me
   return
 end


c Function to return the first child of a node

 integer function child1(me)
   child1=2*me+1
   return
 end

c Function to return the parent of a node

 integer function parent(me)
   parent=me/2
   return
 end

c Function to test if a node is a leaf

 logical function is_leaf(me)
   include 'mpif.h'
!   call pvmfgsize(char(0),npes)
   call MPI_Comm_size(MPI_COMM_WORLD,npes,istat)
   if (me.ge.(npes/2).and.me.le.(npes-1)) then
     is_leaf=.true.
   else
     is_leaf=.false.
   end if
   return
 end

c Function to test if a node will pass messages down

 logical function is_down(me)
   include 'mpif.h'
   integer me,idwn
!   call pvmfgsize(char(0),npes)
   call MPI_Comm_size(MPI_COMM_WORLD,npes,istat)
   is_down=.true.
   idwn=1
   do while (idwn.lt.npes/2)
     if (me.eq.idwn) is_down=.false.
     idwn=2*idwn
   end do
   return
 end

c Function that performs the associative operation

 integer function op(i,j)
   if(j.eq.3) op=3
   if(j.eq.4) op=i
   if(j.eq.5) op=5
   return
 end
```

3.4.4 Debugging Exercise

The following C program, ring.c, constructs a ring topology out of its MPI processes and has each process send data to its neighbor on one side and receive data from its neighbor on the other side. However, as currently written, this program only works if it has four MPI processes. Find a way to make it work with an arbitrary number of processes. (Hint: there is more than one way to do this.)

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  int down,up,rank,data;
  MPI_Status rstat,sstat;
  MPI_Request req;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  /* Create ring topology */
  if ( rank==3 )
    up=0;
  else
    up=rank+1;
  if ( rank==0 )
    down=3;
  else
    down=rank-1;

  /* Send up, receive from down */
  /* Payload is rank */
  MPI_Issend(&rank,1,MPI_INT,up,0,MPI_COMM_WORLD,&req);
  MPI_Recv(&data,1,MPI_INT,down,0,MPI_COMM_WORLD,&rstat);
  MPI_Wait(&req,&sstat);
  MPI_Barrier(MPI_COMM_WORLD);

  if ( data!=down )
    {
      fprintf(stderr,"Rank %d got incorrect data %d from rank %d\n",
              rank,data,down);
    }

  return MPI_Finalize();
}
```

3.4.5 Solution to Debugging Exercise

There are at least two ways to make ring.c work with an arbitrary number of processes. The simplest, shown in ring-fixed.c below,is to use the number of MPI processes returned by MPI_COMM_SIZE() to compute the rank of the "end' process rather than use the static value 3.

ring-fixed.c

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  int down,up,rank,size,data;
  MPI_Status rstat,sstat;
  MPI_Request req;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  /* Create ring topology */
  if ( rank==(size-1) )
```

```
    up=0;
  else
    up=rank+1;
  if ( rank==0 )
    down=(size-1);
  else
    down=rank-1;

  /* Send up, receive from down */
  /* Payload is rank */
  MPI_Issend(&rank,1,MPI_INT,up,0,MPI_COMM_WORLD,&req);
  MPI_Recv(&data,1,MPI_INT,down,0,MPI_COMM_WORLD,&rstat);
  MPI_Wait(&req,&sstat);
  MPI_Barrier(MPI_COMM_WORLD);

  if ( data!=down )
    {
      fprintf(stderr,"Rank %d got incorrect data %d from rank %d\n",
              rank,data,down);
    }

  return MPI_Finalize();
}
```

Another, more sophisticated way of handling this is to use MPI's built-in Cartesian topology operations to construct a ring. This approach is shown in ring-top.c below.

ring-top.c

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  int down,up,rank,size,data;
  int yes=1;
  MPI_Status rstat,sstat;
  MPI_Request req;
  MPI_Comm ring;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  /* Create ring topology */
  MPI_Cart_create(MPI_COMM_WORLD,1,&size,&yes,yes,&ring);
  MPI_Cart_shift(ring,0,1,&down,&up);

  /* Send up, receive from down */
  /* Payload is rank */
  MPI_Issend(&rank,1,MPI_INT,up,0,ring,&req);
  MPI_Recv(&data,1,MPI_INT,down,0,ring,&rstat);
  MPI_Wait(&req,&sstat);
  MPI_Barrier(MPI_COMM_WORLD);

  if ( data!=down )
    {
      fprintf(stderr,"Rank %d got incorrect data %d from rank %d\n",
              rank,data,down);
    }

  return MPI_Finalize();
}
```

The routine MPI_CART_CREATE() is used to construct the ring topology as a one-dimensional grid of processes with periodicity set to "true", creating a new communicator called *ring*. Then, MPI_CART_SHIFT() is used to determine the neighbors' ranks in the *ring* communicator. Since the neighbors' ranks are known only in the new communicator, it must also be used in calls to MPI_ISSEND() and MPI_RECV().

3.5 Updating Shared Memory Locations

3.5.1 Updating Shared Memory Locations

Introduction

The shared memory programming model is popular for developing parallel applications. This popularity is due to its simple programming interface and the capability it provides for parallelizing all or part of an application. The ability to parallelize sections of code is especially advantageous when you are trying to increase performance of a legacy code that was written using serial programming techniques and algorithms.

In shared memory programming, data is communicated between threads by storing data in a shared memory region that both threads use to access and update variables. All that you need to control is the order and access mechanisms that each thread uses to read and write information. However, if you do not control these correctly you will encounter programming errors such as updating shared memory regions out of order.

Objectives

In this lesson you will learn about updating shared memory locations, the basic nature of problems encountered, and how to debug an application with this type of error.

3.5.2 Common uses of Shared Memory Variables

We now give a brief overview of two common uses of shared memory variables:

1. distributing loops among multiple threads
2. indexing

Distributing Loops Among Multiple Threads

Loops are primarily used to perform the following tasks:

- Update a matrix or array
- Read data from a matrix or array and reduce to a smaller subset

In shared memory programming, loop iterations are commonly distributed among parallel threads. This is most easily done with OpenMP and the !$OMP DO directive.

When updating a matrix or array, it is often possible to structure the algorithm to update the matrix in an independent fashion. This allows for easy distribution of work to the various threads.

When an operation is desired on the data, perhaps summing the entries of an array or calculating a matrix coefficent, the data is first reduced independently on each thread and then combined at the end of the loop. You can do this manually by adding the appropriate code to your program or automatically by using the reduction clause in OpenMP.

Indexing

A running index is a common programming technique used in older serial programs. This technique uses an incremental counter that is incremented based on some logic statement. Since the increment is not based exclusively on the loop index and does not increment with each loop iteration, it is difficult to parallelize.

One example of a running index is found in linear algebra applications that utilize sparse matrices. With sparse matrices, it is desirable to store only the non-zero elements of an array to reduce the overall amount of storage and number of computations. You can do this by using a running index to store only the non-zero entries in three, one-dimensional arrays. The first two arrays store the i,j coordinate of the non-zero element and the third array stores the corresponding value. There are many algorithms available to do this. In this lesson, we give a simplified example that summarizes the basic concept.

For example, take the 3x3 dense matrix initialized as shown below:

1	4	7
2	5	8
3	6	9

This could be represented in the following alternate format given by three separate one-dimensional arrays:

Row	Column	Value
1	1	1
2	1	2
3	1	3
1	2	4
2	2	5
3	2	6
1	3	7
2	3	8
3	3	9

For dense matrices this method does not offer much benefit. For large sparce matrices, however, using this method can result in significant memory savings. Given the following sparse matrix representation:

1		7
2	5	
		9

the corresponding reduced format is:

Row	Column	Value
1	1	1
2	1	2
2	2	5
1	3	7
3	3	9

This example illustrates the memory savings that can be achieved.

For linear algebra problems, the indexing process need only be performed once prior to solution. For a large percentage of problems indexing is not the primary bottleneck, rather it is the solution. There are, however, many situations where the problem gets automatically re-meshed or some other aspect of the problem is modified and the system of equations need to be re-generated. This causes more frequent use of the indexing algorithm and hence, increases the benefit from parallelization.

3.5.3 Serial Implementation of an Indexing Algorithm

Shown below is a serial Fortran 90 program, serial_index, that uses a running index to build three arrays from a single two-dimensional matrix. For completeness, our sample code only includes an array initialization and operates on a 3x3 matrix. In real applications, the matrix would be significantly larger and assembled from a physical problem rather than a simple initialization. The purpose of spending time establishing this sample algorithm is to show how the non-zero values are assembled using the running index.

```
Program serial_index

integer,parameter:: NUM_ROWS=3
```

```
integer,parameter:: NUM_COLS=3

REAL,dimension(NUM_ROWS,NUM_COLS) :: data_2d
REAL, dimension(NUM_ROWS*NUM_COLS) :: data_1d
REAL, dimension(NUM_ROWS*NUM_COLS) :: JR
REAL, dimension(NUM_ROWS*NUM_COLS) :: JC

icount=1
do i=1,NUM_COLS
  do j=1,NUM_ROWS
    data_2d(j,i)=icount
    icount=icount+1
  enddo
enddo

pointer=0

do i=1,NUM_COLS
  do j=1,NUM_ROWS
    if (data_2d(j,i).eq.0) cycle
      pointer=pointer+1
      data_1d(pointer)=data_2d(j,i)
      JR(pointer)=j
      JC(pointer)=i
  enddo
enddo

write(6,*) 'pointer=',pointer
do i=1,pointer
  write(6,*) i,data_1d(i),JR(i),JC(i)
enddo

stop
end Program serial_index
```

Compiling this program and running it yields the following results, which match with the figures from the previous section.

```
pointer= 9.000000
1 1.000000 1.000000 1.000000
2 2.000000 2.000000 1.000000
3 3.000000 3.000000 1.000000
4 4.000000 1.000000 2.000000
5 5.000000 2.000000 2.000000
6 6.000000 3.000000 2.000000
7 7.000000 1.000000 3.000000
8 8.000000 2.000000 3.000000
9 9.000000 3.000000 3.000000
```

Since the matrix is full of non-zero values, this represents a trivial case. Changing the initialization section to more closely represent a sparse matrix such as:

```
data_2d=0
data_2d(1,1)=1
data_2d(2,1)=2
data_2d(2,2)=5
data_2d(1,3)=7
data_2d(3,3)=9
```

would result in the following output:

```
pointer= 5.000000
1 1.000000 1.000000 1.000000
2 2.000000 2.000000 1.000000
3 5.000000 2.000000 2.000000
4 7.000000 1.000000 3.000000
5 9.000000 3.000000 3.000000
```

3.5.4 Parallel Implementation of an Indexing Algorithm

We utilize OpenMP directives to parallelize our indexing algorithm and gain *speedup* on the double nested loop. The modified program with the OpenMP directives is as follows:

```
Program parallel_index

  integer,parameter:: NUM_ROWS=3
  integer,parameter:: NUM_COLS=3

  REAL,dimension(NUM_ROWS,NUM_COLS) :: data_2d
  REAL, dimension(NUM_ROWS*NUM_COLS) :: data_1d
  REAL, dimension(NUM_ROWS*NUM_COLS) :: JR
  REAL, dimension(NUM_ROWS*NUM_COLS) :: JC
  integer :: pointer
  integer :: omp_get_thread_num

  icount=1
  do i=1,NUM_COLS
    do j=1,NUM_ROWS
      data_2d(j,i)=icount
      icount=icount+1
    enddo
  enddo

  pointer=0

!$OMP PARALLEL default(shared)
!$OMP DO private(i,j,myid)
  do i=1,NUM_COLS
    myid=omp_get_thread_num()
    write(6,*) "Thread=",myid," loop=",i
    do j=1,NUM_ROWS
      if (data_2d(j,i).eq.0) cycle
      pointer=pointer+1
      data_1d(pointer)=data_2d(j,i)
      JR(pointer)=j
      JC(pointer)=i
    enddo
  enddo
!$OMP END DO
!$OMP END PARALLEL

  do i=1,pointer
    write(6,*) i,data_1d(i),JR(i),JC(i)
  enddo

  stop
end Program parallel_index
```

Compiling and running this program yields the following results:

```
Thread=           0 loop=           1
Thread=           1 loop=           3
Thread=           0 loop=           2
         1   1.000000       1.000000       1.000000
         2   2.000000       2.000000       1.000000
         3   3.000000       3.000000       1.000000
         4   7.000000       1.000000       3.000000
         5   8.000000       2.000000       3.000000
         6   9.000000       3.000000       3.000000
         7   4.000000       1.000000       2.000000
         8   5.000000       2.000000       2.000000
         9   6.000000       3.000000       2.000000
```

You can see by looking at the first two columns of the output that the new reduced data did not get generated in the same order as the serial program.

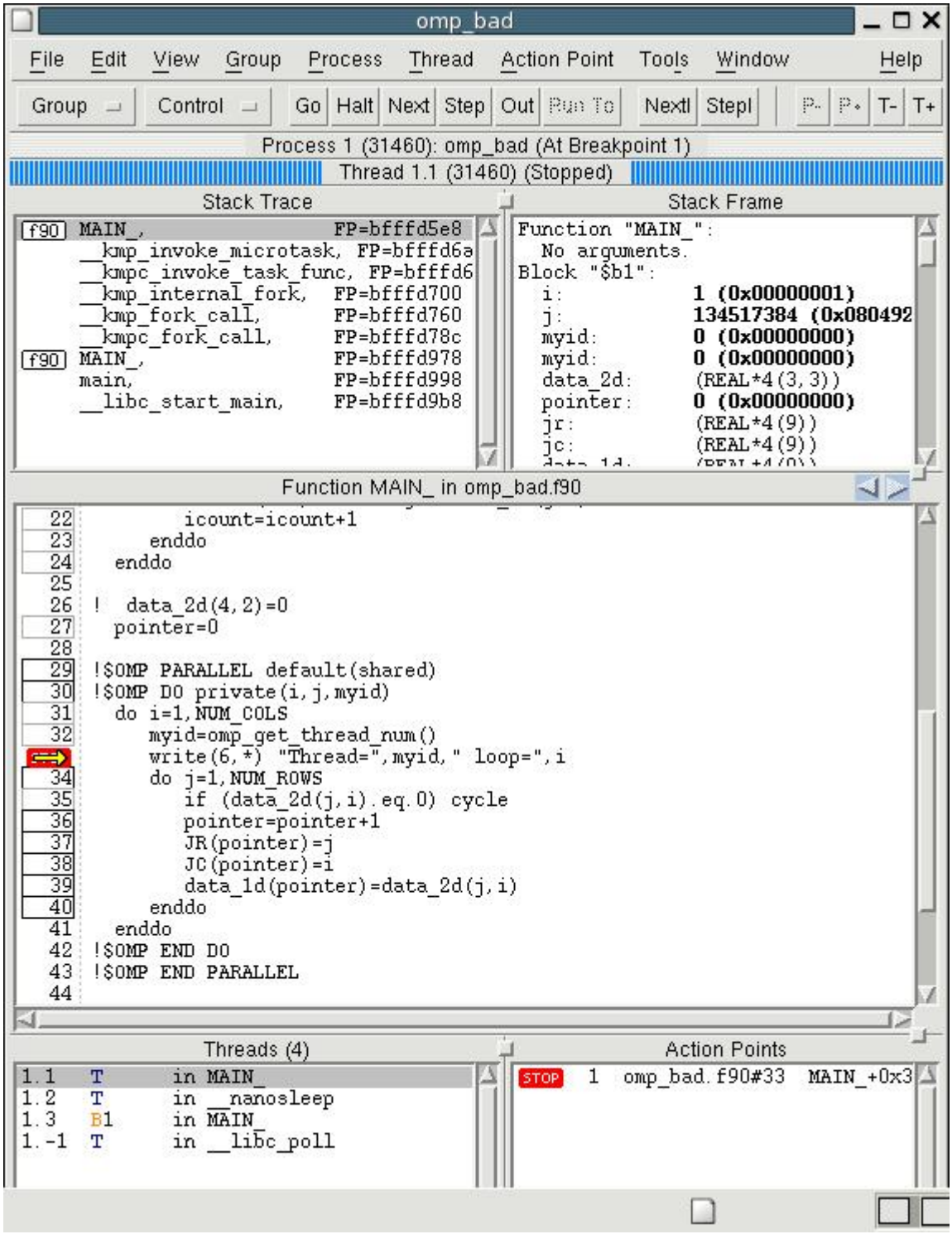
Debugging the Sample Code

You could use print statements and some deductive reasoning to determine the cause of the incorrect ordering error in our parallel sample code. However, an easier method is to use a debugger. Here we use the Totalview® debugger.

Compiling our sample code with debugging turned on and OpenMP enabled yields a multi-threaded executable that can be debugged. To look at the problem initially, a break statement is put at the line

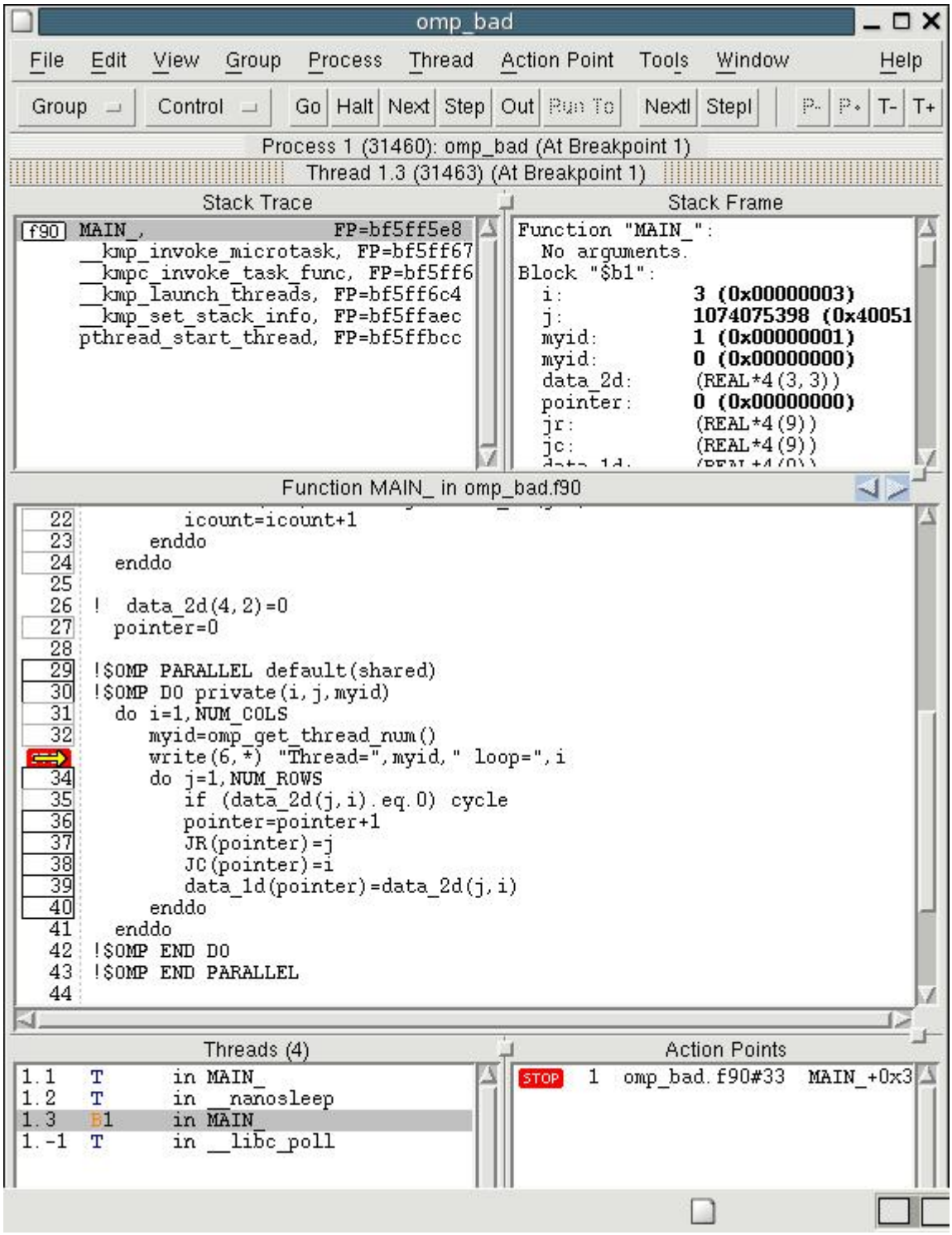
```
write(6,*) "Thread=",myid," loop=",i
```

Since this line is right after the loop that is parallelized with the OpenMP directive, we stop at this point to monitor the code variables for each thread. A screenshot of the debugger when it hits the first break point is given below.



Looking at the header bar, you can see that we are looking at thread 1 (or the first process). We can see that the *i* loop, which was declared private, has been initialized to the value of the first loop iteration. The serial process loops down the first column, processes all of the non-zero elements, moves to the second column, and so on.

To switch to the second thread, we click on the "T+" button in the top right hand side of the screen. A screen similar to the following is displayed:



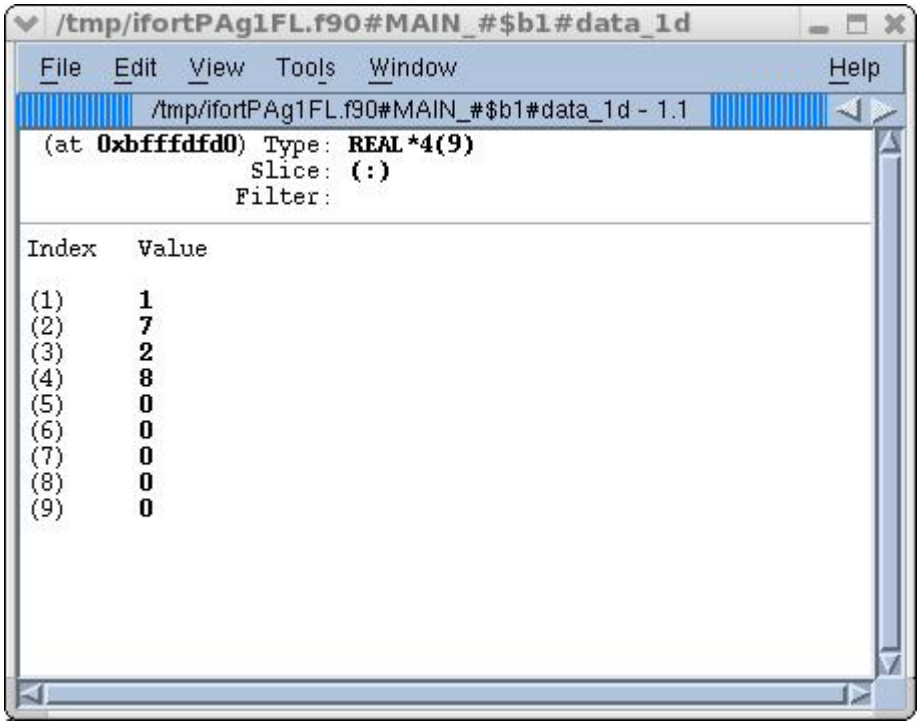
Looking at the second thread, the critial point to note is how the loop index for the OpenMP parallelized *i* loop is displayed. Here we see that *i* is set to 3 and that the value 2 has been skipped. This is because the default OpenMP schedule policy has been implemented. Being a "chunk" policy, even sized chunks of the loop are distributed to each process. This gives the coarsest decomposition and generally minimizes synchronization and shared memory communication.

If we continued through this iteration and went to the next one, we would see that thread 1 would get the *i=2* loop. The program assigned the first half of the *i* loop (values 1 and 2) to the first processor and then assigned the second half of the *i* loop (value 3) to the second processor. Since both processors started working concurrently, the *i=1* loop and *i=3* loop were executed before the *i=2* loop. This caused the first column to be processed, then the third column, and finally the second. Looking again at the program output, this matches how the reduced data was assembled out of order. Note that it is possible to deduce that the threads processed the columns out of order by looking at the program output, but the debugger made it very easy and clear to see that at one time, one thread was working on column 1 and the other thread

was working on column 3.

Another useful technique is to monitor a shared variable in an external window. If the breakpoint in our example was changed to line 40, the end of the inner loop, the debugger will effectively stop after each iteration of the inner loop. Loops where the two-dimensional matrix value is non-zero will involve an update of the *data_1d* array. In Totalview®, double clicking on the variable in the "Stack Frame" window opens up a new window and displays the contents of that variable. You can leave this window open and, as long as you stay within the scope in which the variable exists, its value will be updated automatically.

Below is a screenshot of the window opened for the *data_1d* array in the above program.



The breakpoint was placed at line 30 and several loop iterations were performed. You can see that the array is not getting assembled in the anticipated order.

3.5.5 Correcting the Parallel Algorithm

Now that using the debugger has pinpointed the problem with our parallel algorithm, the code can be revisited to determine the best solution to solve the problem.

Modified Parallel Indexing Algorithm

In our original code, we inserted OpenMP directives to gain parallel speedup on the double nested loop. Program output showed that there was a change in behavior. Utilizing the debugger, we found that the update process, the process by which non-zero values were placed into the reduced data arrays, had a serial dependency and could not be executed out of order.

The best method to resolve the dependency in this situation is to separate out the serial update of the reduced array. This involves two steps. Keeping in mind that the inner *j* loop processes one column, we allow each column to be processed independently by storing the reduced data in a temporary array. The reduced data from each column is then assembled into the final reduced array in serial order. While this latter stage will not result in parallel speedup, we are only serializing the reduced assembly, which involves fewer array elements than the full array. Experience has been that even with this serial component, near linear speedup can be seen up to four threads.

The modified program is given below.

```
Program omp_good

  integer,parameter:: NUM_ROWS=3
  integer,parameter:: NUM_COLS=3

  REAL,dimension(NUM_ROWS,NUM_COLS) :: data_2d
```

```
  REAL, dimension(NUM_ROWS*NUM_COLS) :: data_1d
  REAL, dimension(NUM_COLS) :: data_local
  REAL, dimension(NUM_COLS) :: JR_local
  REAL, dimension(NUM_ROWS*NUM_COLS) :: JR
  REAL, dimension(NUM_ROWS*NUM_COLS) :: JC

  integer :: pointer,local_ptr

  integer :: omp_get_thread_num

  data_1d=0.0
  JR=0
  JC=0
  icount=1
  do i=1,NUM_COLS
    do j=1,NUM_ROWS
      data_2d(j,i)=icount
      write(6,*) icount,j,i,data_2d(j,i)
      icount=icount+1
    enddo
  enddo

  pointer=0

!$OMP PARALLEL default(shared)
!$OMP DO private(i,j,local_ptr,data_local,JR_local) ordered schedule(static,1)
  do i=1,NUM_COLS
    myid=omp_get_thread_num()
    write(6,*) "Thread=",myid," loop=",i
    local_ptr=0
    do j=1,NUM_ROWS
      if (data_2d(j,i).eq.0) cycle
      local_ptr=local_ptr+1
      data_local(local_ptr)=data_2d(j,i)
      JR_local(local_ptr)=j
    enddo
!$OMP ORDERED
!$OMP CRITICAL
  do j=1,local_ptr
    pointer=pointer+1
    data_1d(pointer)=data_local(j)
    JR(pointer)=JR_local(j)
    JC(pointer)=i
  enddo
!$OMP END CRITICAL
!$OMP END ORDERED

  enddo
!$OMP END DO
!$OMP END PARALLEL

  do i=1,pointer
     write(6,*) i,data_1d(i),JR(i),JC(i)
  enddo

  stop
end Program omp_good
```

We specified two new OpenMP options in our modified code. First, even though the columns of the two-dimensional matrix can be processed independently, they still must be assembled serially. It is possible to reduce all of the data into temporary variables and then do the final assembly at one time. In this algorithm, a slightly different approach is taken where each thread only moves on to another column once its current data has been written to the reduced data array.

To achieve this, the "ordered" option is added to the OpenMP loop. This works in conjunction with the !$OMP ORDERED directive. When an OpenMP loop is specified with the ordered option, threads will enter the !$OMP ORDERED section of the code based on the order of their thread id, i.e., the first thread to enter the code block will be thread 0, The second will be thread 1, and so on.

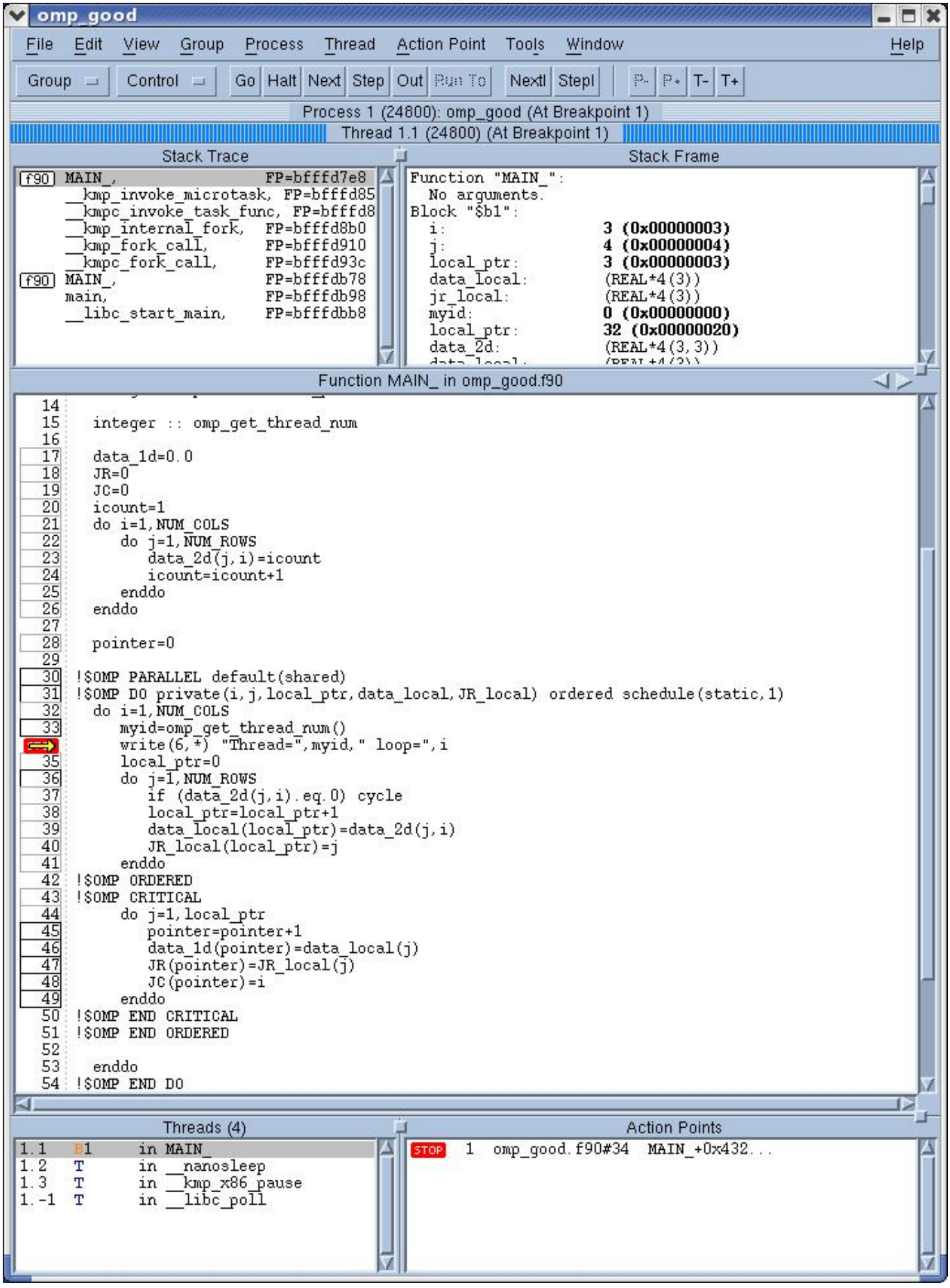
This combined with the !$OMP CRITICAL directive allows us to control the order in which threads update the reduced data arrays. The loop scheduling was also set to static with a granularity of 1. The loop iterations are scheduled incrementally on each processor. Based on the number of processors available, that many columns will be processed in parallel. Then the code enters the ORDERED and CRITICAL section, which means that the first thread will process its reduced data and then the second thread will process its reduced data. This continues for as many threads that are in existence. Once done, they go back and work on the next set of columns. Again, this adds more overhead to the process, but good speedup has been seen on smaller SMP systems.

Compiling and running our modified sample program yields the following results:

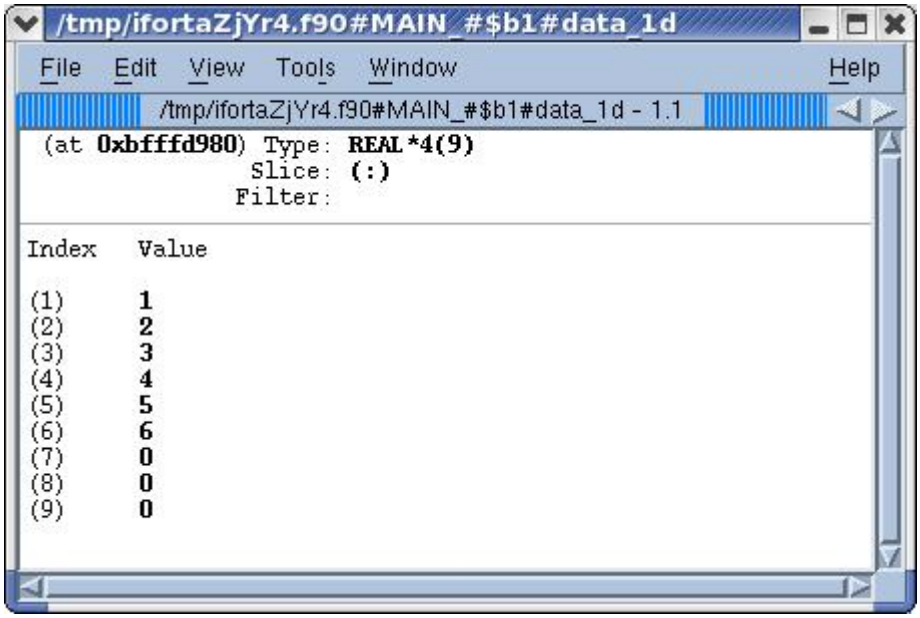
```
Thread= 1 loop= 1
Thread= 0 loop= 3
Thread= 0 loop= 2
1 1.000000 1.000000 1.000000
2 2.000000 2.000000 1.000000
3 3.000000 3.000000 1.000000
4 4.000000 1.000000 2.000000
5 5.000000 2.000000 2.000000
6 6.000000 3.000000 2.000000
7 7.000000 1.000000 3.000000
8 8.000000 2.000000 3.000000
9 9.000000 3.000000 3.000000
```

You can see by looking at the first two columns of the output that the new reduced data is generated in correct order.

It is insightful to use Totalview® on the modified code to see how the shared variables are now updated and how the debugger behaves with the addition of the CRITICAL directive. In the screenshot below, the breakpoint is left outside the inner loop. One specific behavioral change is that due to the CRITICAL and ORDERED sections, some of the threads may be blocked at the entry to the section. When a thread is blocked at at CRITICAL or ORDERED section, its program counter is within the OpenMP library and therefore displays the assembly code of the library and not the Fortran or C program. This usually implies that the thread of interest has to specifically be selected and monitored. Switching from one thread to the next to look at variables is more difficult when CRITICAL and ORDERED sections are included.



To confirm that the assembly of the reduced data array is being processed correctly, a window showing the status of the *data_1d* array can be opened while switching back and forth between threads. While jumping between threads and stepping through the program, you can see that the array is being updated correctly.



The solution presented by our modified program requires a little more synchronization, but minimizes the amount of temporary arrays and, hence, memory that needs to be allocated.

3.5.6 Debugging Exercise

While the corrected parallel algorithm for the example problem does produce the same results as the serial algorithm, it might be of interest to look at the performance of the parallel loop. Upon closer examination, the "!$OMP CRITICAL" directive appears to have a high overhead. To attempt to improve parallel scalability by eliminating some of the overhead, the CRITICAL directive can be eliminated as well as the static scheduling directive, resulting in the program shown below:

```
Program modification2

 integer,parameter:: NUM_ROWS=30
 integer,parameter:: NUM_COLS=30

 REAL,dimension(NUM_ROWS,NUM_COLS) :: data_2d
 REAL, dimension(NUM_ROWS*NUM_COLS) :: data_1d
 REAL, dimension(NUM_COLS) :: data_local
 REAL, dimension(NUM_COLS) :: JR_local
 REAL, dimension(NUM_ROWS*NUM_COLS) :: JR
 REAL, dimension(NUM_ROWS*NUM_COLS) :: JC

 integer :: pointer,local_ptr

 integer :: omp_get_thread_num

 data_1d=0.0
 JR=0
 JC=0
 icount=1
 do i=1,NUM_COLS
 do j=1,NUM_ROWS
 data_2d(j,i)=icount
 icount=icount+1
 enddo
 enddo

 pointer=0

!$OMP PARALLEL default(shared)
!$OMP DO private(i,j,local_ptr,data_local,JR_local) ordered
 do i=1,NUM_COLS
 myid=omp_get_thread_num()
 write(6,*) "Thread=",myid," loop=",i
 local_ptr=0
 do j=1,NUM_ROWS
```

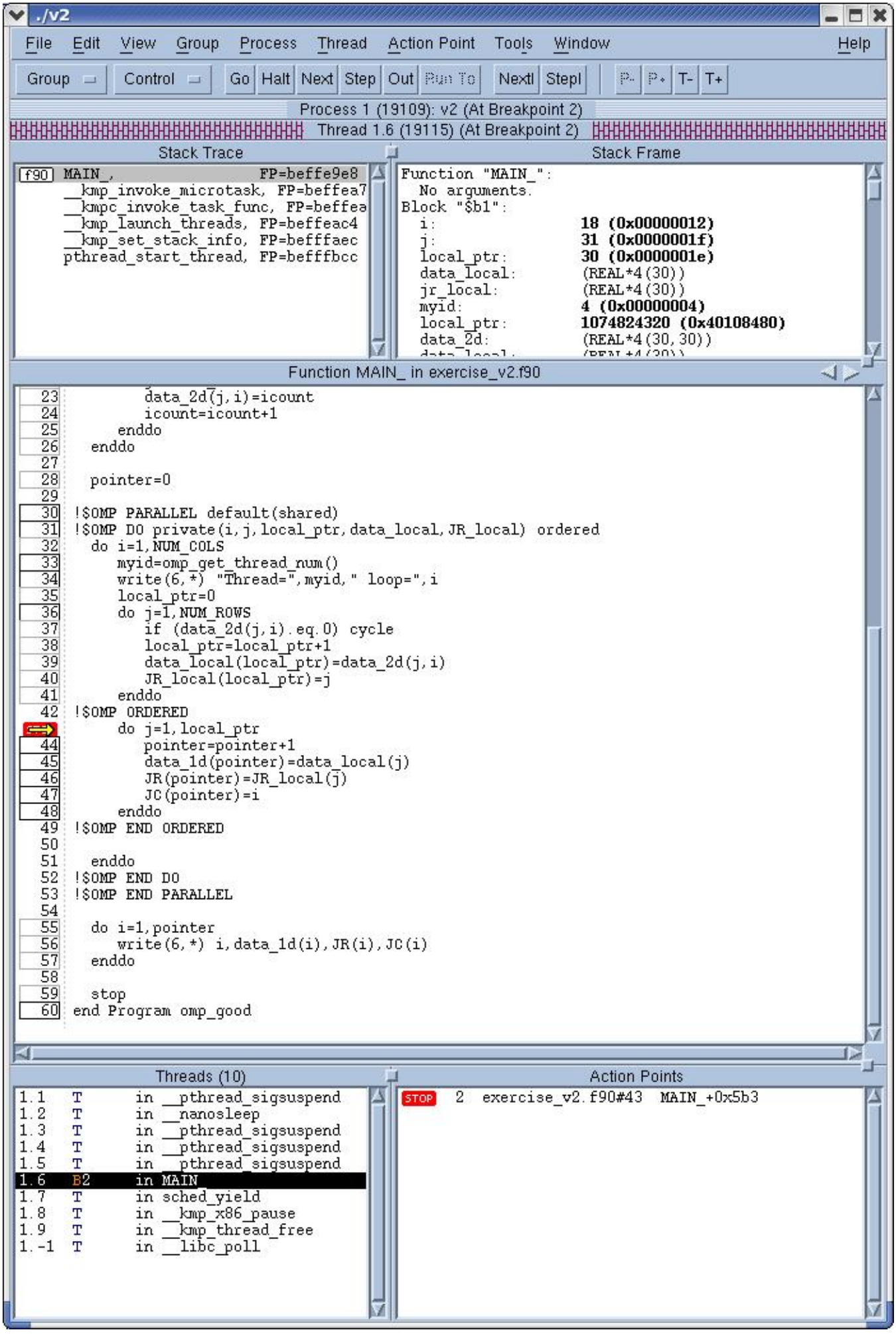
```
 if (data_2d(j,i).eq.0) cycle
 local_ptr=local_ptr+1

data_local(local_ptr)=data_2d(j,i)
 JR_local(local_ptr)=j
 enddo
!$OMP ORDERED
 do j=1,local_ptr
 pointer=pointer+1

data_1d(pointer)=data_local(j)
 JR(pointer)=JR_local(j)
 JC(pointer)=i
 enddo
!$OMP END ORDERED

 enddo
!$OMP END DO
!$OMP END PARALLEL

 do i=1,pointer
 write(6,*) i,data_1d(i),JR(i),JC(i)
 enddo

 stop
end Program modification2
```

For this exercise, determine how to use the debugger to verify that the updating of the shared arrays and shared pointer are being processed correctly in this modified code.

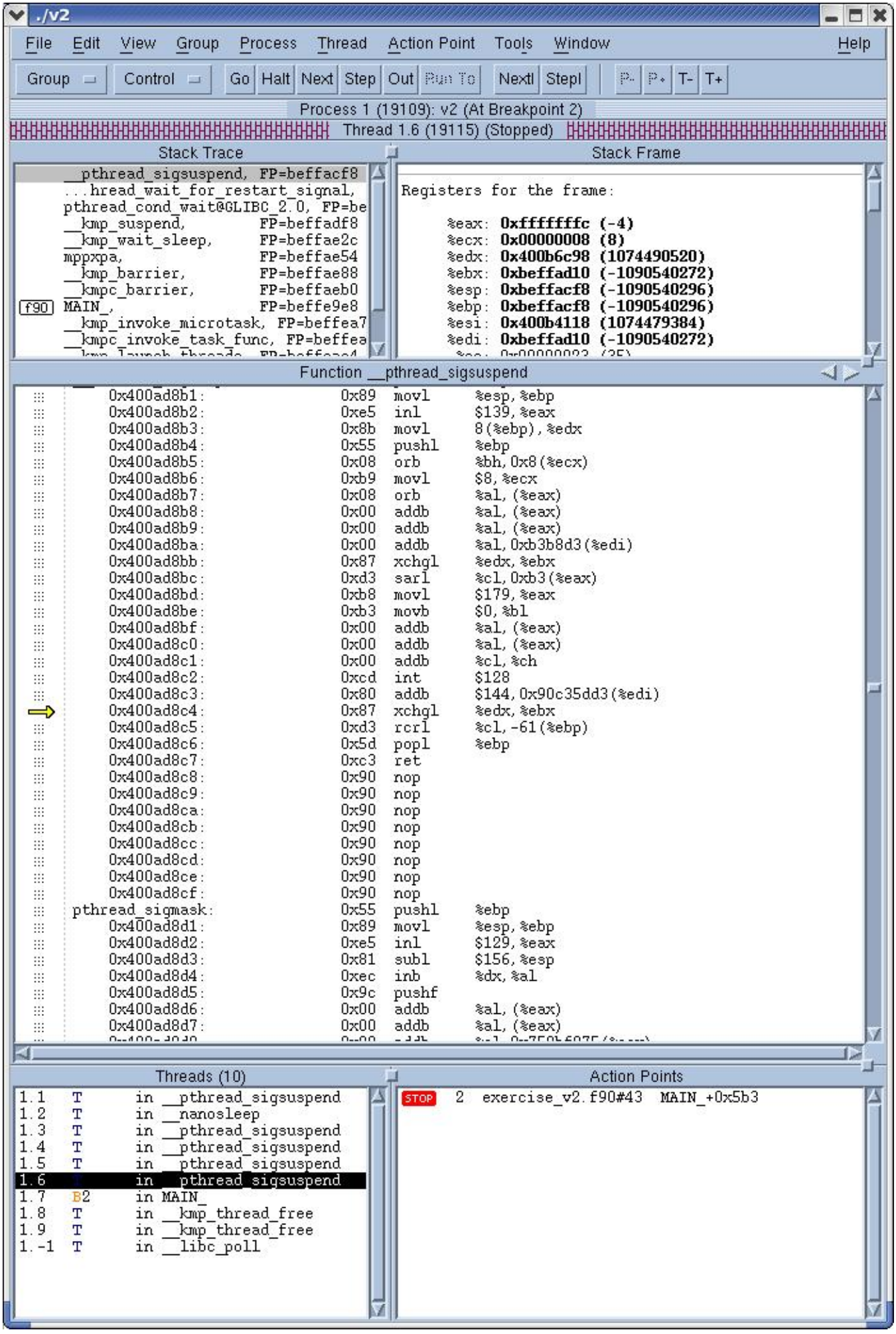
3.5.7 Exercise Solution

To use the debugger to verify that the updating of the shared arrays and shared pointer are being processed correctly, we first monitor the *data_1d* array as the program progresses to see how it is being assembled by the multiple threads. This is best done within the ORDERED region.

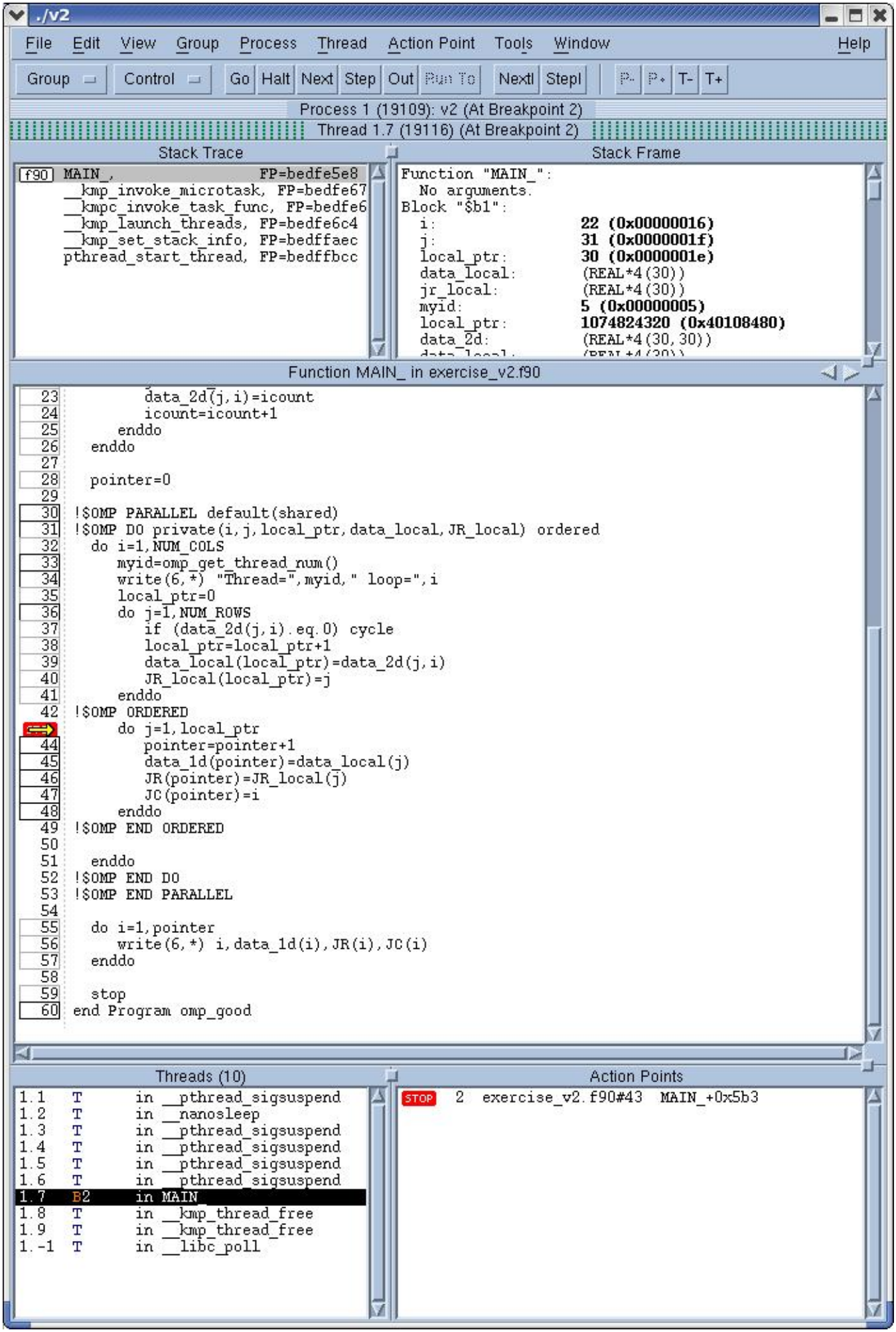
We run the application in Totalview® with 8 threads and a 30x30 matrix to make the program larger and easier to see possible parallel problems. A breakpoint was set at the entrance to the loop within the ORDERED section. The screenshot shown below was obtained for thread id 4 and column 18. You can see by looking at the "Threads" window that only one thread is at the routine and all of the other threads are in various wait states.


./v2
File Edit View Group Process Thread Action Point Tools Window Help
Group Control Go Halt Next Step Out Run To Nextl Stepl P- P+ T- T+
Process 1 (19109): v2 (At Breakpoint 2)
Thread 1.6 (19115) (At Breakpoint 2)
Stack Trace
f90 MAIN_, FP=beffe9e8
__kmp_invoke_microtask, FP=beffea7
__kmpc_invoke_task_func, FP=beffea
__kmp_launch_threads, FP=beffeac4
__kmp_set_stack_info, FP=befffaec
pthread_start_thread, FP=befffbcc
Stack Frame
Function "MAIN_":
No arguments.
Block "$b1":
i: 18 (0x00000012)
j: 31 (0x0000001f)
local_ptr: 30 (0x0000001e)
data_local: (REAL*4(30))
jr_local: (REAL*4(30))
myid: 4 (0x00000004)
local_ptr: 1074824320 (0x40108480)
data_2d: (REAL*4(30,30))
Function MAIN_ in exercise_v2.f90
23 data_2d(j,i)=icount
24 icount=icount+1
25 enddo
26 enddo
27
28 pointer=0
29
30 !$OMP PARALLEL default(shared)
31 !$OMP DO private(i,j,local_ptr,data_local,JR_local) ordered
32 do i=1,NUM_COLS
33 myid=omp_get_thread_num()
34 write(6,*) "Thread=",myid," loop=",i
35 local_ptr=0
36 do j=1,NUM_ROWS
37 if (data_2d(j,i).eq.0) cycle
38 local_ptr=local_ptr+1
39 data_local(local_ptr)=data_2d(j,i)
40 JR_local(local_ptr)=j
41 enddo
42 !$OMP ORDERED
do j=1,local_ptr
44 pointer=pointer+1
45 data_1d(pointer)=data_local(j)
46 JR(pointer)=JR_local(j)
47 JC(pointer)=i
48 enddo
49 !$OMP END ORDERED
50
51 enddo
52 !$OMP END DO
53 !$OMP END PARALLEL
54
55 do i=1,pointer
56 write(6,*) i,data_1d(i),JR(i),JC(i)
57 enddo
58
59 stop
60 end Program omp_good
Threads (10)
1.1 T in __pthread_sigsuspend
1.2 T in __nanosleep
1.3 T in __pthread_sigsuspend
1.4 T in __pthread_sigsuspend
1.5 T in __pthread_sigsuspend
1.6 B2 in MAIN
1.7 T in sched_yield
1.8 T in __kmp_x86_pause
1.9 T in __kmp_thread_free
1.-1 T in __libc_poll
Action Points
STOP 2 exercise_v2.f90#43 MAIN_+0x5b3

By clicking the 'Go' button several times, you can see by examining the variable *i* that sequential columns are processed into the *data_1d* array. When the next value of *i* will be processed on another thread, the current thread will exit the ORDERED region and allow the thread with the next iteration to enter. When this happens, because Totalview® is still focused on the thread that just left the ORDERED section, the thread goes into an OMP wait at the !$OMP ORDERED directive while the thread that has the next iteration has moved into the ORDERED secton and is held at the breakpoint. This is illustrated in the following screenshot:


./v2
File Edit View Group Process Thread Action Point Tools Window Help
Group Control Go Halt Next Step Out Run To Nexti Stepi P- P+ T- T+
Process 1 (19109): v2 (At Breakpoint 2)
Thread 1.6 (19115) (Stopped)
Stack Trace
__pthread_sigsuspend, FP=beffacf8
...hread_wait_for_restart_signal,
pthread_cond_wait@GLIBC_2.0, FP=be
__kmp_suspend, FP=beffadf8
__kmp_wait_sleep, FP=beffae2c
mppxpa, FP=beffae54
__kmp_barrier, FP=beffae88
__kmpc_barrier, FP=beffaeb0
f90 MAIN_, FP=beffe9e8
__kmp_invoke_microtask, FP=beffea7
__kmpc_invoke_task_func, FP=beffea
Stack Frame
Registers for the frame:
%eax: 0xfffffffc (-4)
%ecx: 0x00000008 (8)
%edx: 0x400b6c98 (1074490520)
%ebx: 0xbeffad10 (-1090540272)
%esp: 0xbeffacf8 (-1090540296)
%ebp: 0xbeffacf8 (-1090540296)
%esi: 0x400b4118 (1074479384)
%edi: 0xbeffad10 (-1090540272)
Function __pthread_sigsuspend
0x400ad8b1: 0x89 movl %esp,%ebp
0x400ad8b2: 0xe5 inl $139,%eax
0x400ad8b3: 0x8b movl 8(%ebp),%edx
0x400ad8b4: 0x55 pushl %ebp
0x400ad8b5: 0x08 orb %bh,0x8(%ecx)
0x400ad8b6: 0xb9 movl $8,%ecx
0x400ad8b7: 0x08 orb %al,(%eax)
0x400ad8b8: 0x00 addb %al,(%eax)
0x400ad8b9: 0x00 addb %al,(%eax)
0x400ad8ba: 0x00 addb %al,0xb3b8d3(%edi)
0x400ad8bb: 0x87 xchgl %edx,%ebx
0x400ad8bc: 0xd3 sarl %cl,0xb3(%eax)
0x400ad8bd: 0xb8 movl $179,%eax
0x400ad8be: 0xb3 movb $0,%bl
0x400ad8bf: 0x00 addb %al,(%eax)
0x400ad8c0: 0x00 addb %al,(%eax)
0x400ad8c1: 0x00 addb %cl,%ch
0x400ad8c2: 0xcd int $128
0x400ad8c3: 0x80 addb $144,0x90c35dd3(%edi)
0x400ad8c4: 0x87 xchgl %edx,%ebx
0x400ad8c5: 0xd3 rcrl %cl,-61(%ebp)
0x400ad8c6: 0x5d popl %ebp
0x400ad8c7: 0xc3 ret
0x400ad8c8: 0x90 nop
0x400ad8c9: 0x90 nop
0x400ad8ca: 0x90 nop
0x400ad8cb: 0x90 nop
0x400ad8cc: 0x90 nop
0x400ad8cd: 0x90 nop
0x400ad8ce: 0x90 nop
0x400ad8cf: 0x90 nop
pthread_sigmask: 0x55 pushl %ebp
0x400ad8d1: 0x89 movl %esp,%ebp
0x400ad8d2: 0xe5 inl $129,%eax
0x400ad8d3: 0x81 subl $156,%esp
0x400ad8d4: 0xec inb %dx,%al
0x400ad8d5: 0x9c pushf
0x400ad8d6: 0x00 addb %al,(%eax)
0x400ad8d7: 0x00 addb %al,(%eax)
Threads (10)
1.1 T in __pthread_sigsuspend
1.2 T in __nanosleep
1.3 T in __pthread_sigsuspend
1.4 T in __pthread_sigsuspend
1.5 T in __pthread_sigsuspend
1.6 T in __pthread_sigsuspend
1.7 B2 in MAIN_
1.8 T in __kmp_thread_free
1.9 T in __kmp_thread_free
1.-1 T in __libc_poll
Action Points
STOP 2 exercise_v2.f90#43 MAIN_+0x5b3

At this point, click to the thread that is currently in the ORDERED region. To do this, simply click on the thread that is listed as being in the "MAIN" section. In this case, you want to switch from thread 1.6 to thread 1.7 by clicking on the thread 1.7 line in the Threads window. After doing this, the following screen is displayed:


./v2
File Edit View Group Process Thread Action Point Tools Window Help
Group Control Go Halt Next Step Out Run To Nextl Stepl P- P+ T- T+
Process 1 (19109): v2 (At Breakpoint 2)
Thread 1.7 (19116) (At Breakpoint 2)
Stack Trace
f90 MAIN_, FP=bedfe5e8
__kmp_invoke_microtask, FP=bedfe67
__kmpc_invoke_task_func, FP=bedfe6
__kmp_launch_threads, FP=bedfe6c4
__kmp_set_stack_info, FP=bedffaec
pthread_start_thread, FP=bedffbcc
Stack Frame
Function "MAIN_":
No arguments.
Block "$b1":
i: 22 (0x00000016)
j: 31 (0x0000001f)
local_ptr: 30 (0x0000001e)
data_local: (REAL*4(30))
jr_local: (REAL*4(30))
myid: 5 (0x00000005)
local_ptr: 1074824320 (0x40108480)
data_2d: (REAL*4(30,30))
Function MAIN_ in exercise_v2.f90
23 data_2d(j,i)=icount
24 icount=icount+1
25 enddo
26 enddo
27
28 pointer=0
29
30 !$OMP PARALLEL default(shared)
31 !$OMP DO private(i,j,local_ptr,data_local,JR_local) ordered
32 do i=1,NUM_COLS
33 myid=omp_get_thread_num()
34 write(6,*) "Thread=",myid," loop=",i
35 local_ptr=0
36 do j=1,NUM_ROWS
37 if (data_2d(j,i).eq.0) cycle
38 local_ptr=local_ptr+1
39 data_local(local_ptr)=data_2d(j,i)
40 JR_local(local_ptr)=j
41 enddo
42 !$OMP ORDERED
do j=1,local_ptr
44 pointer=pointer+1
45 data_1d(pointer)=data_local(j)
46 JR(pointer)=JR_local(j)
47 JC(pointer)=i
48 enddo
49 !$OMP END ORDERED
50
51 enddo
52 !$OMP END DO
53 !$OMP END PARALLEL
54
55 do i=1,pointer
56 write(6,*) i,data_1d(i),JR(i),JC(i)
57 enddo
58
59 stop
60 end Program omp_good
Threads (10)
1.1 T in __pthread_sigsuspend
1.2 T in __nanosleep
1.3 T in __pthread_sigsuspend
1.4 T in __pthread_sigsuspend
1.5 T in __pthread_sigsuspend
1.6 T in __pthread_sigsuspend
1.7 B2 in MAIN
1.8 T in __kmp_thread_free
1.9 T in __kmp_thread_free
1.-1 T in __libc_poll
Action Points
STOP 2 exercise_v2.f90#43 MAIN_+0x5b3

We can now see that we jumped to the next thread, with id 5, and the value of *i* has gone to the next iteration, which in this case is column 22.

3.6 Serial or Ordered Calculation Errors

3.6.1 Serial or Ordered Calculation Errors

Introduction

A common problem in OpenMP programs is the potential for conflict between threads trying to read or write from a common memory location simultaneously. This is analogous to the more common problem of concurrency control in multi-user databases. When this type of error occurs, program output is inconsistent for multiple runs and, needless to say, results in incorrect calculations.

Objectives

In this lesson, you will learn about serial or ordered calculation errors in OpenMP and how to identify them using the Totalview® debugger. A sample Fortran program is provided that you may debug on your own while following along with the procedure described here.

3.6.2 Sample Program

The following sample program performs some basic statistical calculations on a list of numbers. It calculates the maximum value in the list and where in the list it occurs. It also calculates the arithmetic mean of the list. Parallelism is obtained by the use of OpenMP, and it is designed to be run on four threads. Each thread gets 75 of the 300 numbers in the list, sums them, scans through them to see if any are higher than the global maxima, and writes the sum and maxima to the shared variables, *the_max* and *the_mean*.

```
PROGRAM parallel_stats


!  This program has been designed for the purposes of demonstrating common bugs and
!  is deliberately limited and inefficient.

use omp_lib

parameter (N=300)
  integer I
  integer myid,istart,iend,nthreads,nper
  real the_max, b(N),tmp_max, the_mean, tmp

   OPEN(unit=10,file="b.data")
   OPEN(unit=12,file="found.results")

 DO i=1,300
    READ(10,*) b(i)
 END DO

the_max = 0.0

the_mean = 0.0

!$omp parallel private(myid, istart, iend, tmp_max, tmp)

nthreads = OMP_GET_NUM_THREADS()
nper = N/nthreads
tmp=0.0

myid = OMP_GET_THREAD_NUM()

istart = myid*nper + 1
iend = istart + nper - 1

      tmp_max = the_max

      DO I=istart,iend
         tmp = tmp+b(I)
```

```
           IF(b(I).GT.tmp_max) THEN
             tmp_max=b(I)
             WRITE(12,*)"New maximum",b(I)," found at position",I
           END IF
      END DO

      the_mean = the_mean + tmp

      IF (the_mean.EQ.tmp_max) THEN
        WRITE(12,*)"Data is Uniform"
      END IF

      the_max = tmp_max

!$omp end parallel

      the_mean = the_mean / N

    WRITE(12,*)"The maximum is", the_max, "and the mean is",the_mean

END PROGRAM parallel_stats
```

The input data file is b.data.

On an AIX based system the program can be compiled with:

```
% xlf90_r -qsmp=omp -g -o omp_max.exe O0 openmp_max.f
```

Set the number of required threads to 4 with:

```
% setenv OMP_NUM_THREADS 4
```

or use the equivalent for your shell if it is different than tcsh.

Running the program results in a file "found.results":

```
% cat found.results
 New maximum, 10.00000000 , found at position 1
 New maximum, 35.00000000 , found at position 2
 New maximum, 42.00000000 , found at position 8
 New maximum, 11.00000000 , found at position 231
 New maximum, 49.00000000 , found at position 233
 New maximum, 13.00000000 , found at position 76
 New maximum, 9.000000000 , found at position 151
 New maximum, 14.00000000 , found at position 152
 New maximum, 26.00000000 , found at position 153
 New maximum, 35.00000000 , found at position 154
 New maximum, 41.00000000 , found at position 155
 New maximum, 48.00000000 , found at position 165
 New maximum, 82.00000000 , found at position 200
 New maximum, 45.00000000 , found at position 78
 New maximum, 49.00000000 , found at position 95
 New maximum, 135.0000000 , found at position 100
 New maximum, 45.00000000 , found at position 13
 New maximum, 50.00000000 , found at position 14
 The maximum is 50.00000000 and the mean is -0.5333333611
```

This output should look very suspicious as the reported maximum of 50 is less than some of the intermediate maxima.

3.6.3 Debugging the Sample Program

It is always a good idea when running an OpenMP program to run it single threaded to see that the parallelism has not led to a change in the output. Therefore, this is our first debugging step:

```
% setenv OMP_NUM_THREADS 1

% cat found.results
 New maximum, 10.00000000 , found at position 1
 New maximum, 35.00000000 , found at position 2
```

```
 New maximum, 42.00000000 , found at position 8
 New maximum, 45.00000000 , found at position 13
 New maximum, 50.00000000 , found at position 14
 New maximum, 135.0000000 , found at position 100
 The maximum is 135.0000000 and the mean is -0.5333333611
```

The mean is the same in both cases, which is good but the maximum has changed indicating a serious problem. Another simple debugging step that is often helpful in finding OpenMP errors is to run multiple times in parallel to see if the results are consistent. For ease of comparison, add an extra line to the output to send the final result to the screen as well as the file:

```
WRITE(*,*)"The maximum is", the_max, "and the mean is",the_mean
```

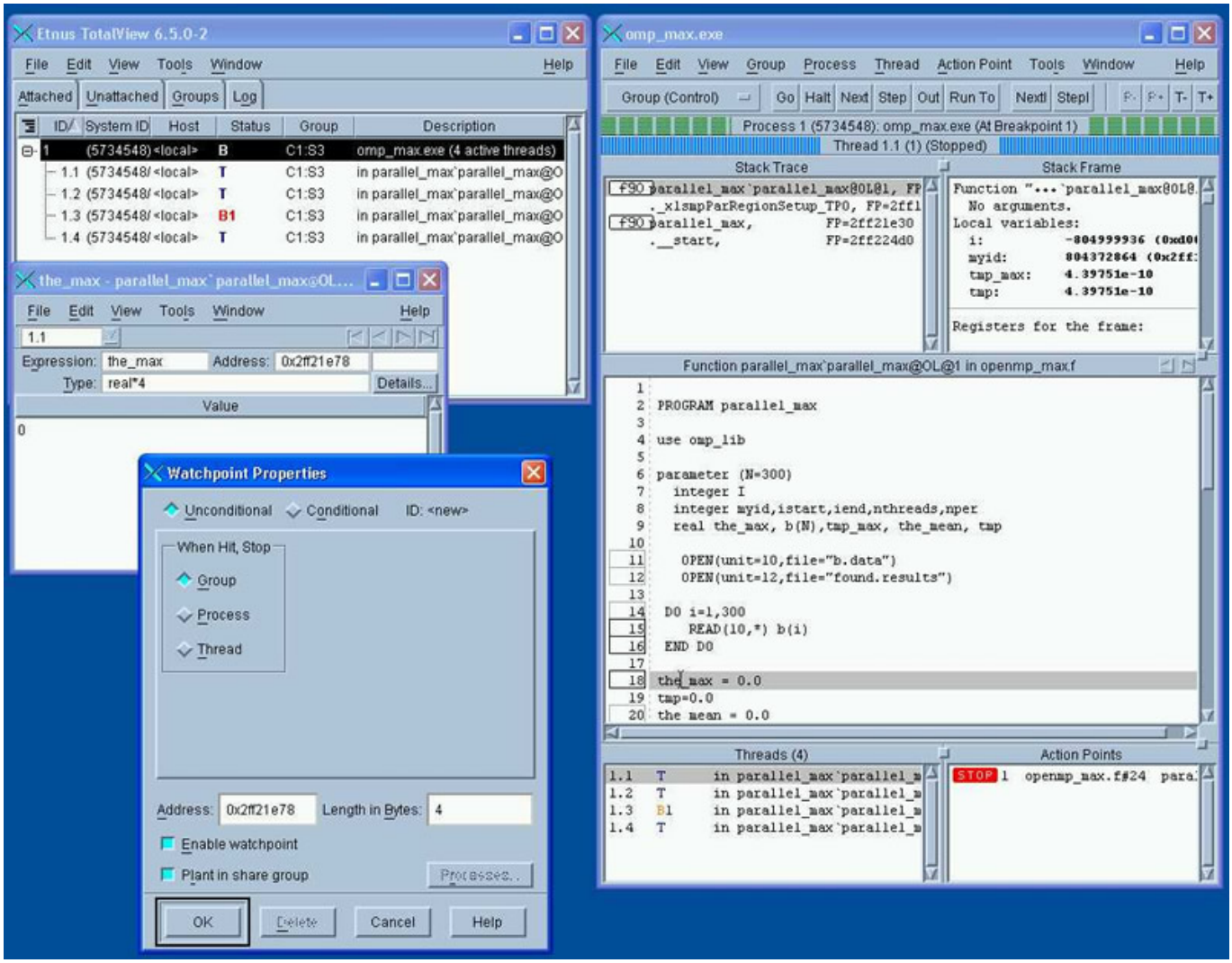
```
% setenv OMP_NUM_THREADS 4
% omp_max.exe
 The maximum is 135.0000000 and the mean is -0.5333333611
% omp_max.exe
 The maximum is 49.00000000 and the mean is -0.5333333611
% omp_max.exe
 The maximum is 135.0000000 and the mean is -0.5333333611
% omp_max.exe
 The maximum is 135.0000000 and the mean is -0.5333333611
% omp_max.exe
 The maximum is 50.00000000 and the mean is -0.5333333611
```

As suspected, the mean is being calculated correctly but the maximum is not. Worse yet, it is inconsistent from run to run. To track down the cause of this bug we use TotalView® watchpoints to follow how *the_max* is being changed.

After starting Totalview® using

```
% totalview omp_max.exe
```

set a breakpoint just after the entry to the parallel section. Once the program reaches this point and stops, right click on *the_max* and select 'dive'. A new window will open showing the value of *the_max*. Now choose 'Tools' and then 'Create Watchpoint'. Your display should now look similar to the figure below.



The defaults given are fine for what we need so just select 'OK'. This sets the program to watch for changes in the value of *the_max*. Any time it is changed, the program will stop and we can examine the change and which thread made it.

Since we already know that our program's behavior is inconsistent, the results will vary from run to run. A typical run will show the value of *the_max* initially set to '0.0' on all threads. Thread 2 will then change the value to '135',but then other threads will change the value as soon as they reach the line:

```
the_max = tmp_max
```

with the value of the last thread to reach this point being the output of the program. Clearly, the problem is that the threads are working independently of each other and comparing their maximum to the maximum they initially read into *tmp_max*, which will normally be '0.0'. There is no check to see if any other thread has already changed the value of *the_max*.

There are a number of ways to fix this problem, but the easiest method to implement, although one of the least efficient is to use the OpenMP directive OMP CRITICAL around this part of the code, giving the final program:

```
PROGRAM parallel_stats


C  This program has been designed for the purposes of demonstrating common bugs and C  is deliberately
limited and inefficient.

use omp_lib

parameter (N=300)
  integer I
  integer myid,istart,iend,nthreads,nper
  real the_max, b(N),tmp_max, the_mean, tmp
```

```
   OPEN(unit=10,file="b.data")
   OPEN(unit=12,file="found.results")

 DO i=1,300
    READ(10,*) b(i)
 END DO

the_max = 0.0
tmp=0.0
the_mean = 0.0

!$omp parallel private(myid, istart, iend, tmp_max, tmp)

nthreads = OMP_GET_NUM_THREADS()
nper = N/nthreads

myid = OMP_GET_THREAD_NUM()

istart = myid*nper + 1
iend = istart + nper - 1

!$omp critical

      tmp_max = the_max

      DO I=istart,iend
         tmp = tmp+b(I)

           IF(b(I).GT.tmp_max) THEN
             tmp_max=b(I)
             WRITE(12,*)"New maximum",b(I)," found at position",I
           END IF
      END DO

      the_mean = the_mean + tmp

      IF (the_mean.EQ.tmp_max) THEN
        WRITE(12,*)"Data is Uniform"
      END IF

      the_max = tmp_max

!$omp end critical

!$omp end parallel

      the_mean = the_mean / N

    WRITE(12,*)"The maximum is", the_max, "and the mean is",the_mean
    WRITE(*,*)"The maximum is", the_max, "and the mean is",the_mean

END PROGRAM parallel_stats
```

With this change we get a consistent and correct answer:

```
The maximum is 135.0000000 and the mean is -0.5333333611
```

3.6.4 Debugging Exercise

This program uses OpenMP to parallelize the calculation of a simple summation.

```
PROGRAM EXERCISE
        IMPLICIT NONE

        INTEGER, PARAMETER:: XNum=1000, YNum=10
        INTEGER, DIMENSION(XNum):: TOT
        INTEGER I,J

        TOT=0.0
```

```
!$OMP PARALLEL PRIVATE(I,J)

!$OMP DO SCHEDULE(DYNAMIC,4)
        DO I=1,XNum
                DO J=1,YNum
                        TOT(I) = TOT(I) + (I*I*J)
                END DO

! Write out the first 25 values

                IF(I
```

3.6.5 Exercise Solution

The problem in the program is that each thread writes out its results as soon as it reaches the WRITE statement, regardless of whether it is its turn or not. To fix the bug, it is necessary to enforce ordering with the ORDERED directive around the WRITE statement.

```
PROGRAM EXERCISE
        IMPLICIT NONE

        INTEGER, PARAMETER:: XNum=1000, YNum=10
        INTEGER, DIMENSION(XNum):: TOT
        INTEGER I,J

        TOT=0.0

!$OMP PARALLEL PRIVATE(I,J)

!$OMP DO SCHEDULE(DYNAMIC,4) ORDERED
        DO I=1,XNum
                DO J=1,YNum
                        TOT(I) = TOT(I) + (I*I*J)
                END DO

!$OMP ORDERED
                IF(I
```

3.7 MPI Parallel I/O Errors

3.7.1 MPI Parallel I/O Errors

Introduction

You may be surprised that the MPI library supports parallel I/O. Strictly speaking, the parallel I/O routines are part of *MPI-2* but the MPI libraries on most high performance computing systems today not only provide capabilities of MPI-1 but also some of the capabilities of MPI-2. One of the MPI-2 capabilities often provided is parallel I/O. These MPI parallel I/O (MPI-PIO) routines produce "true parallel I/O" meaning that several MPI processes write data to the same disk file simultaneously. Parallel I/O has been attempted not using the MPI-PIO routines but careful analysis of the data flow from the program to disk showed that serial bottlenecks occur.

As is typical of the MPI library overall, the parallel I/O section is extremely thorough and extensive. There are literally hundreds of routines related to MPI-PIO that allow for quite a number of approaches for parallel I/O. In this lesson, our sample program represents the simplest form of MPI-PIO and only the MPI-PIO routines that relate to the bug in the program are described in detail. Therefore, it should not be taken as a template for how all MPI-PIO should be done. It is just one approach and we use it to illustrate how to debug MPI-PIO errors rather than to teach how to do MPI-PIO.

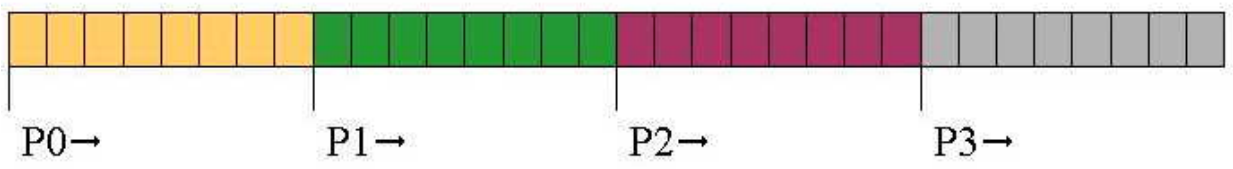
Objectives

In this lesson, you will learn the basics on MPI parallel I/O and how to debug a Fortran 90 program with parallel I/O errors using the Totalview® debugger. A sample code is given that you can download and debug as you follow the lesson. A programming exercise is also provided to help you further understand how to debug this type of error.

3.7.2 Sample Program with an MPI-PIO Error

Before writing an MPI-PIO program you should know what data will be written out, where in the data file the data should be placed, and who (i.e., which MPI process) should write what part of the data. For the sample program we consider the output file to be one long stream of bytes. Each MPI process writes out eight different integers to the file in order of rank.

That is, the rank 0 process starts at the beginning of the file and writes out its eight integers consecutively, then the rank 1 process writes out its eight integers beginning where rank 0 left off, and so on. The following graphic depicts how the output file should look if the program runs correctly:



Each colored area of the output file represents where a certain process will write out its eight integers. In MPI-PIO this area is called the process' view of the file.

Our sample program, nogood3.f90, is a straightforward attempt to have each of four processes write their eight integers into a file (in parallel). The data should be written to the file as shown in the process view figure above. The following is our sample Fortran 90 code:

```
program nogood3
  USE MPI
  integer, dimension(8) :: value=(/10,11,12,13,14,15,16,17/)
  integer :: amode,OUT,etype,filetype,info=0
  integer :: rank,go,err
  integer (KIND=MPI_OFFSET_KIND) :: disp
  integer :: state(MPI_STATUS_SIZE)
  call MPI_INIT(err)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
  amode=ior(MPI_MODE_CREATE,MPI_MODE_WRONLY)
  call MPI_FILE_OPEN(MPI_COMM_WORLD,'testout.dat',amode, &
                info,OUT,err)
  disp=8*rank
  etype=MPI_INTEGER
  filetype=MPI_INTEGER
  call MPI_FILE_SET_VIEW(OUT,disp,etype,filetype, &
            'NATIVE', info,err)
  do i=1,8
    go=(rank+1)*value(i)
    call MPI_FILE_WRITE(OUT,go,1,MPI_INTEGER,state,err)
  end do
  call MPI_FILE_CLOSE(OUT,err)
  call MPI_FINALIZE(err)
end program nogood3
```

Below is a list of the standard MPI-PIO routines used in our program:

1. MPI_FILE_OPEN() connects the actual name of the disk file, 'testout.dat', with the internal name, 'OUT', that is used in the code.
2. MPI_FILE_SET_VIEW() defines the view of the OUT file for each process. There are two critical pieces of information that are passed into the view definition. One is the *etype* argument that indicates the data type that is written out to the file. In our sample program, all data is type MPI_INTEGER. The second is the *disp* argument. Each process uses this argument to tell MPI where its view begins by specifying the origin as a displacement from the beginning of the file.
3. MPI_FILE_WRITE() is the standard file write command. With one call to MPI_FILE_WRITE each MPI process writes one of its integers into its view of OUT. Each process has an individual (private) file pointer that is automatically incremented after every data transfer.
4. MPI_FILE_CLOSE() is called at the end of the file data transfer. It breaks the "connection" between the variable *OUT* and the actual disk file 'testout.dat' that was established with the MPI_FILE_OPEN() routine.

Compiling and Running the Sample Code

We compile and run our sample program on a Cray X1 supercomputer by entering the following commands: (Note that your results may be different if you are using a different system.)

```
Cray-X1$ ftn -O ssp nogood3.f90
Cray-X1$ aprun -n4 ./a.out
```

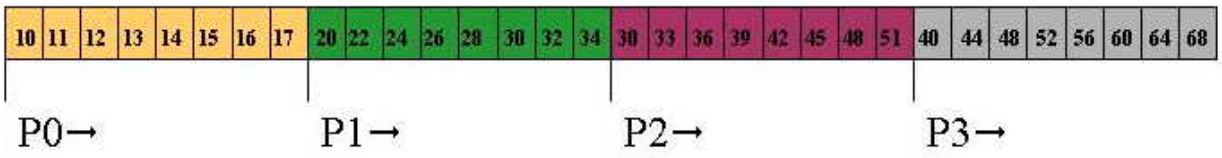
The output is sent to a file called 'testout.dat' and is written in the native binary of the machine used. Having binary I/O as the default for all the MPI-PIO routines is an excellent design feature since binary I/O can often be a hundred times faster, or more, than ASCII I/O. On the other hand, you must have a way to read the binary file to see if the correct integer values were printed out. A handy utility command built into UNIX, called 'od', generates an ASCII version of a binary file where 'od' stands for 'octal dump'. This command has an extremely useful set of options for tailoring the format of the ASCII view.

The Expected Output

Assuming access to the binary file 'testout.dat', what output should the program produce? Each of the four processors will execute the

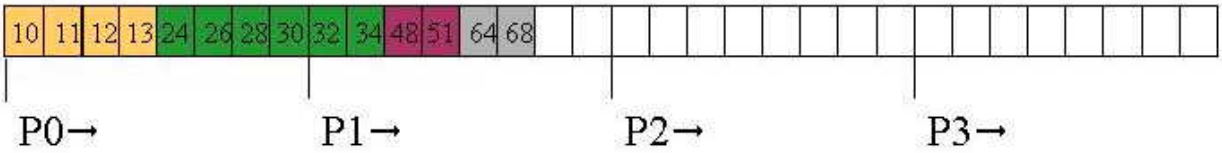
```
do ... end do
```

near the end of the code. In each iteration of this loop the value of the integer variable *go* is written out to the file. As you can see in the assignment statement for *go*, its value is the processor's rank (plus 1) times an element of the array *value*. The entire *value* array has been initialized at declaration. So, in this manner, processor 0 prints out the *value* array as it is, processor 1 prints out two times each element of the *value* array, and so on. The graphical representation of the expected output should look as follows:



The Actual Output

We use octal dump to view the actual output contained in 'testout.dat', which is depicted graphically as:



Looking at the output, you can see that something is wrong with the program. All the data were not written to the file; each processor did not even output all eight of its integers; and only processor 0 began writing its output at the proper displacement for the file beginning.

3.7.3 Debugging the Sample Program

The incorrect output generated by the sample program raises some obvious questions. First, we ask if the processors even had the correct data to write out. Second, were the displacements set correctly for each processor? We use the Totalview® parallel debugger to answer these questions. Specifically, we use the command line interface for Totalview® on the Cray X1. First, the parallel program begins running and second, Totalview is attached to the running code. The following two commands show the recompilation necessary for debugging and the start of the code execution:

```
armstrong$ ftn -g -o nogood3 -O ssp nogood3.f90
armstrong$ aprun -n4 ./nogood3
```

The debugging session takes place in another window in the same directory. Just to double check, the 'ps' command is used to confirm that four MPI processes are actually running:

```
armstrong$ ps -u dje
        PID TTY      TIME CMD
     174926 pts/24  0:01 sh
     174985 pts/25  0:01 sh
     176274 pts/22  5:20 nogood3
     176903 pts/22  0:00 sh
     176904 pts/26  0:00 ps.2421
     176910 pts/22  5:20 nogood3
     176913 pts/22  5:20 nogood3
```

```
     176923 pts/22  5:20 nogood3
     176936 pts/26  0:00 sh
```

The name of the running program is 'nogood3' and there are indeed four processes with that name.

Attaching Totalview® to the Executing Program

To attach the debugger to 'nogood3' a *Process Identification Number* (PID) is needed. We can get the PID from the output table of the 'apstat' command:

```
armstrong$ apstat
   Id     UID    PEs  Depth    Flags    W-PID  W-Signl Modules
======= ======= ===== ===== ========== ======= ======= =======
 176913    3348     4     1 0x1e60901859     222      16       1
```

Using the PID in the first column, we start Totalview® and attach it to the parallel program:

```
armstrong$ totalviewcli -e "dattach nogood3 176913"
Cray X1 TotalView 6.3.1.0 based on Etnus 6.3.1-1
Copyright 1999-2003 by Etnus, LLC. ALL RIGHTS RESERVED.
Copyright 1999 by Etnus, Inc.
Copyright 1996-1998 by Dolphin Interconnect Solutions, Inc.
Copyright 1989-1996 by BBN Inc.
Reading symbols for process 1, executing "nogood3"
Library nogood3, with 2 asects, was linked at 0x01000000, and initially loaded at 0xff00000040000000
Mapping 201956 bytes of ELF string data from 'nogood3'...done
Indexing 134912 bytes of DWARF '.debug_frame' symbols from 'nogood3'...done
Skimming 1065619 bytes of DWARF '.debug_info' symbols from 'nogood3'...done
Attached to process 1 (176913), named "nogood3"
Attached to forked process 2 (176274), with parent 1
Reading symbols for process 2, executing "nogood3"
Attached to related process 2 (176274)
Attached to forked process 3 (176910), with parent 1
Reading symbols for process 3, executing "nogood3"
Attached to related process 3 (176910)
Attached to forked process 4 (176923), with parent 1
Reading symbols for process 4, executing "nogood3"
Attached to related process 4 (176923)
Thread 1.1 has appeared
Thread 1.1 stopped: Stop Signal
Thread 2.1 has appeared
Thread 3.1 has appeared
Thread 4.1 has appeared
d1.
```

We now have the Totalview® prompt and are ready to begin executing debugging commands. As you can see in the Totalview® start-up comments, there is a strange naming convention used for the individual parallel processes. First, each process has a processor number (from 1-4) and it also has a thread number (1-4). So each separate stream of parallel work is identified as a process and a thread. As an example, the Totalview® ID for process 3, which has one thread (itself) is 3.1.

Debugging Commands

Since Totalview® can debug a program with multiple processes and threads, it needs to know what you want it to focus on. The focus is defined as the process, thread, or group of processes or threads on which the compiler commands will be run. For our program, the focus should be on the group of four processes to which the program is attached. Thus, often the first step when parallel debugging is to set the focus:

```
d1. dfocus g
g1.
```

3.7.4 Corrected Sample Program

Having identified the error in our sample program, we can fix it. Here is the corrected MPI-PIO program that produces the expected results:

```
program gone3
  USE MPI
  integer, dimension(8) :: value=(/10,11,12,13,14,15,16,17/)
```

```
   nteger :: amode,OUT,etype,filetype,intsize,info=0
   nteger :: rank,go,err
  integer (KIND=MPI_OFFSET_KIND) :: disp
  integer :: state(MPI_STATUS_SIZE)
  call MPI_INIT(err)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
  amode=ior(MPI_MODE_CREATE,MPI_MODE_WRONLY)
  call MPI_FILE_OPEN(MPI_COMM_WORLD,'testout.dat',amode, &
                info,OUT,err)
  call MPI_TYPE_EXTENT(MPI_INTEGER,intsize,err)
  disp=8*rank*intsize
  etype=MPI_INTEGER
  filetype=MPI_INTEGER
  call MPI_FILE_SET_VIEW(OUT,disp,etype,filetype, &
            'NATIVE', info,err)
  do i=1,8
    go=(rank+1)*value(i)
    call MPI_FILE_WRITE(OUT,go,1,MPI_INTEGER,state,err)
  end do
  call MPI_FILE_CLOSE(OUT,err)
  call MPI_FINALIZE(err)
end program gone3
```

Comparing this program with the incorrect version the following three changes were made:

1. An new integer *intsize* was declared.
2. The new integer, *intsize*, was given a value by using the MPI utility routine MPI_TYPE_EXTENT. This function fills *intsize* with the number of bytes used for an integer on whatever machine the code is running on. In the debugging, it was assumed that an integer took up 4 bytes, which may or may not be true for a specific machine. This corrected program is therefore more portable.
3. Finally, the value of *intsize* is used as a scaling factor in the critical calculation of *disp*.

3.7.5 Debugging Exercise

Write an MPI-PIO program in which three MPI processes simultaneously write to a disk file called 'tri.dat'. Each processor should write out a different set of ten real values. (You can choose or calculate whatever values you like).

The output should be in rank-order as in the sample program. Processor 0 should start writing at the beginning of the file, Processor 1 should start writing after the end of Processor 0's output, and so on. In addition to each processor writing out ten reals, they should also append 32 bytes of empty disk space as a buffer before the file position at which the next processor will begin its output. NOTE: The buffers are empty disk space, they are **not** filled with zeros.

3.7.6 Exercise Solution

The Fortran 90 program shown below is the solution for the exercise. The main difference in this program is in the calculation of *disp* for each of the MPI processors.

```
program answer
  USE MPI
  real, dimension(10) :: readings
  integer :: amode,SOL,etype,filetype,realsize,info=0
  integer :: rank,go,err,i
  integer (KIND=MPI_OFFSET_KIND) :: disp
  integer :: state(MPI_STATUS_SIZE)
  call MPI_INIT(err)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
  do i=1,10
    readings(i)=2.0*rank + i
  end do
  amode=ior(MPI_MODE_CREATE,MPI_MODE_WRONLY)
  call MPI_FILE_OPEN(MPI_COMM_WORLD,'sol.dat',amode, &
                info,SOL,err)
  call MPI_TYPE_EXTENT(MPI_REAL,realsize,err)

  if (rank == 0) then
    disp=10*rank*realsize
  else if (rank == 1) then
```

```
    disp=10*rank*realsize+32
  else if (rank == 2) then
    disp=10*rank*realsize+64
  end if

  etype=MPI_INTEGER
  filetype=MPI_INTEGER
  call MPI_FILE_SET_VIEW(SOL,disp,etype,filetype, &
            'NATIVE', info,err)
  do i=1,10
    go=(rank+1)*readings(i)
    call MPI_FILE_WRITE(SOL,go,1,MPI_REAL,state,err)
  end do
  call MPI_FILE_CLOSE(SOL,err)
  call MPI_FINALIZE(err)
end program answer
```

After compiling, running and analyzing the following binary file produced:

```
armstrong$ ftn -O ssp sol.f90
armstrong$ aprun -n3 ./a.out
armstrong$ od -Ad -dx -V sol.dat
0000000 00000 00001 00000 00002 00000 00003 00000 00004
          0000  0001  0000  0002  0000  0003  0000  0004
0000016 00000 00005 00000 00006 00000 00007 00000 00008
          0000  0005  0000  0006  0000  0007  0000  0008
0000032 00000 00009 00000 00010 00000 00001 00000 00001
          0000  0009  0000  000a  0000  0001  0000  0001
0000048 00000 00001 00000 00001 00000 00001 00000 00001
          0000  0001  0000  0001  0000  0001  0000  0001
0000064 00000 00001 00000 00001 00000 00006 00000 00008
          0000  0001  0000  0001  0000  0006  0000  0008
0000080 00000 00010 00000 00012 00000 00014 00000 00016
          0000  000a  0000  000c  0000  000e  0000  0010
0000096 00000 00018 00000 00020 00000 00022 00000 00024
          0000  0012  0000  0014  0000  0016  0000  0018
0000112 00000 00006 00000 00006 00000 00006 00000 00006
          0000  0006  0000  0006  0000  0006  0000  0006
0000128 00000 00006 00000 00006 00000 00006 00000 00006
          0000  0006  0000  0006  0000  0006  0000  0006
0000144 00000 00015 00000 00018 00000 00021 00000 00024
          0000  000f  0000  0012  0000  0015  0000  0018
0000160 00000 00027 00000 00030 00000 00033 00000 00036
          0000  001b  0000  001e  0000  0021  0000  0024
0000176 00000 00039 00000 00042 00000 00015 00000 00015
          0000  0027  0000  002a  0000  000f  0000  000f
0000192 00000 00015 00000 00015 00000 00015 00000 00015
          0000  000f  0000  000f  0000  000f  0000  000f
0000208 00000 00015 00000 00015
          0000  000f  0000  000f
0000216
armstrong$
```

This output is correct but strange. The correct part is that each process prints out its ten real data values with a spacing of 32 bytes placed between "actual" outputs. The strange part is that the 32 byte buffers are filled with the first value written out by each processor! This may be either a general array initialization feature of Fortran 90 or it may be an capability built into the Cray X1's F90 compiler.

3.8 Incorrect Distribution of Local Memory

3.8.1 Incorrect Distribution of Local Memory

Introduction

When a set of computations is to be carried out in parallel, one absolute requirement is that the global data on one process be correctly distributed across those processes on which the parallel computation is to be performed. If this data is not distributed correctly, programming errors result.

In parallel computing, the basic communication protocol involves the transfer of information and data between individual

processes. There are two different types of interprocess communications that are used to transfer data. They are:

1. point-to-point communications
2. collective communications

Point-to-point communication involves the transfer of data between a single **pair** of processes wherein one member of the pair sends data to the second member, which then receives the data from the sending process. In contrast, *collective communication,* involves the transfer of data among a single **group** of multiple processes specified by an MPI-defined object called an intracommunicator.

Objectives

In this lesson you will learn how to debug a parallel program in which there is a coding error that results in global data being distributed across processes incorrectly.

3.8.2 Point-to-Point Communications

Point-to-point communications in MPI involves a call to two complementary MPI library functions: (1) a function that can send data and (2) a function that can receive the data that is being sent. The following are two such complementary functions in C:

```
MPI_Send(&sdata, count, MPI_datatype, target, tag, comm)

MPI_Recv(&rdata, count, MPI_datatype, source, tag, comm., &status)
```

The list of arguments passed to these functions are

- the memory address of the data being sent (sdata) and received (rdata),
- the number of data elements being sent/received,
- the MPI data type of the data,
- the rank of the process sending (source) and receiving (target) the data,
- an integer value, called a message tag, identifying the message,
- the name of the communicator identifying which processes can be contacted during the communication, and
- the memory address or a variable, status, that contains information pertaining to the source, the message tag, and the number of elements that are actually received by the target process.

Point-to-point communication errors can be caused by a number of different types of programming mistakes. Some of these errors can completely disrupt communications between the sending and receiving processes. For example, in the following code fragment the rank 0 process (*proc0*) is to receive an array of four integers from the rank 1 process (*proc1*) after which it will send an array of six integers back to *proc1*. The data that is to be received from *proc1* will be sent to *proc0* immediately upon receipt of the data sent to it by *proc0*.

```
if (rank = = 0)
{
   MPI_Recv(array2, 6, MPI_INT, 1, tag2, MPI_COMM_WORLD, &status);
   MPI_Send(array1, 4, MPI_INT, 1, tag1, MPI_COMM_WORLD);
}
else if (rank = = 1)
{
   MPI_Recv(array1, 4, MPI_INT, 0, tag1, MPI_COMM_WORLD, &status);
   MPI_Send(array2, 6, MPI_INT, 0, tag2, MPI_COMM_WORLD);
}
```

Unfortunately, *proc0* cannot send *array1* to *proc1* until it receives *array2* from *proc1* and *proc1* cannot send to *proc0* until it receives *array1* from *proc0*. That is, neither process can send data until the other process sends data first. Therefore, each process waits for the other process to send data, which results in no data being sent at all. This situation, called *deadlock*, causes a complete disruption in point-to-point communication. Coding errors that result in deadlock occur frequently in parallel programs involving point-to-point communications. However, deadlocks represent a rather easily recognized bug because at some point in the run the program simply appears to hang. A program using point-to-point communication that appears to have stopped doing anything at all is likely to be deadlocked within some point-to-point communication events.

Of course, as you might have guessed, there are some point-to-point communication errors whose effects are much less obvious than a deadlocked communication event. For example, a parallel programming error that allows a run to complete successfully but gives an incorrect answer for a computation can be quite difficult to debug. An example of this type of programming error is one that causes incorrect data to be sent to a receiving process within a point-to-point

communication event. This lesson illustrates the debugging of some of these types of errors using the TotalView® debugger.

3.8.3 Debugging Sample Codes

3.8.3.1 Debugging Sample Codes

In the following two sections, we debug two sample codes.

3.8.3.2 Point-to-Point Communication Error

The example below shows a parallel code written in C, vectorProduct.c, that computes the vector product (cross product) of two three-dimensional vectors

$$c = a \times b$$

where the i^{th} component of the vector c is given by

$$c_i = a_j b_k \cdot a_k b_j = e_{i\,j\,k}\, a_j\, b_k$$

where e_{ijk} is the Levi-Civita or permutation symbol, defined by

$$e_{i\,j\,k} = +1, \text{ if } (i, j, k) \text{ is an even permutation of } (1, 2, 3)$$
$$= -1, \text{ if } (i, j, k) \text{ is an odd permutation of } (1, 2, 3)$$
$$= 0, \text{ otherwise}$$

For example,

$$c_1 = (+1)a_2\, b_3 + (-1)a_3\, b_2 = a_2\, b_3 - a_3\, b_2$$

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

int main( int argc, char *argv[])
{
  int root = 0;
  int ndim;
  int nproc, procId, index, ndx, procIndex, source, target, tag;
  int nelem, nelem_in_gblArray, nelem_in_locArray;
  int *a, *b, *locA, *locB, *c;
  int locC = 0;
  int a_dot_c, b_dot_c;

  MPI_Status status;

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&nproc);

  MPI_Comm_rank(MPI_COMM_WORLD,&procId);

  ndim = 3;
  nelem_in_gblArray = ndim;
  nelem_in_locArray = (ndim - 1);

/* Dynamically allocate memory for global arrays */

  a = malloc(nelem_in_gblArray * sizeof(int));
  b = malloc(nelem_in_gblArray * sizeof(int));
  c = malloc(nelem_in_gblArray * sizeof(int));

/* Dynamically allocate memory for local arrays */

  locA = malloc(nelem_in_locArray * sizeof(int));
  locB = malloc(nelem_in_locArray * sizeof(int));

/* Initialize local arrays on each process */
```

```
  for (ndx = 0; ndx < nelem_in_locArray; ndx++)
  {
    locA[ndx] = 0;
    locB[ndx] = 0;
  }

  for (index = 0; index < nelem_in_gblArray; index++)
  {
    c[index] = 0;
  }

/* Reinitialize local arrays on the root (proc 0) process */

  if (procId == root)
  {
    a[0] = 5;
    a[1] = -8;
    a[2] = 2;
    b[0] = -25;
    b[1] = 10;
    b[2] = -4;

  /* Print the vectors 'a' and 'b' on the root process */

    printf("The elements of the global arrays are a = [ ");

    for (index = 0; index < nelem_in_gblArray; index++)
    {
      printf("%d ", a[index]);
    }

    printf("] ,  b = [ ");

    for (index = 0; index < nelem_in_gblArray; index++)
    {
      printf("%d ", b[index]);
    }

    printf("]\n\n");

  }

  nelem = 1;

  for (procIndex = 0; procIndex < nproc; procIndex++)
  {
    source = root;
    target = procIndex;
    tag = target;
    ndx = -1;

    for (index = 0; index < nelem_in_locArray; index++)
    {
      if (index != procIndex)
      {
        ndx++;

        if (procId == source)
        {
          if (target != source)
          {
            MPI_Send(&a[index], nelem, MPI_INT, target, tag, MPI_COMM_WORLD);
            MPI_Send(&b[index], nelem, MPI_INT, target, tag, MPI_COMM_WORLD);
          }

          else
          {
            locA[ndx] = a[index];
            locB[ndx] = b[index];
```

```
          }
        }

        else if (procId == target)
        {
           MPI_Recv(&locB[ndx], nelem, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
           MPI_Recv(&locA[ndx], nelem, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
        }
      }
    }
  }

/* Compute vector component of cross-product on each process */

  for (ndx = 0; ndx < nelem_in_locArray; ndx++)
  {
    locC += pow(-1, procId) * ((pow(-1, ndx) * locA[ndx] * locB[ndx + (int)pow(-1, ndx)]));
  }

/* Send individual components of vector 'c' back to the root process */

  target = root;   /* Specify target as root process */
  tag = procId;

  if (procId != target)
  {
    MPI_Send(&locC, nelem, MPI_INT, target, tag, MPI_COMM_WORLD);
  }

  else if (procId == target)
  {
    for (procIndex = 0; procIndex < nproc; procIndex++)
    {
      source = procIndex;
      tag = source;
      if (source != target)
      {
        MPI_Recv(&c[procIndex], nelem, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
      }
      else
      {
        c[procIndex] = locC;
      }
    }
  }

/* Print components of cross-product on root process */

  if (procId == root)
  {
    printf("The components of the cross-product on the root process are c = [ ");
    for (index = 0; index < nelem_in_gblArray; index++)
    {
      printf("%d  ", c[index]);
    }

    printf("]\n\n");
  }

/* Check to see if vector c is normal to vector a or b */

  if (procId == root)
  {
    a_dot_c = 0;
    b_dot_c = 0;
    for (index = 0; index < nelem_in_gblArray; index++)
    {
      a_dot_c += a[index] * c[index];
      b_dot_c += b[index] * c[index];
```

```
    }

    printf("The dot product of vector a and vector c = %d\n\n", a_dot_c);
    printf("The dot product of vector b and vector c = %d\n\n", b_dot_c);
  }

  MPI_Finalize();

  return 0;
}
```

This program uses point-to-point communication, MPI_SEND() and MPI_RECV(), to distribute different subsets of the components of vectors *a* and *b* among three separate processes. The three individual components of vector *c* are then computed in parallel (the local value of *c* (*locC*) on process *n* is the n^{th} component of the global vector *c*). Each local value of *c* is then sent back to the root (rank 0) process, which then prints out the computed components of vector *c*.

To check the results of the computation the dot products $a \cdot c$ and $b \cdot c$ are computed on the root process. Both dot products must be zero since $c = a \times b$ must be perpendicular to both *a* and *b*.

When we compile and run this program on three processes, we obtain the following results:

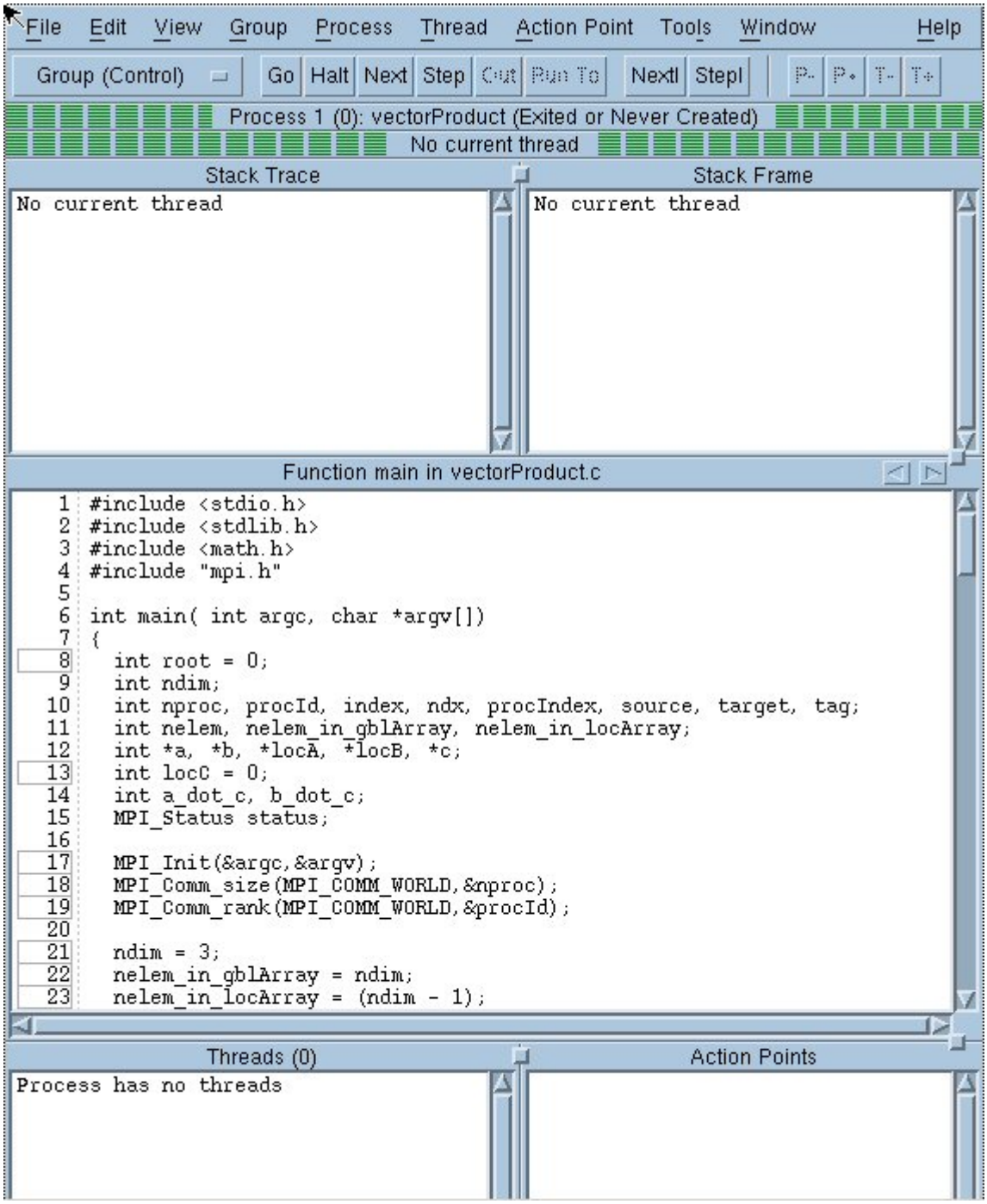
```
$ mpicc -g -lm -o vectorProduct vectorProduct.c

$ mpirun -np 3 vectorProduct

The elements of the global arrays are a = [ 5  -8  2 ] ,  b = [ -25  10  -4 ]

The components of the cross-product on the root process are c = [ 0   0   150  ]

The dot product of vector a and vector c = 300

The dot product of vector b and vector c = -600
```

These results tell us that vector *c* is parallel to the *z*-axis. This would be true only if both vectors *a* and *b* were parallel to the *x,y* plane, which they obviously are not since both of these vectors have a nonzero *z*-component. This is confirmed by the fact that neither $a \cdot c$ nor $b \cdot c$ are zero, which tells us that *c* is not normal to the plane containing *a* and *b*. Clearly our program contains a bug!

To debug our program, we rerun it in TotalView®.

```
$ mpirun -dbg=totalview -np 3 vectorProduct
```

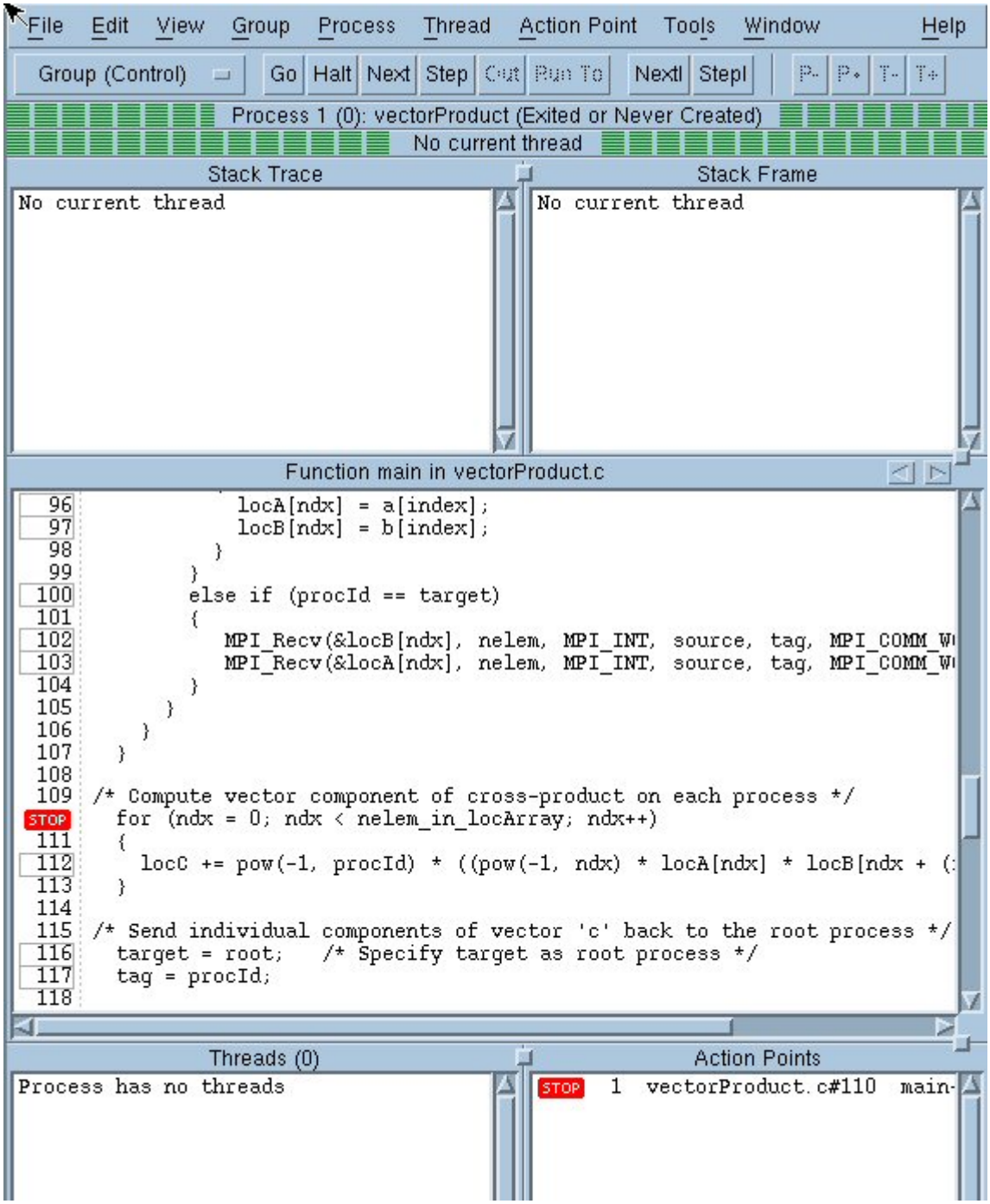
After starting TotalView®, the Process Window for Process 1 (our rank 0 process) appears on the screen as illustrated in the following figure:



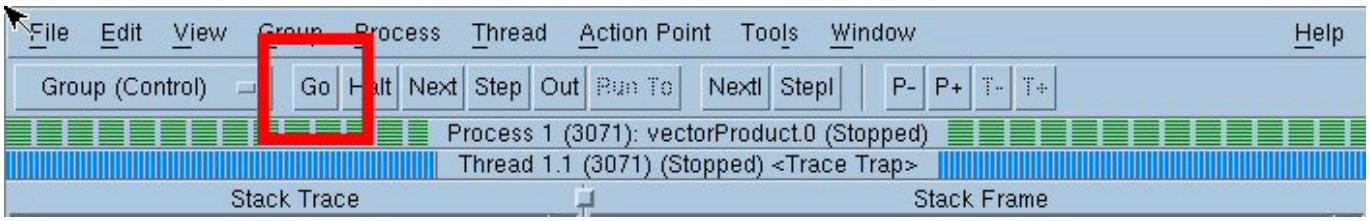
As we can see from the output in the Process Window shown above, vectors *a* and *b* have been initialized correctly on the root process, but vector *c* has not been computed correctly. Therefore, we might guess that one of three possible errors has occurred during this run. They are:

1. The components of *a* and *b* were distributed across the three processes incorrectly (a point-to-point communication error),
2. the data was distributed correctly, but the local value of *c* (*locC*) was computed incorrectly on one or more processes (a computational error), or
3. the values of *locC* were computed correctly on each process, but the values were sent back to the root process incorrectly or were not sent back to the root process at all (a point-to-point communication error).

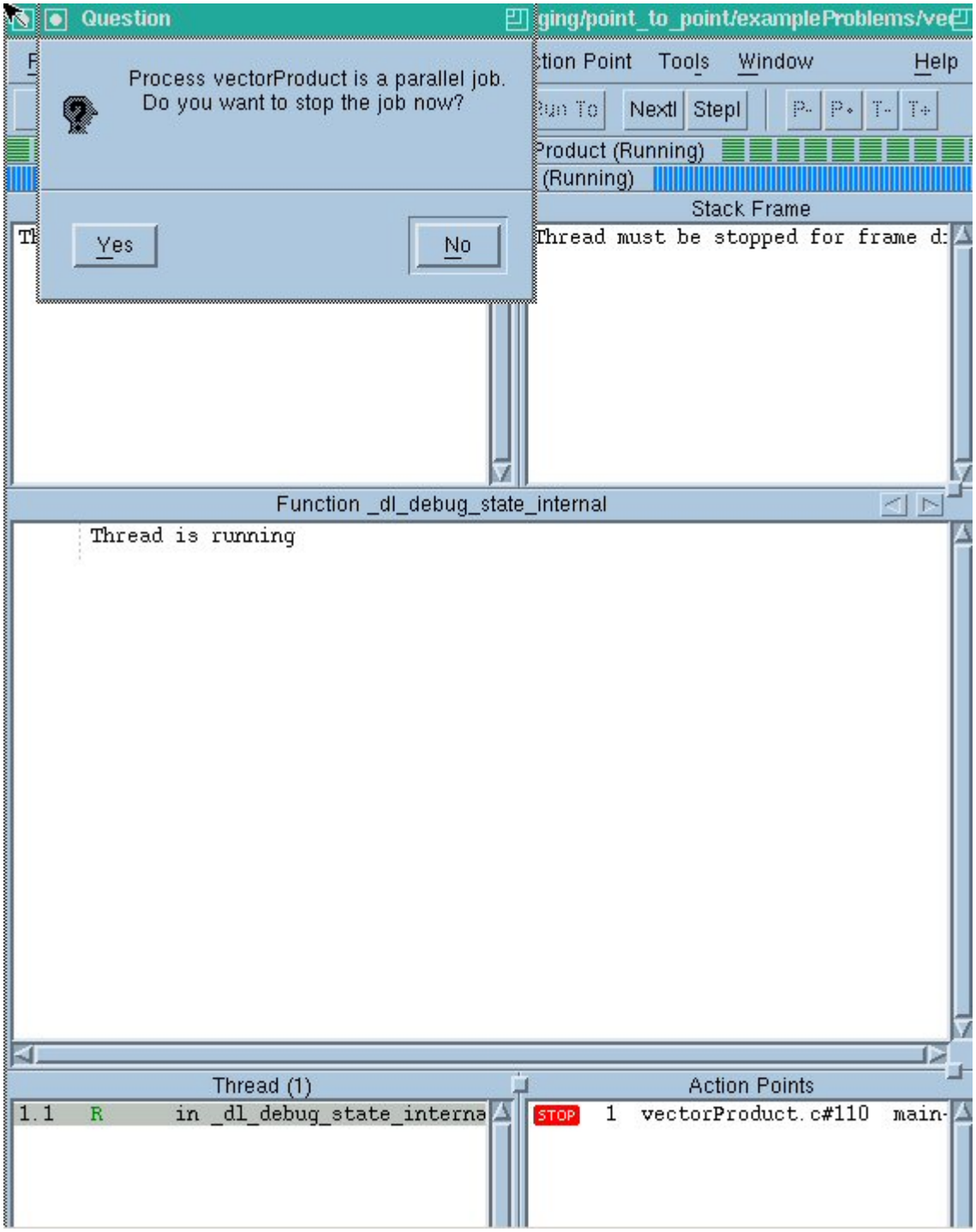
We begin by looking at the values of *locC* on each of our three processes. If we scroll down our code in the Process Window, we see that the computation of *locC* starts at line 110. Therefore, we set our first breakpoint at line 110 by clicking the left mouse button once on number 110 in the left column of the Function pane containing the C code. After setting our breakpoint, we see that the Action Points pane confirms that our breakpoint has been set at line 110 in main().



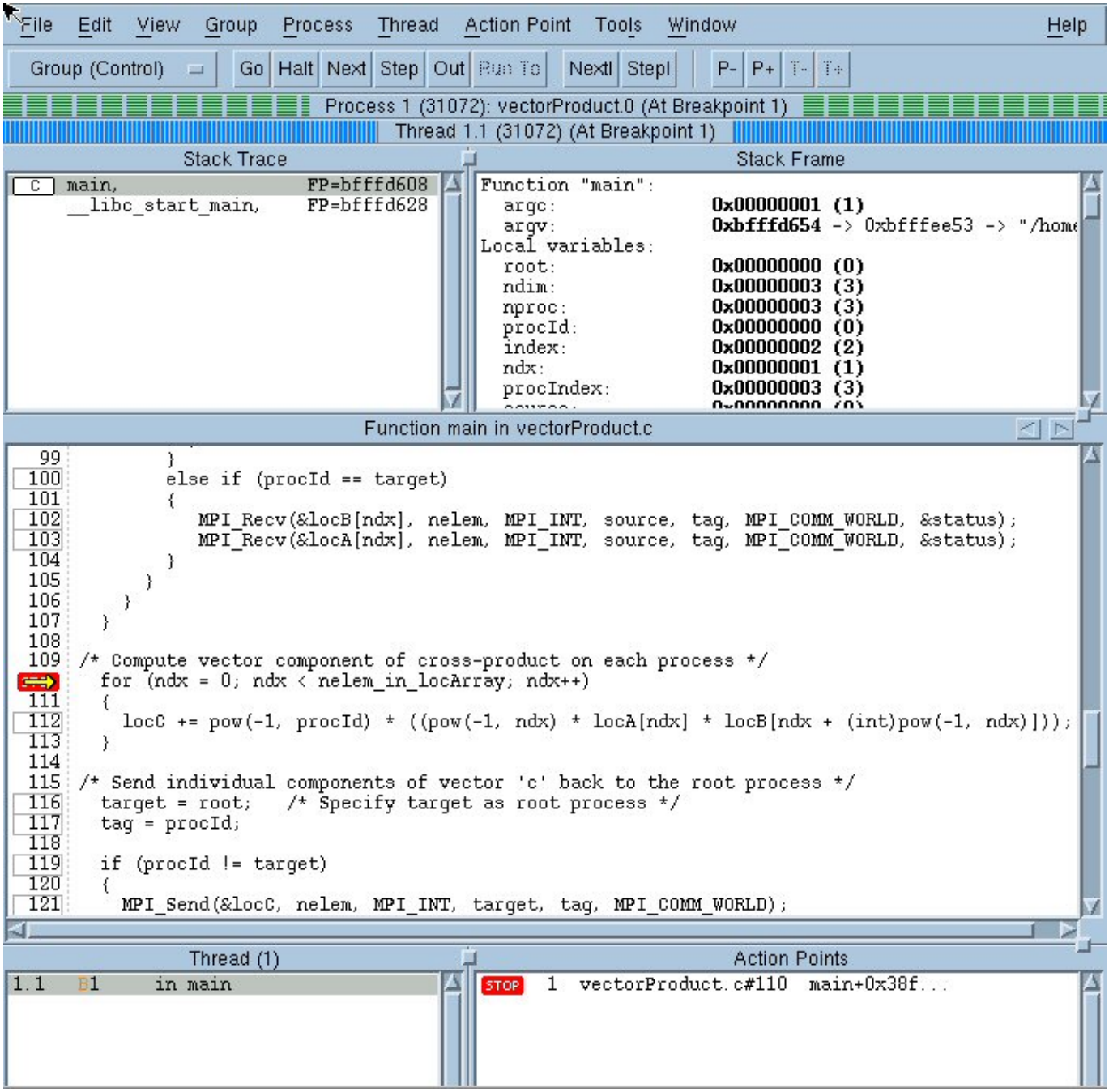
We start our run by clicking on the 'Go' button located in the second row of the Process Window as shown in the illustration below.



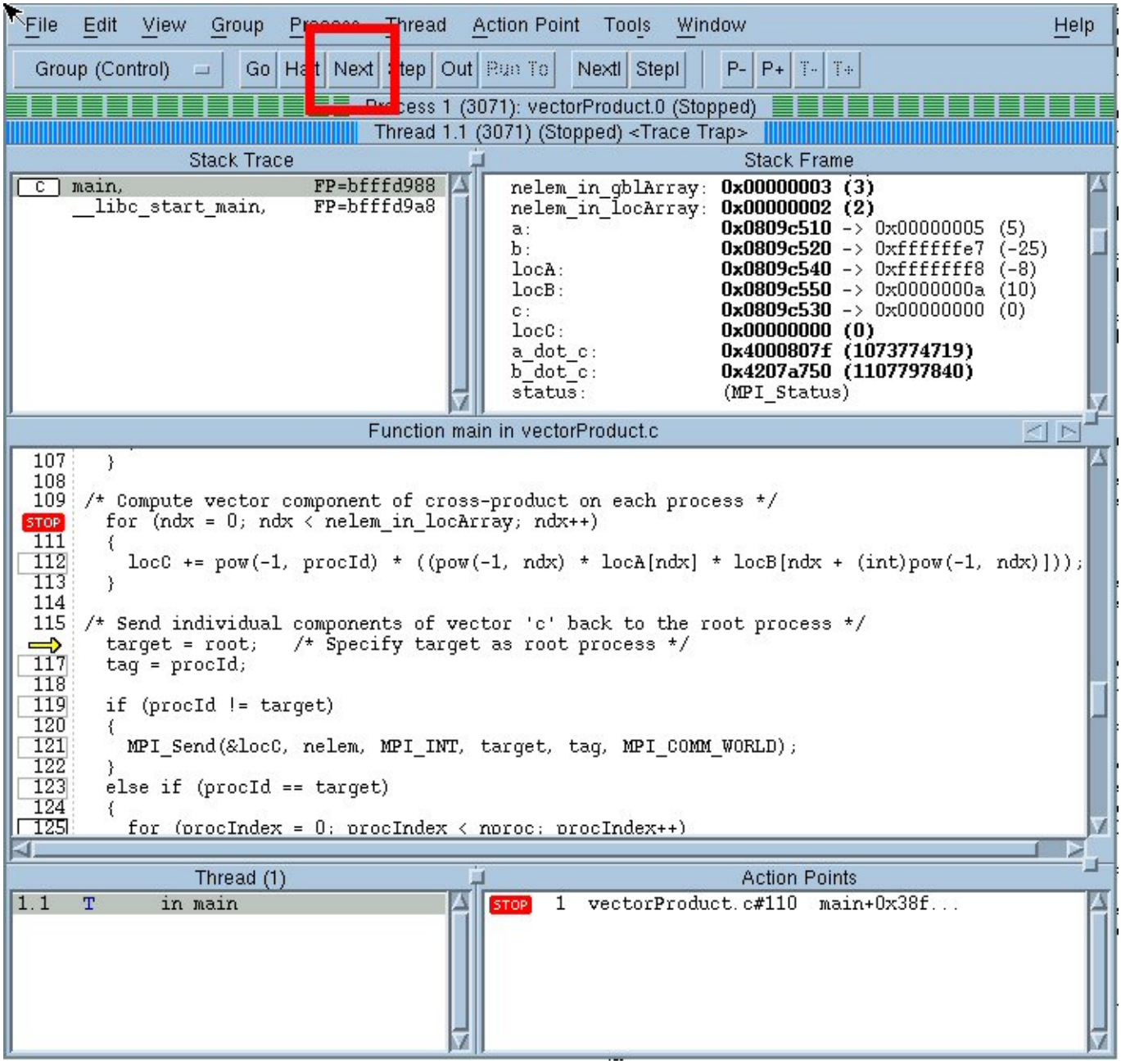
After starting our run, TotalView® displays a popup Question window, shown in the following figure, saying that this run is a parallel job and asking if we want to stop the job.



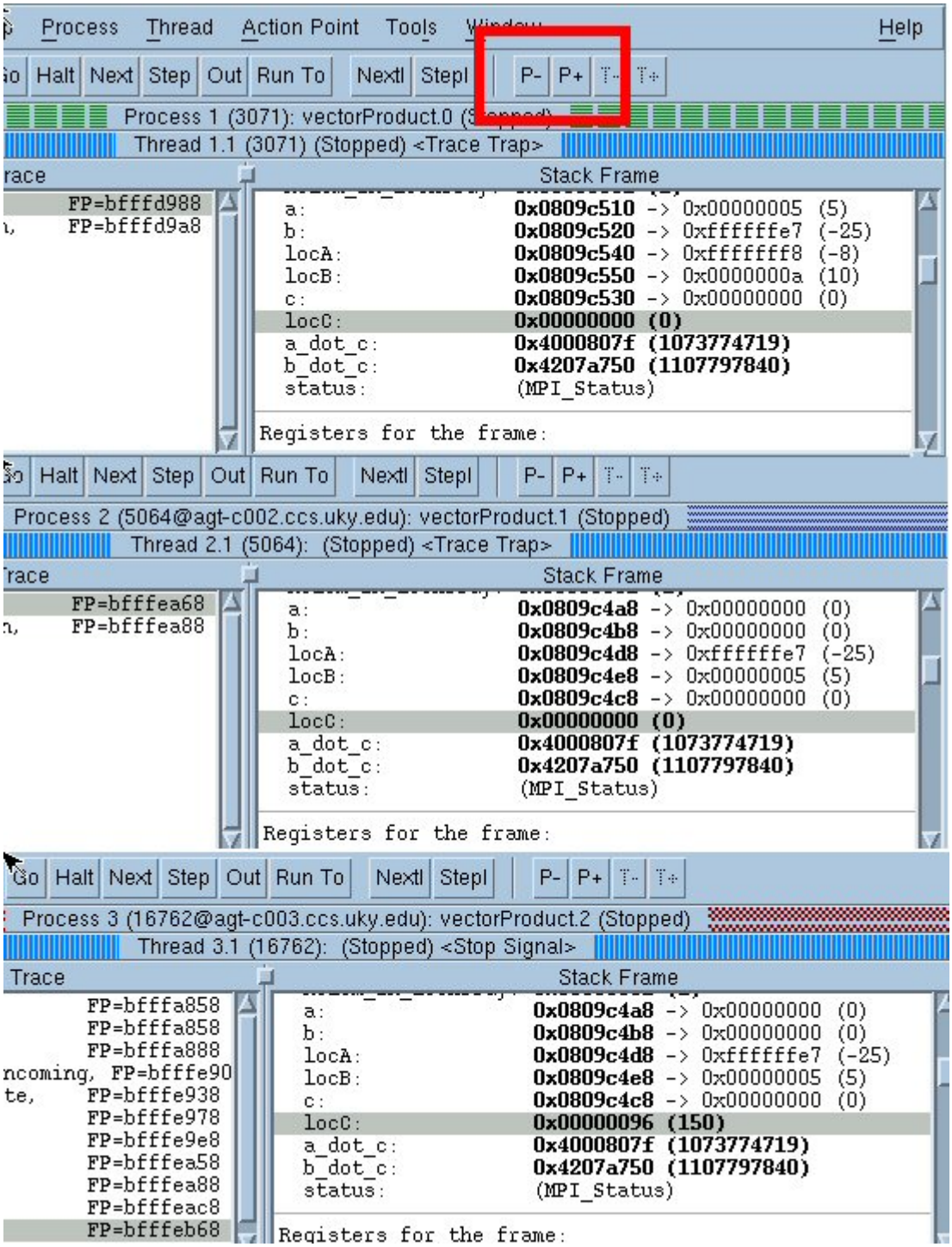
We click the 'No' button to begin our parallel run. As shown in the figure below, our job runs until it reaches the breakpoint we set at line 110 and then halts.



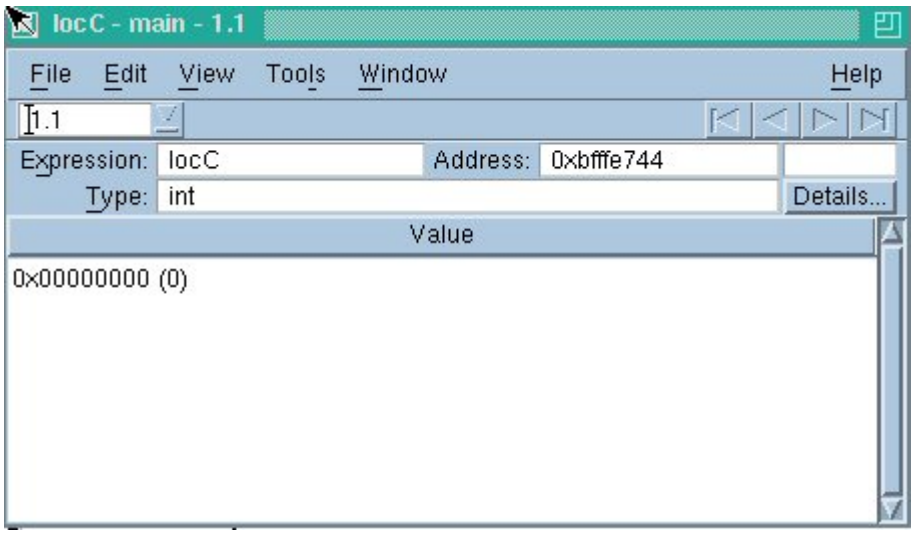
We walk through the *nelem_in_locArray* iterations of this for loop by clicking the 'Next' button (as shown in the following figure) until the yellow arrow identifying the line number moves to line 116.



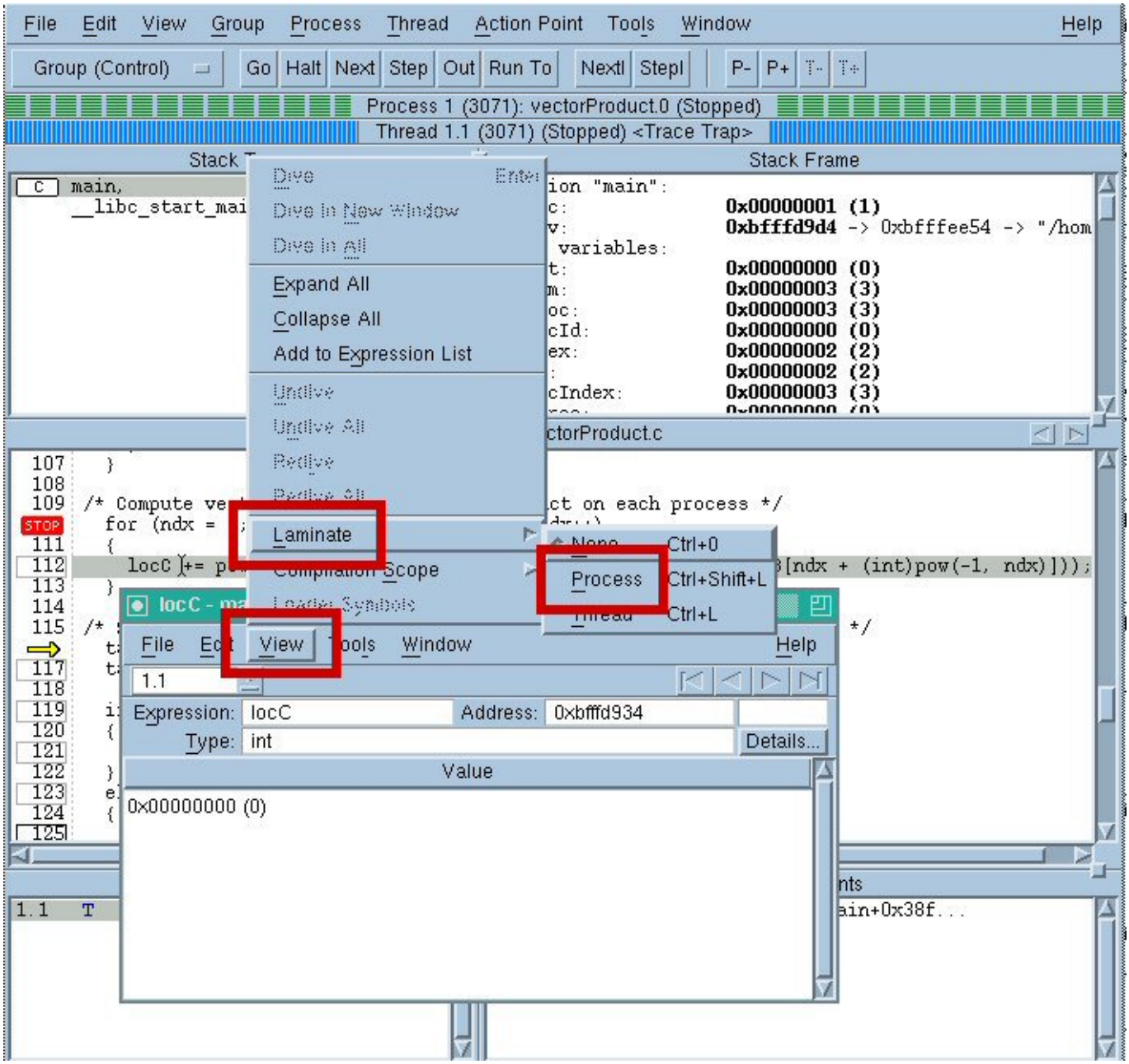
At this point, we can examine the value of *locC* on each process in either of two ways. First, we can scroll down the list of variables in the Stack Frame pane until we come to the variable *locC* whose memory address and numerical value are listed immediately to the right of the variable name. We can then view the value of *locC* on all other processes by clicking on the 'P-' or 'P+' buttons located at the extreme right in the second row of the Process Window as shown in the figure below.



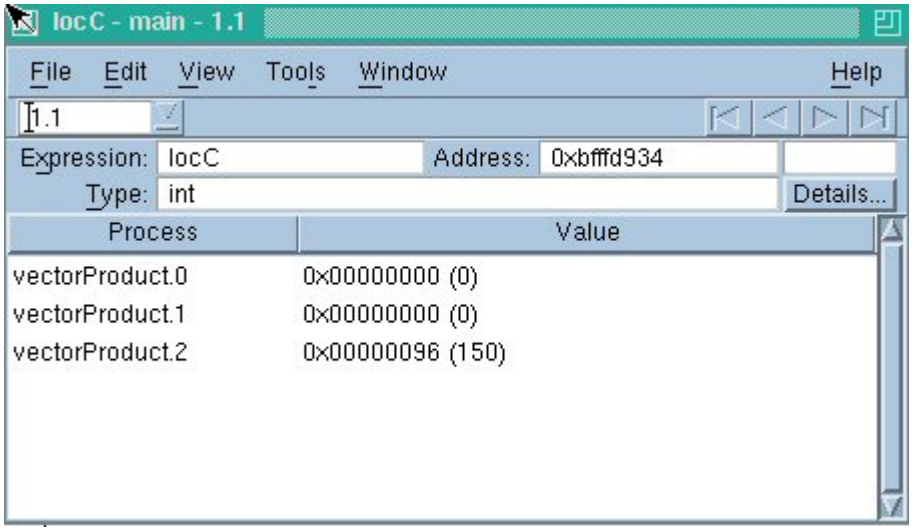
A faster and much easier way to examine the value of *locC* on each process is to double-click on the variable name in line 112 in the Function pane of the Process Window. This brings up a variable window containing the memory address and the value of *locC* on process 1 (our rank 0 process) as illustrated below.



To see the values for *locC* on all other processes in this same variable window, we can click on 'View', then click on 'Laminate', then click on 'Process' as illustrated below.

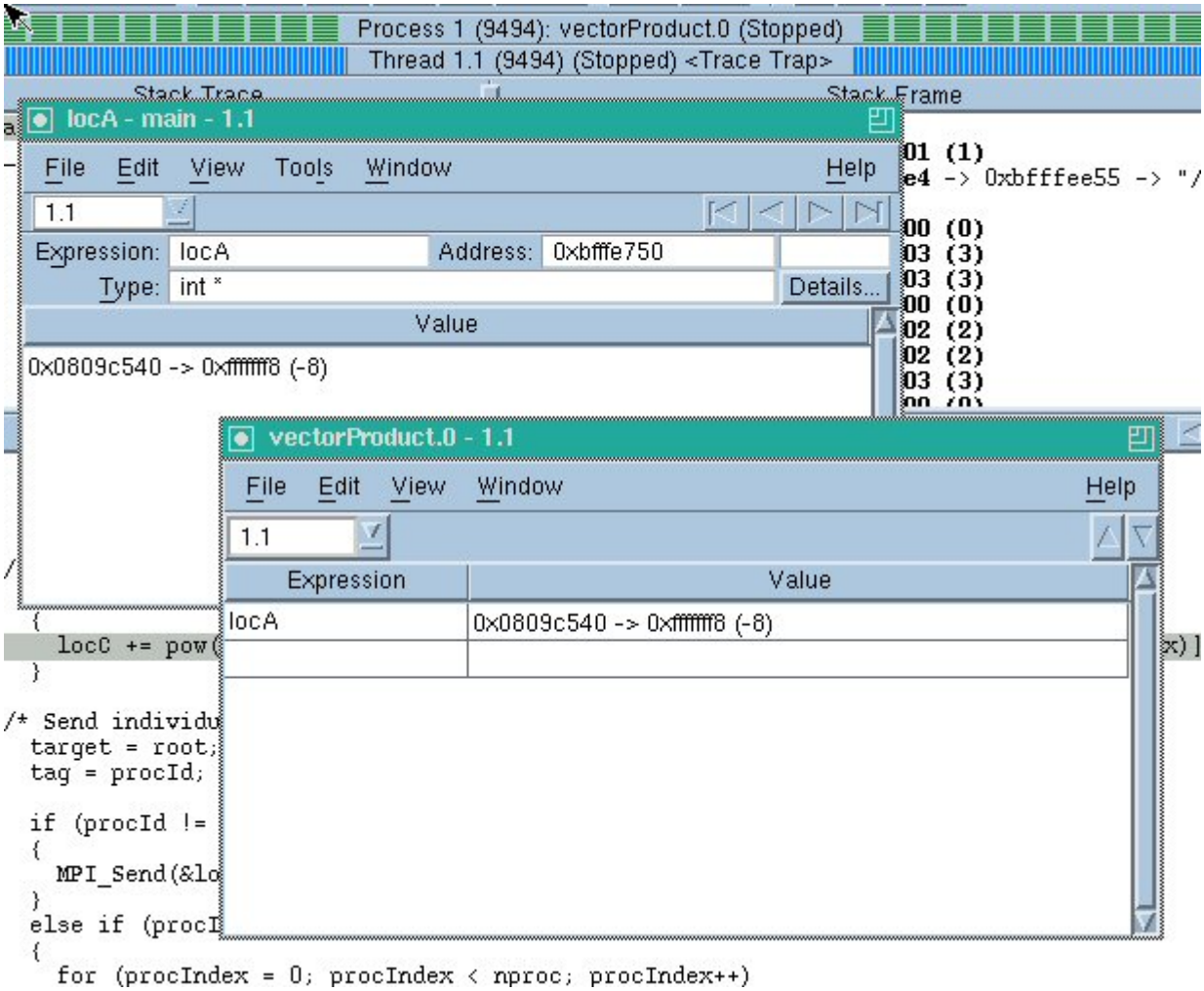


This gives us the variable window shown in the following figure:

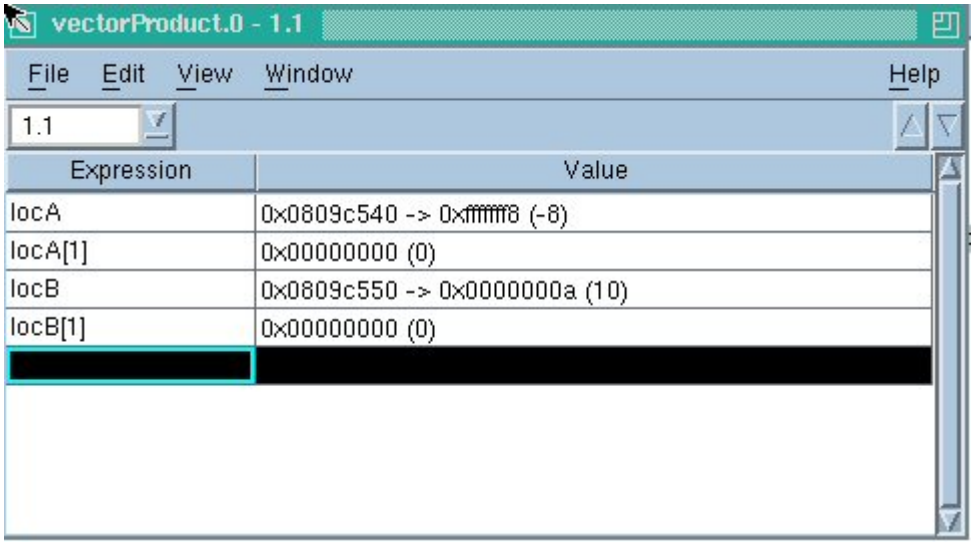


As we can see, by using either of these two methods the values of *locC* on our three processes are identical to the corresponding components of the vector *c* on the root process. This tells us that our error is not in the point-to-point communication used to send the values of *locC* from each process back to the root process on which vector *c* is generated. Therefore, the error must be in the computation of *locC* on each of the individual processes.

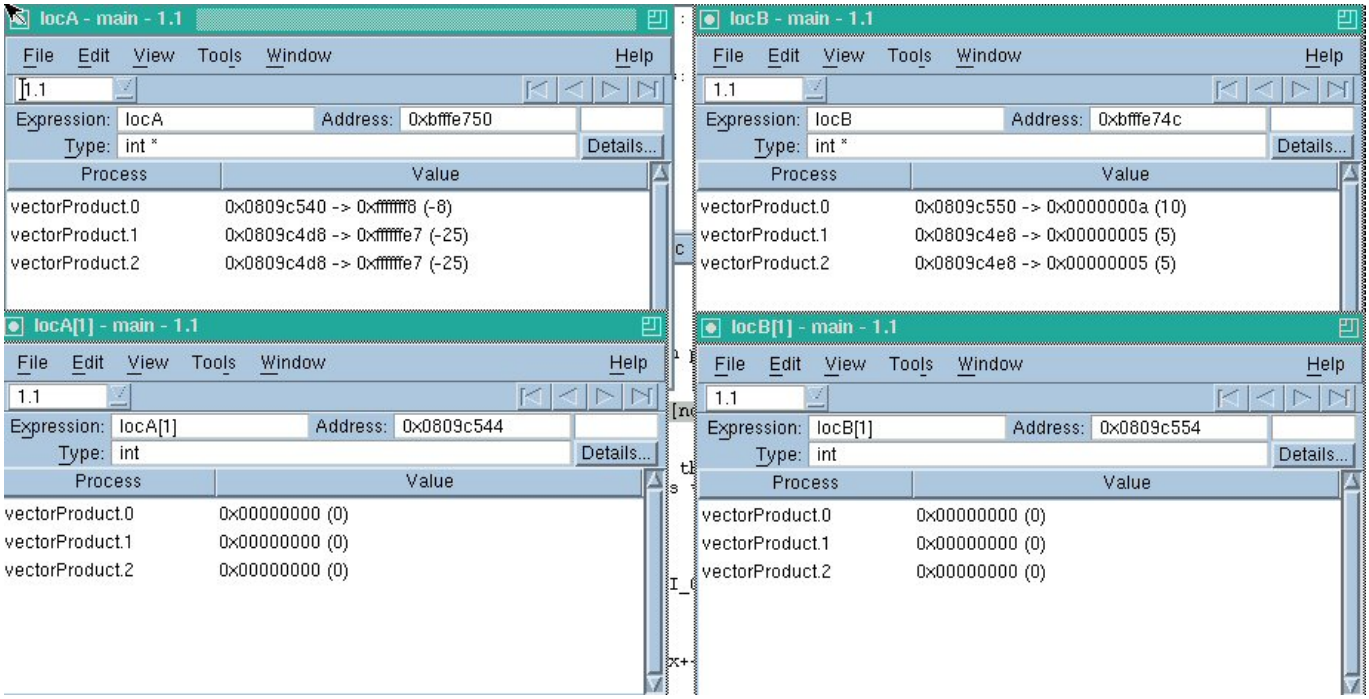
Since the arrays *locA* and *locB* are used to compute *locC* on each process, we take a look at both of these arrays to see if anything looks unusual. We can look at the values of the elements for both arrays on each of our three processes by double-clicking on the variable name *locA* in line 112 in the Function pane of the Process Window, to bring up the variable window for *locA*. Next click on 'View' and then on 'Add to Expression List'. This brings up a second variable window shown in the following figure:



We add the variables *locA[1]*, *locB*, and *locB[1]* to this list:



and then double-click on each of the three variables added to bring up a separate variable window for each one of them. Then we can look at the values for each of these four variables on all three processes in the same way we did for *locC* (Click 'View', 'Laminate', 'Process').



As we can see from the values for these variables, the second element of both local arrays is zero on all three of our processes. Since the global arrays *a* and *b* on the root process from which *locA* and *locB* were generated have all nonzero elements, then we know our error must be related to the way in which the elements of *a* and *b* are being distributed across our three processes. That is, we have an error that is causing the wrong data to be distributed across processes. If we look back through our program we see that the section of code that distributes data across our processes begins on line 73. Before we attempt to identify any errors in this section of code, we make sure that we understand exactly what this code is doing (or, at least, what it is supposed to be doing).

In order to compute

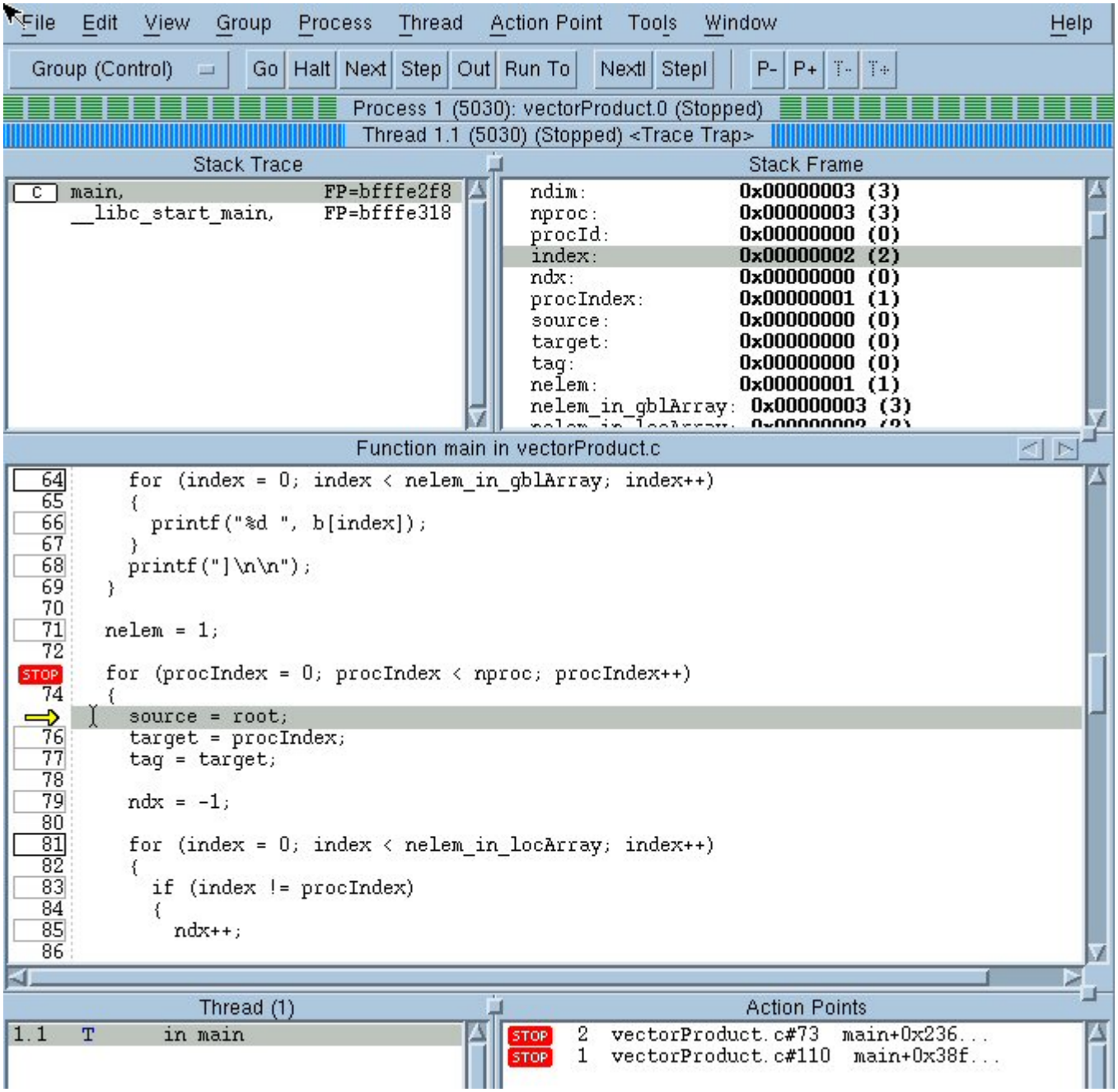
c[i] = a[j] b[k] - a[k] b[j]

a[j], *a[k]*, *b[j]*, and *b[k]* must be stored on process *i* (on the process whose *procIndex* = *i*). Therefore, during each iteration of the for loop on line 73, the three components of vectors *a* and *b* are queried on the root (rank 0) process and the two components from each vector whose index does not equal *procIndex* are selected. If *procIndex = 1 (2)*, the four selected data elements are sent from the root process to the rank 1 (rank 2) process and stored in the local arrays *locA* and *locB*. If instead, *procIndex= 0*, the four elements are stored directly on the root process in the local arrays *locA* and *locB*. It is these local arrays that are then used to compute *locC* on each process.

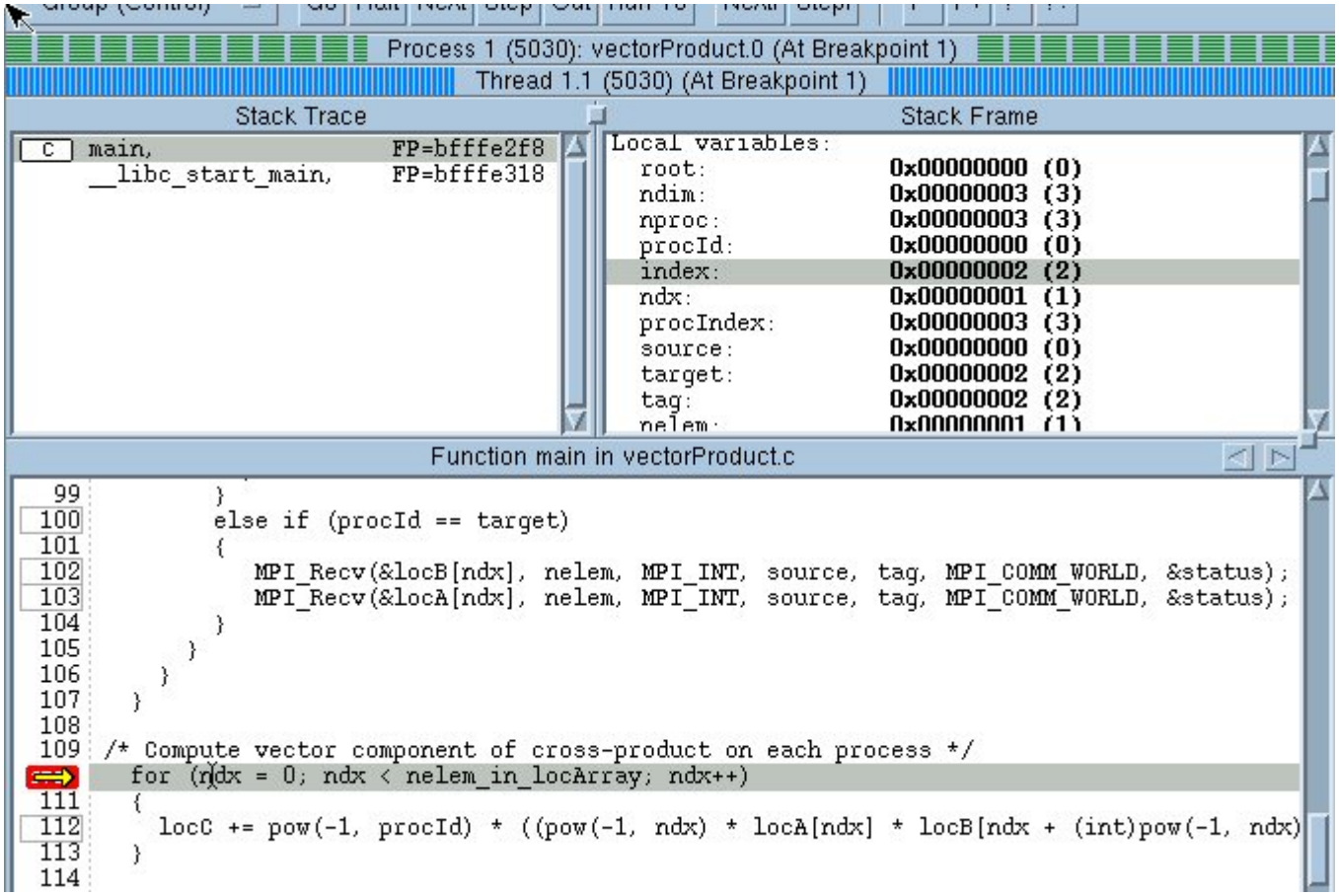
The fact that the elements of the global arrays *a* and *b* are not being distributed across processes correctly suggests that our coding error is causing either one of two possible problems. One possibility is that the pairs of elements from each of the vectors *a* and *b* are not being selected correctly on the root process. This would mean that our error is occurring prior to the point-to-point communication event itself. The second possibility is that the coding error is located within our point-to-point communication code.

In order to identify which of these two alternatives we have, we insert a new breakpoint at line 73 and rerun our code. After our program halts at line 73 we move through this section of code line by line and observe how the values of each of the variables within the for loop change as we iterate through the loop to see if we can spot anything that looks unusual or incorrect. We can watch each of these variables change by looking in the Stack Frame pane of the Process Window. Since the selection of pairs of elements from the global arrays *a* and *b* takes place only on the root process (in the section of code beginning with the second for loop on line 81), then we only need to observe the variables in the Stack Frame for our root process.

In fact, when we do this we find something that looks incorrect. Each time the program exits the second for loop on line 81 and returns to the first for loop on line 73 (*procIndex* increments from 0 to 1), the value of the counter variable index in the second for loop is only 2.



Similarly, when the program exits this entire section of code (procIndex increments from 2 to 3) and moves to line 110, the value of the counter variable index in the second for loop is only 2.



These values tell us that the second for loop is only going through two iterations before the loop is exited. However, since both *a* and *b* are three component arrays, three iterations of our second for loop would be required to query all three components of each global array. But if only two of the three elements of each global array are being queried and one of those fails to satisfy the condition line 83 of our program

```
index ! = procIndex
```

(which is the case when *procIndex = 0, 1*), then only one of the elements from both *a* and *b* is sent to the process whose rank is *procIndex* and the second element (for which *index = procIndex*) is stored on the root process. As a consequence, one component of both *locA* and *locB* is zero on the rank 0 and rank 1 processes resulting in the result that *locC = 0* on each of these processes. These are exactly the results that we obtained in the output from our run.

If we look carefully at the for loop in line 81 of our code, we see exactly what our coding error is. We have set the maximum value of our counter variable *index* to be *nelem_in_locArray*, which is 2 since each local array contains only two elements. What we actually want is for the maximum value to be 3 since there are three elements in each of the global vectors *a* and *b*. Consequently, we should have used *nelem_in_gblArray* in this for loop since *nelem_in_gblArray = 3*. Therefore, to fix our error we need to correct line 81 of our code to read:

```
for (index = 0; index < nelem_in_gblArray; index++)
{
  .   .   .
}
```

When we make this correction, recompile, and rerun our code we obtain the following output:

```
$ mpicc -g -lm -o vectorProduct vectorProduct.c

$ mpirun -np 3 vectorProduct

The elements of the global arrays are a = [ 5 -8 2 ] ,  b = [ -25 10 -4 ]

The components of the cross-product on the root process are c = [ 12  30  150  ]

The dot product of vector a and vector c = 120
```

```
The dot product of vector b and vector c = -600
```

We see from this output that our code change has corrected the original error since all three components of the vector c = a x b are nonzero. Unfortunately, vector c is still pointing in the wrong direction since both $a \cdot c$ and $b \cdot c$ are non-zero (i.e., c is perpendicular to neither a nor b). Obviously, we still have an error somewhere in our program. If c is not normal to the plane containing a and b, then the components of c must be incorrect. But if the components of c are incorrect, there still must be an error in the way that the components are being distributed across processes. That is, we still must be sending the wrong data to one or more of our processes. We leave the task of finding it to you in the end-of-lesson exercise.

3.8.3.3 Collective Communications Call Error

The code shown below, scatter.c, is a C program for computing the elements of the vector B for the equation

A x = B

where A is an n x m matrix, x is an m x 1 column vector, and B is an n x 1 column vector.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

int main( int argc, char *argv[])
{
  int nproc, procId, root;
  int procIndex, index, rowIndex, colIndex;
  int nelem, nelem_in_gblArray, nelem_in_locArray;
  int nterms;

  MPI_Status status;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&procId);

  int **gblA, **locA;
  int *gblx, *gblB, *locx, *locB;

  int nrows_gblA = 4;
  int ncols_gblA = 6;
  int nrows_locA = nrows_gblA;
  int ncols_locA = ncols_gblA/nproc;
  int nelem_gblx = ncols_gblA;
  int nelem_locx = ncols_locA;
  int nelem_gblB = nrows_gblA;
  int nelem_locB = nelem_gblB;

/* Allocate memory for the local vectors 'locx' and 'locB' on all processes */

  locx = malloc(nelem_locx*sizeof(int));
  locB = malloc(nelem_locB*sizeof(int));

/* Define proc 0 as root process */

  root = 0;

  if (procId == root)
  {

  /* Allocate memory for the global vector 'gblB' on the root process */

    gblB = malloc(nelem_gblB*sizeof(int));

  /* Construct a 2D array on the root process */

    gblA = (int**)malloc(nrows_gblA * sizeof(int));

    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
```

```
      gblA[rowIndex] = (int*)malloc(ncols_gblA * sizeof(int));
    }

  /* Initialize the 2D array */

    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      for (colIndex = 0; colIndex < ncols_gblA; colIndex++)
      {
        gblA[rowIndex][colIndex] = rowIndex + colIndex;
      }
    }
  }

/* Print elements of the global array 'gblA' on the root process */

  if (procId == root)
  {
    printf("The elements of the global matrix 'gblA' on the root process are\n");

    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      for (colIndex = 0; colIndex < ncols_gblA; colIndex++)
      {
        printf(" %d  ", gblA[rowIndex][colIndex]);
      }
      printf(" \n");
    }
    printf(" \n");
  }

  if (procId == root)
  {
    gblx = malloc(nelem_gblx*sizeof(int));

    for (index = 0; index < nelem_gblx; index++)
    {
      gblx[index] = pow((index + 1), 2.0);
    }
  }

/* Print elements of global array 'gblx' on root process */

  if (procId == root)
  {
    printf("The elements of 'gblx' on the root process are: [ ");

    for (index = 0; index < nelem_gblx; index++)
    {
      printf("%d ", gblx[index]);
    }

    printf("]\n\n");
  }

/* Construct a local 2D array to receive scattered data */

  locA = (int**)malloc(nrows_locA * sizeof(int));

  for (rowIndex = 0; rowIndex < nrows_locA; rowIndex++)
  {
    locA[rowIndex] = (int*)malloc(ncols_locA * sizeof(int));
  }

/* Initialize the local arrays 'locA', 'locx', and 'locB'  */

  for (rowIndex = 0; rowIndex < nrows_locA; rowIndex++)
  {
    for (colIndex = 0; colIndex < ncols_locA; colIndex++)
```

```
    {
      locA[rowIndex][colIndex] = 0;
    }
  }

  for (index = 0; index < nelem_locx; index++)
  {
    locx[index] = 0;
  }

  for (index = 0; index < nelem_locB; index++)
  {
    locB[index] = 0;
  }

/* Specify number of elements scattered to each separate process */

  nelem = ncols_gblA/nproc;

/* Scatter elements of 'gblA' and 'gblx' across all 'nproc' processes */

  for (index = 0; index < ncols_gblA; index++)
  {
    MPI_Scatter(&(gblA[0][index]), nelem, MPI_INT, &locA[0][index], nelem, MPI_INT, root,
MPI_COMM_WORLD);
  }

  MPI_Scatter(gblx, nelem, MPI_INT, locx, nelem, MPI_INT, root, MPI_COMM_WORLD);

/* Carry out local matrix multiplication on all processes */

  for (rowIndex = 0; rowIndex < nrows_locA; rowIndex++)
  {
    for (colIndex = 0; colIndex < ncols_locA; colIndex++)
    {
      locB[rowIndex] += locA[rowIndex][colIndex] * locx[colIndex];
    }
  }

/* Specify number of terms to be included in the sum of elements of 'locB' on each process */

  nterms = 1;

  for (index = 0; index < nelem_gblB; index++)
  {
    MPI_Reduce(&locB[index], &gblB[index], nterms, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
  }

/* Print elements of the global array B=Ax on the root process */

  if (procId == root)
  {
    printf("The transpose of the global array B=Ax on the root process is: [ ");

    for (index = 0; index < nelem_gblB; index++)
    {
      printf("%d ", gblB[index]);
    }

    printf("]\n\n");
  }

  MPI_Finalize();

  return 0;
}
```

This program is quite similar to the example broadcast.c in the lesson "Debugging Collective Communications". However, in this program we are not distributing rows of the global matrix *gblA* across processes using point-to-point

communications. Instead, we are distributing the columns of *gblA* across processes using a collective communications call to MPI_SCATTER( ). This also requires that we scatter elements of the global vector *gblx* across processes instead of broadcasting the entire global vector to every process.

When we compile and run this code on three processes, we obtain the output shown below.

```
$ mpicc -g -o scatter scatter.c

$ mpirun -np 3 scatter

The elements of the global matrix 'gblA' on the root process are

 0   1   2   3   4   5
 1   2   3   4   5   6
 2   3   4   5   6   7
 3   4   5   6   7   8

The elements of 'gblx' on the root process are: [  1   4   9   16   25   36  ]

The transpose of the global vector B=Ax on the root process is: [  350   649   0   0  ]
```

The fact that the last two elements in the vector *gblB* are zero looks very suspicious. The k^{th} element of *gblB* on the root process is

$$gblB[k] = ak0\ x0 + ak1\ x1 + . . . + akn\ xn$$

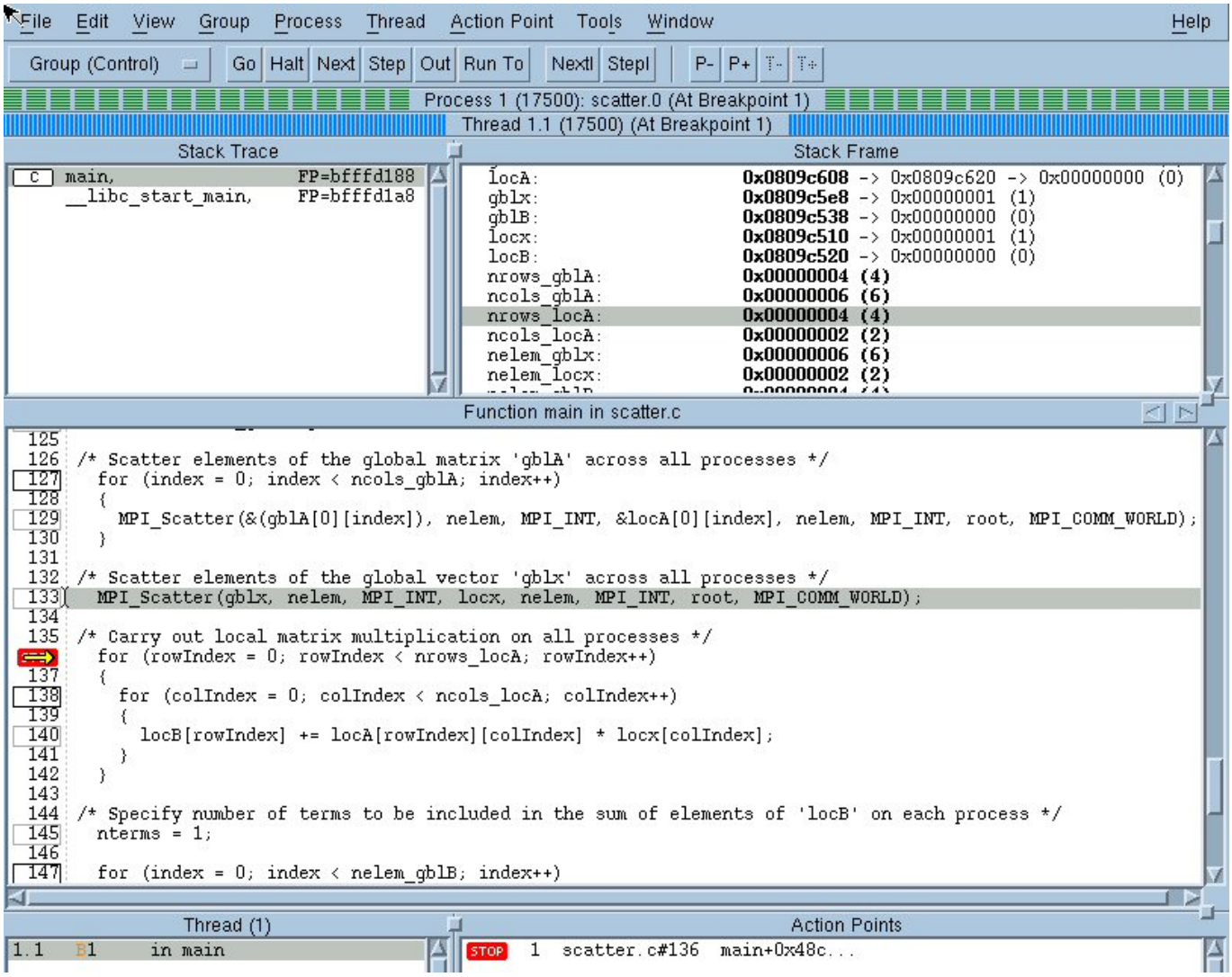
which equals zero for $\{ a\ k\ j \} \cdot 0$ and $\{ x\ j \} \cdot 0$, only if $x0 = x1 = . . . = xn = 0$ or $ak\ 0 = ak\ 1 = . . . = ak\ n = 0$. Since all of the elements in *gblx* are positive then *gblB[k] = 0* only if $\{ a\ k\ j \} \cdot 0$, for all *akj* in the global matrix *gblA*. But there is no row in *gblA* in which all of the elements are zero. Consequently, the fact that *gblB[2]* and *gblB[3]* are both zero tells us that we definitely have a coding error that is causing some or all of the elements of *gblB* to be computed incorrectly.

Specifically, an incorrect valuation of *gblB* could result from any of three types of errors:

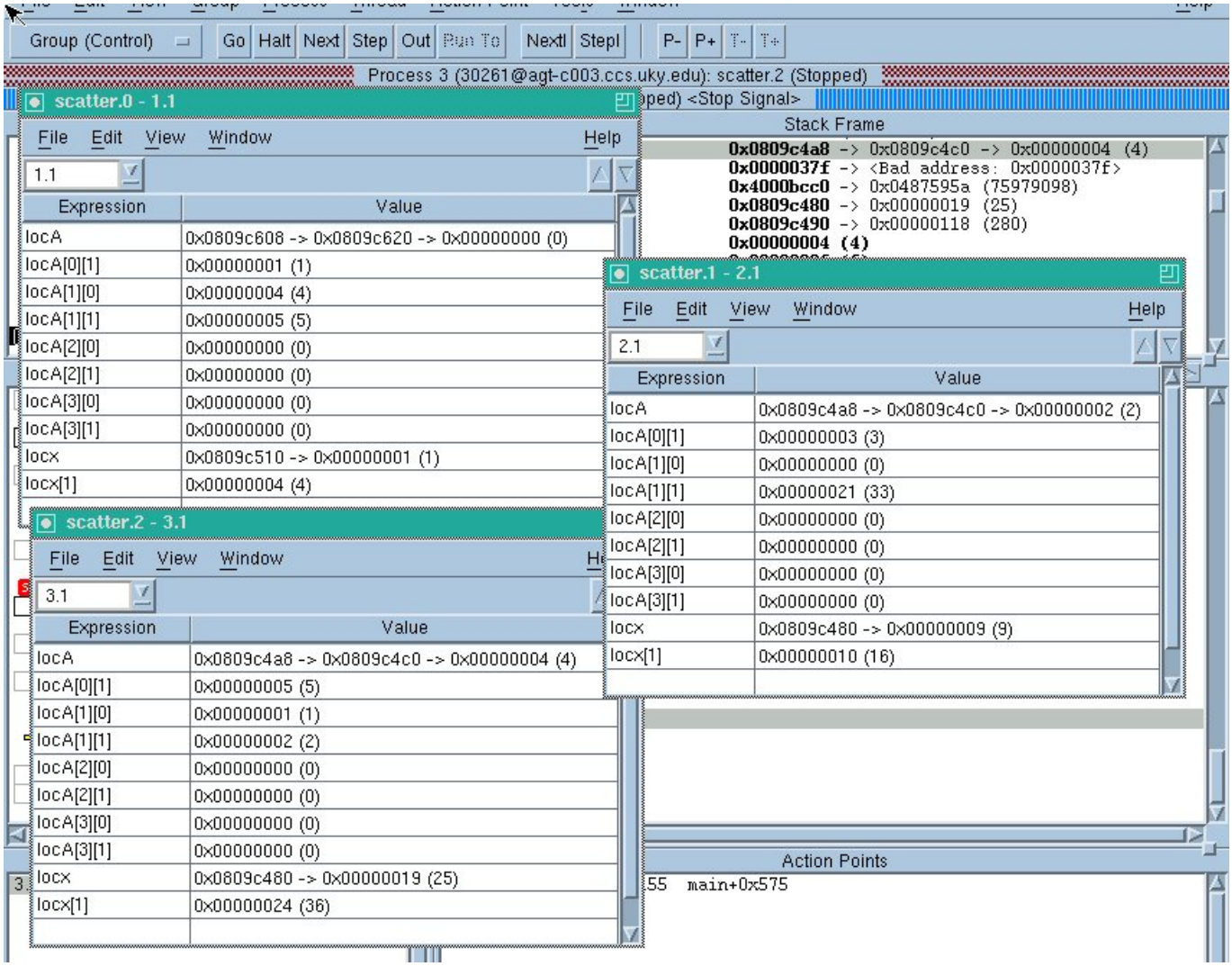
1. the elements of *gblA* and/or *gblx* are being scattered across processes incorrectly,
2. the local computation of *B=Ax* is being carried out incorrectly on one or more processes, or
3. the collective communications function MPI_REDUCE( ) is being called incorrectly.

To debug our code, we run our program in TotalView® and check for each of these three types of errors in our program. To check for errors in the distribution of *gblA* and/or *gblx* across processes we examine the elements of the local arrays *locA* and *locx* (the receiving buffers for the scattered elements of the corresponding global arrays)on each process.

As shown in the figure below, we have set an initial breakpoint on line 136 immediately after the second call to MPI_SCATTER( ) on line 133. In the Stack Frame pane of the Process Window in the same figure we see that *locA* is a 4 x 2 matrix (*nrows_locA = 4*, *ncols_locA = 2*) and *loc*x (actually, the transpose of the column vector *locx*) is a 1 x 2 matrix.



Next we focus on *locA* and *locx* and examine all eight elements of *locA* and both elements of *locx* on all three processes. When we do this, we see exactly what our problem is. As shown in the figure below, the vector *locx* is correct on all three of our processes, but the local matrix *locA* is incorrect on every process. Not only are the majority of the elements of *locA* zero, but the value of *locA[1][1] = 33* on proc 1.



Clearly, this is impossible because *gblA* on proc 0 does not even contain an element whose value is 33. This suggests that elements outside of the boundaries of *gblA* are being distributed across processes. Furthermore, the fact that the last two rows of *gblA* are zero on all processes suggests that the final two rows of *gblA* are not being scattered at all.

Therefore, our coding error must be in the call to MPI_SCATTER( ) on line 129. If we look carefully at the parameters that are being passed to MPI_SCATTER( ),

```
MPI_Scatter(&(gblA[0][index]), nelem, MPI_INT, &locA[0][index], nelem,

                                                MPI_INT, root, MPI_COMM_WORLD); ,
```

we can see exactly where our error is located. During each iteration of the for( ) loop in line 127 we are scattering *nelem = 2* adjacent elements of the first row of beginning with the element in column index, where index is varying from 0 to (ncols_gblA □ 1) = 5. Each pair of scattered elements,

```
{ gblA[0][index],  gblA[0][index + 1] }
```

is then stored in the first row beginning in column index of the local matrix *locA*,

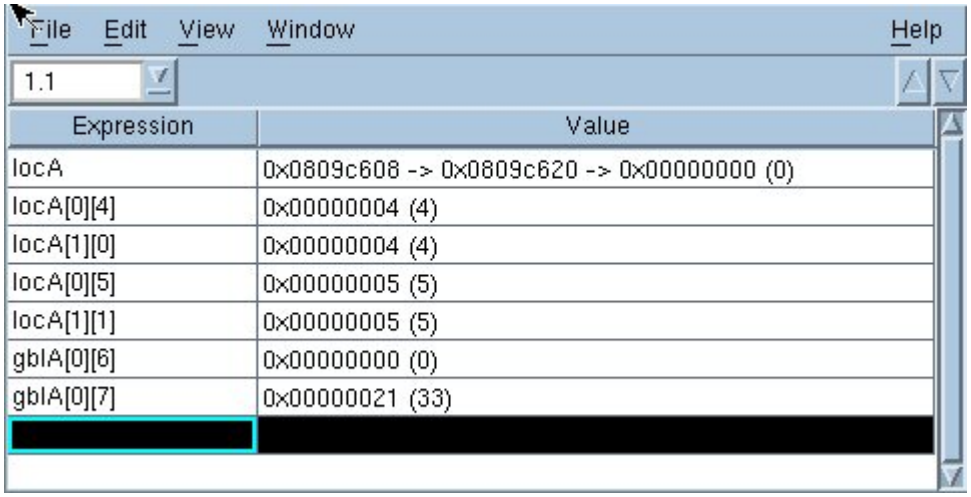
```
{ locA[0][index],  locA[0][index + 1] }
```

Unfortunately, the boundaries of *locA[0][k]* and *gblA[0][k]* are encountered when *k = 1* and *k = 5*, respectively. Consequently, when *index = 1*, the second element of the scattered pair is stored at *locA[0][2]*, which is outside the boundary of the first row of *locA*. And when *index > 1*, both scattered pairs are stored outside the boundary of the first row of *locA*.

If we look at the addresses of the elements being stored outside the boundary of the first row of *locA*,

&locA[0][4] = &locA[1][0] and &locA[0][5] = &locA[1][1]

as shown in the following figure.



>

Furthermore, when *index = 5*, we are scattering data located at *gblA[0][6]* on proc 0, which is outside the boundary of the first row of *gblA*. In fact, this is exactly why the data value 33 shows up as one of the elements of on proc 1,

*(&gblA[0][6]) = 33

as shown in the figure above.

Obviously, we do not want to distribute *gblA* across processes by scattering contiguous pairs of elements from row 1 of *gblA* by stepping through successive columns of *gblA*. Instead, we want to scatter *nproc* pairs of elements (for *nelem = 2*) beginning in column 1 while stepping through successive rows of *gblA*. That is, we want to scatter the elements

```
{ gblA[index][0], gblA[index + 1][0] }
```

while iterating through index from 0 to *(nrows_gblA □ 1) = 3*.

Therefore, we can correct the error in our program by making the following changes in lines 127 and 129 of our code.

```
for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
  {
    MPI_Scatter(&gblA[rowIndex][0], nelem, MPI_INT, &locA[rowIndex][0], nelem,
                                                          MPI_INT, root, MPI_COMM_WORLD);
  }
```

When we recompile and run this modified code on three processes, we get the following correct output:

```
$ mpicc -g -o scatter scatter.c
$ mpirun -np 3 scatter

 0    1    2    3    4    5
 1    2    3    4    5    6
 2    3    4    5    6    7
 3    4    5    6    7    8

The elements of 'gblx' on the root process are: [ 1 4 9 16 25 36 ]

The transpose of the global array B=Ax on the root process is: [ 350 441 532 623 ]
```

3.8.4 Debugging Exercises

Exercise 1:

After correcting one error in the example code vectorProduct.c in this lesson, we discovered that our corrected code still contained at least one additional error. Debug the corrected code, shown below, to find the remaining error(s) and make the additional correction(s) to this code.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

int main( int argc, char *argv[])
{
  int root = 0;
  int ndim;
  int nproc, procId, index, ndx, procIndex, source, target, tag;
  int nelem, nelem_in_gblArray, nelem_in_locArray;
  int *a, *b, *locA, *locB, *c;
  int locC = 0;
  int a_dot_c, b_dot_c;
  MPI_Status status;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&procId);

  ndim = 3;
  nelem_in_gblArray = ndim;
  nelem_in_locArray = (ndim - 1);

/* Dynamically allocate memory for global arrays */
  a = malloc(nelem_in_gblArray * sizeof(int));
  b = malloc(nelem_in_gblArray * sizeof(int));
  c = malloc(nelem_in_gblArray * sizeof(int));

/* Dynamically allocate memory for local arrays */
  locA = malloc(nelem_in_locArray * sizeof(int));
  locB = malloc(nelem_in_locArray * sizeof(int));

/* Initialize local arrays on each process */
  for (ndx = 0; ndx < nelem_in_locArray; ndx++)
  {
    locA[ndx] = 0;
    locB[ndx] = 0;
  }

  for (index = 0; index < nelem_in_gblArray; index++)
  {
    c[index] = 0;
  }

/* Reinitialize local arrays on the root (proc 0) process */
  if (procId == root)
  {
    a[0] = 5;
    a[1] = -8;
    a[2] = 2;
    b[0] = -25;
    b[1] = 10;
    b[2] = -4;

  /* Print the vectors 'a' and 'b' on the root process */
    printf("The elements of the global arrays are a = [ ");
    for (index = 0; index < nelem_in_gblArray; index++)
    {
      printf("%d ", a[index]);
    }
    printf("] ,  b = [ ");

    for (index = 0; index < nelem_in_gblArray; index++)
    {
      printf("%d ", b[index]);
    }
    printf("]\n\n");
  }
```

```
  nelem = 1;

  for (procIndex = 0; procIndex < nproc; procIndex++)
  {
    source = root;
    target = procIndex;
    tag = target;

    ndx = -1;

    for (index = 0; index < nelem_in_gblArray; index++)
    {
      if (index != procIndex)
      {
        ndx++;

        if (procId == source)
        {
          if (target != source)
          {
            MPI_Send(&a[index], nelem, MPI_INT, target, tag, MPI_COMM_WORLD);
            MPI_Send(&b[index], nelem, MPI_INT, target, tag, MPI_COMM_WORLD);
          }
          else
          {
            locA[ndx] = a[index];
            locB[ndx] = b[index];
          }
        }
        else if (procId == target)
        {
           MPI_Recv(&locB[ndx], nelem, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
           MPI_Recv(&locA[ndx], nelem, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
        }
      }
    }
  }

/* Compute vector component of cross-product on each process */
  for (ndx = 0; ndx < nelem_in_locArray; ndx++)
  {
    locC += pow(-1, procId) * ((pow(-1, ndx) * locA[ndx] * locB[ndx + (int)pow(-1, ndx)]));
  }

/* Send individual components of vector 'c' back to the root process */
  target = root;   /* Specify target as root process */
  tag = procId;

  if (procId != target)
  {
    MPI_Send(&locC, nelem, MPI_INT, target, tag, MPI_COMM_WORLD);
  }
  else if (procId == target)
  {
    for (procIndex = 0; procIndex < nproc; procIndex++)
    {
      source = procIndex;
      tag = source;

      if (source != target)
      {
        MPI_Recv(&c[procIndex], nelem, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
      }
      else
      {
        c[procIndex] = locC;
      }
    }
  }
```

```
/* Print components of cross-product on root process */
  if (procId == root)
  {
    printf("The components of the cross-product on the root process are c = [ ");
    for (index = 0; index < nelem_in_gblArray; index++)
    {
      printf("%d  ", c[index]);
    }
    printf("]\n\n");
  }

/* Check to see if vector c is normal to vector a or b */
  if (procId == root)
  {
    a_dot_c = 0;
    b_dot_c = 0;

    for (index = 0; index < nelem_in_gblArray; index++)
    {
      a_dot_c += a[index] * c[index];
      b_dot_c += b[index] * c[index];
    }

    printf("The dot product of vector a and vector c = %d\n\n", a_dot_c);
    printf("The dot product of vector b and vector c = %d\n\n", b_dot_c);
  }

  MPI_Finalize();

  return 0;
}
```

As was previously shown at the end of the example problem in this lesson, this code generates the following incorrect output:

```
$ mpirun -np 3 vectorProduct

The elements of the global arrays are a = [ 5 -8 2 ] ,  b = [ -25 10 -4 ]

The components of the cross-product on the root process are c = [ 12  30  150  ]

The dot product of vector a and vector c = 120

The dot product of vector b and vector c = -600
```

Exercise 2:

The code shown below, scatter.c, is a C program for computing the elements of the vector *B* for the equation

$$A\ x\ =\ B$$

where *A* is an *n x m* matrix, *x* is an *m x 1* column vector, and *B* is an *n x 1* column vector.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

int main( int argc, char *argv[])
{
  int nproc, procId, root;
  int procIndex, index, rowIndex, colIndex;
  int nelem, nelem_in_gblArray, nelem_in_locArray;
  MPI_Status status;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
```

```
  MPI_Comm_rank(MPI_COMM_WORLD,&procId);

  int **gblA, **locA;
  int *gblx, *gblB, *locx, *locB;

  int nrows_gblA = 4;
  int ncols_gblA = 6;
  int nterms;

  int nrows_locA = nrows_gblA;
  int ncols_locA = ncols_gblA/nproc;
  int nelem_gblx = ncols_gblA;
  int nelem_locx = ncols_locA;
  int nelem_gblB = nrows_gblA;
  int nelem_locB = nelem_gblB;

  gblB = malloc(nelem_gblB*sizeof(int));
  locB = malloc(nelem_locB*sizeof(int));
  locx = malloc(nelem_locx*sizeof(int));

/* Define proc 0 as root process */
  root = 0;

  if (procId == root)
  {
  /* Construct a 2D array on the root process */
    gblA = (int**)malloc(nrows_gblA * sizeof(int));

    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      gblA[rowIndex] = (int*)malloc(ncols_gblA * sizeof(int));
    }

  /* Initialize the 2D array */
    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      for (colIndex = 0; colIndex < ncols_gblA; colIndex++)
      {
        gblA[rowIndex][colIndex] = rowIndex + colIndex;
      }
    }
  }

/* Print elements of the global array 'gblA' on the root process */
  if (procId == root)
  {
    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      for (colIndex = 0; colIndex < ncols_gblA; colIndex++)
      {
        printf(" %d  ", gblA[rowIndex][colIndex]);
      }
      printf(" \n");
    }
    printf(" \n\n");
  }

  if (procId == root)
  {
    gblx = malloc(nelem_gblx*sizeof(int));
    for (index = 0; index < nelem_gblx; index++)
    {
      gblx[index] = pow((index + 1), 2.0);
    }
  }

/* Print elements of global array 'gblx' on root process */
  if (procId == root)
  {
```

```
    printf("The elements of 'gblx' on the root process are: [  ");
    for (index = 0; index < nelem_gblx; index++)
    {
      printf("%d  ", gblx[index]);
    }
    printf("]\n\n");
  }

/* Construct a local 2D array to receive scattered data */
  locA = (int**)malloc(nrows_locA * sizeof(int));

  for (rowIndex = 0; rowIndex < nrows_locA; rowIndex++)
  {
    locA[rowIndex] = (int*)malloc(ncols_locA * sizeof(int));
  }

/* Initialize the local arrays 'locA', 'locx', and 'locB'  */
  for (rowIndex = 0; rowIndex < nrows_locA; rowIndex++)
  {
    for (colIndex = 0; colIndex < ncols_locA; colIndex++)
    {
      locA[rowIndex][colIndex] = 0;
    }
  }

  for (index = 0; index < nelem_locx; index++)
  {
    locx[index] = 0;
  }

  for (index = 0; index < nelem_locB; index++)
  {
    locB[index] = 0;
  }

/* Specify number of elements scattered to each separate process */
  nelem = nrows_gblA/nproc;

/* Scatter elements of 'gblA' and 'gblx' across all 'nproc' processes */
  for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
  {
    MPI_Scatter(&gblA[rowIndex][0], nelem, MPI_INT, &locA[rowIndex][0], nelem, MPI_INT, root,
MPI_COMM_WORLD);
  }

  MPI_Scatter(gblx, nelem, MPI_INT, locx, nelem, MPI_INT, root, MPI_COMM_WORLD);

/* Carry out local matrix multiplication on all processes */
  for (rowIndex = 0; rowIndex < nrows_locA; rowIndex++)
  {
    for (colIndex = 0; colIndex < ncols_locA; colIndex++)
    {
      locB[rowIndex] += locA[rowIndex][colIndex] * locx[colIndex];
    }
  }

/* Specify number of terms to be included in the sum of elements of 'locB' on each process */
  nterms = 1;

  for (index = 0; index < nelem_gblB; index++)
  {
    MPI_Reduce(&locB[index], &gblB[index], nterms, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
  }

/* Print elements of the global array B=Ax on the root process */
  if (procId == root)
  {
    printf("The transpose of the global array B=Ax on the root process is: [ ");
    for (index = 0; index < nelem_gblB; index++)
```

```
    {
      printf("%d ", gblB[index]);
    }
    printf("]\n\n");
  }

  MPI_Finalize();

  return 0;
}
```

When we compile this code and run it on three processes, we get the following output:

```
$ mpicc -g -lm -o scatter scatter.c

$ mpirun -np 3 scatter

 0   1   2   3   4   5
 1   2   3   4   5   6
 2   3   4   5   6   7
 3   4   5   6   7   8

The elements of 'gblx' on the root process are: [  1  4  9  16  25  36  ]

The transpose of the global array B=Ax on the root process is: [ 22 36 50 64 ]
```

At first glace there does not appear to be anything glaringly wrong in this output. The global arrays *gblA* and *gblx* are correct and the global vector *gblB* contains four nonzero elements. However, if we look more carefully at our results for $B = Ax$, we can see that they make no sense whatsoever. For example, $gblB[3] = 64$. But

gblB[3] = a30 x0 + a31 x1 + . . . + a35 x5 = 3 (1) + 4 (4) + . . . + 8 (64)

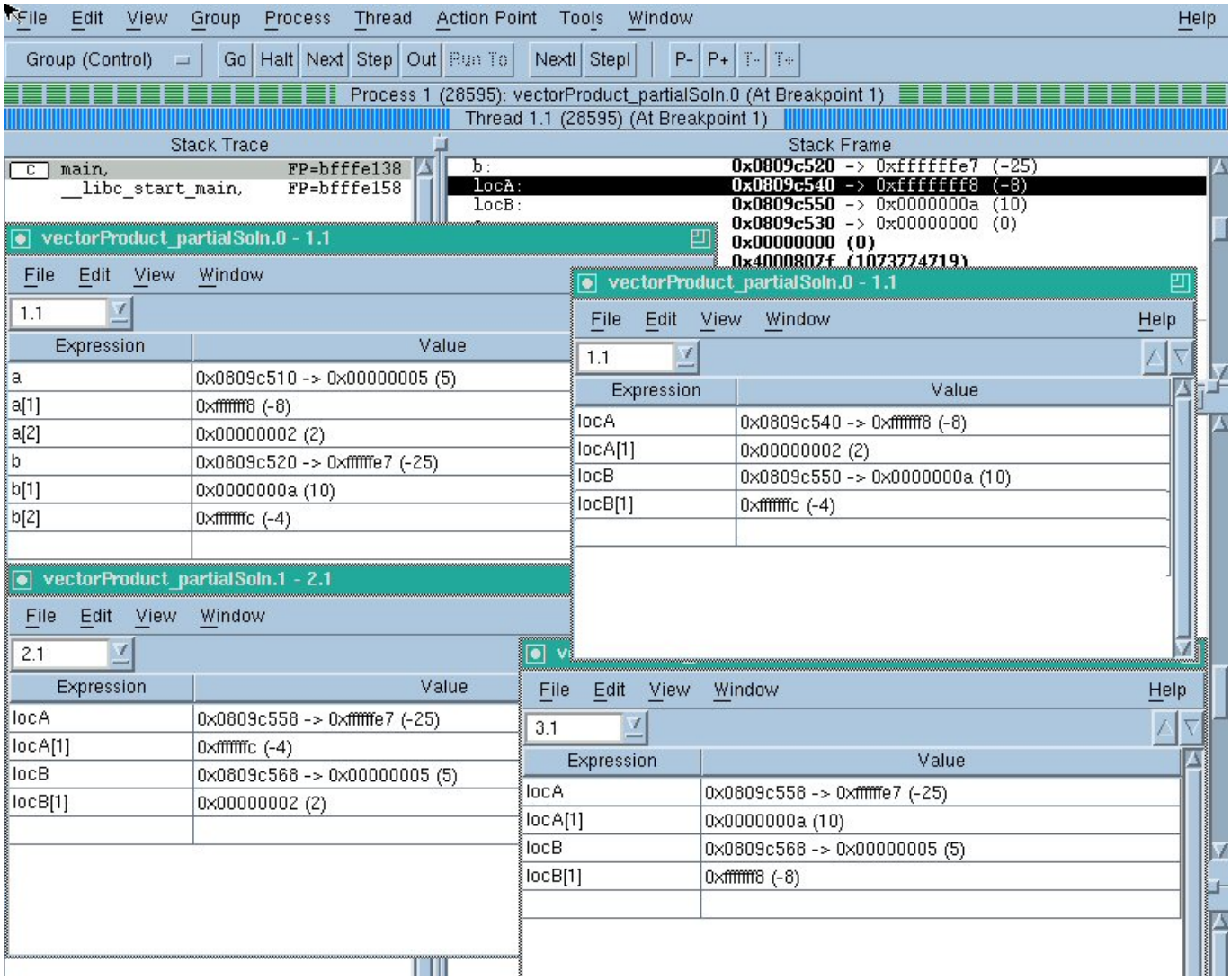
Clearly, even the last term by itself is greater than 64 and since all of the terms in this sum are greater than 0, our results for *gblB* are obviously incorrect. Therefore, there must be one or more coding errors in this program.

3.8.5 Solutions to Exercises

Exercise 1:

The solution illustrated below was obtained using TotalView®.

The fact that the vector *c* is not perpendicular either to *a* or to *b* tells us that the value of one or more of the components of the vector *c* must be incorrect. Therefore, the first thing that we should look at is the value of each of the variables used to compute *c* in the expression on line 112. That is, we should look at the values of *locA* and *locB* that was distributed across processes via the point-to-point communication events called from within the previous for( ) loop beginning on line 81. The value for each of these variables, along with the values of the global data on the root process, is shown in the following figure:

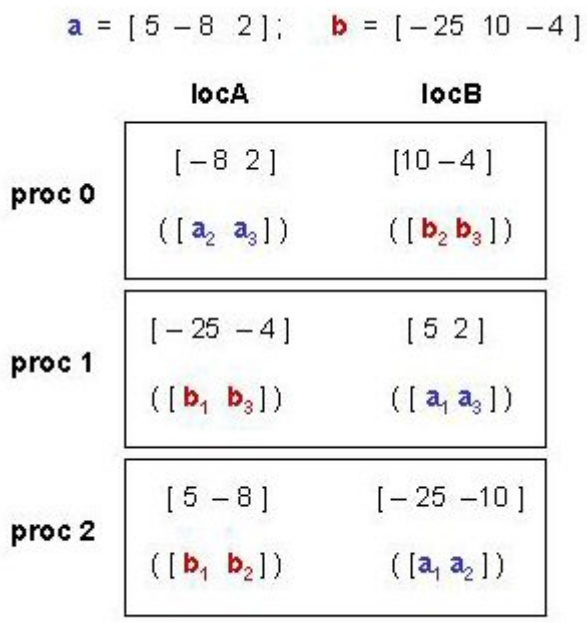


To see exactly where the values in the vectors *locA* and *locB* originate, we look at what our point-to-point communication calls are doing. In each pair of calls to MPI_SEND( ) and MPI_RECV( ), a copy of two elements from the global vector *a* and two from the global vector *b* are sent to the appropriate process,

```
a2 , a3 ;  b2 , b3    -->  proc 0
a1 , a3 ;  b1 , b3    -->  proc 1
a1 , a2 ;  b1 , b2    -->  proc 2
```

and stored as local data in the vectors *locA* and *locB*, respectively. But when we look at this data in each of our three processes, it is immediately clear that there is a problem. The values in *locA* match values in the global vector *a* only on the rank 0 process. Likewise, the values in *locB* match values in the global vector *b* only on the rank 0 process.

A logical initial guess might be that we have an error that is causing random data values to be distributed across our three processes. However, if we look carefully at the data values that have been stored in *locA* and *locB* on the rank 1 and rank 2 processes, we see that these values are actually not random and all. Instead, the values in *locA* exactly match those in the global vector *b*, whereas the values in *locB* exactly match those in the global vector *a*. In fact, even the correct pairs of data values are being distributed to the correct processes as illustrated in the following figure:



The error appears to be that the local vectors in which our data is stored are in reverse order on the rank 1 and rank 2 processes; viz.,

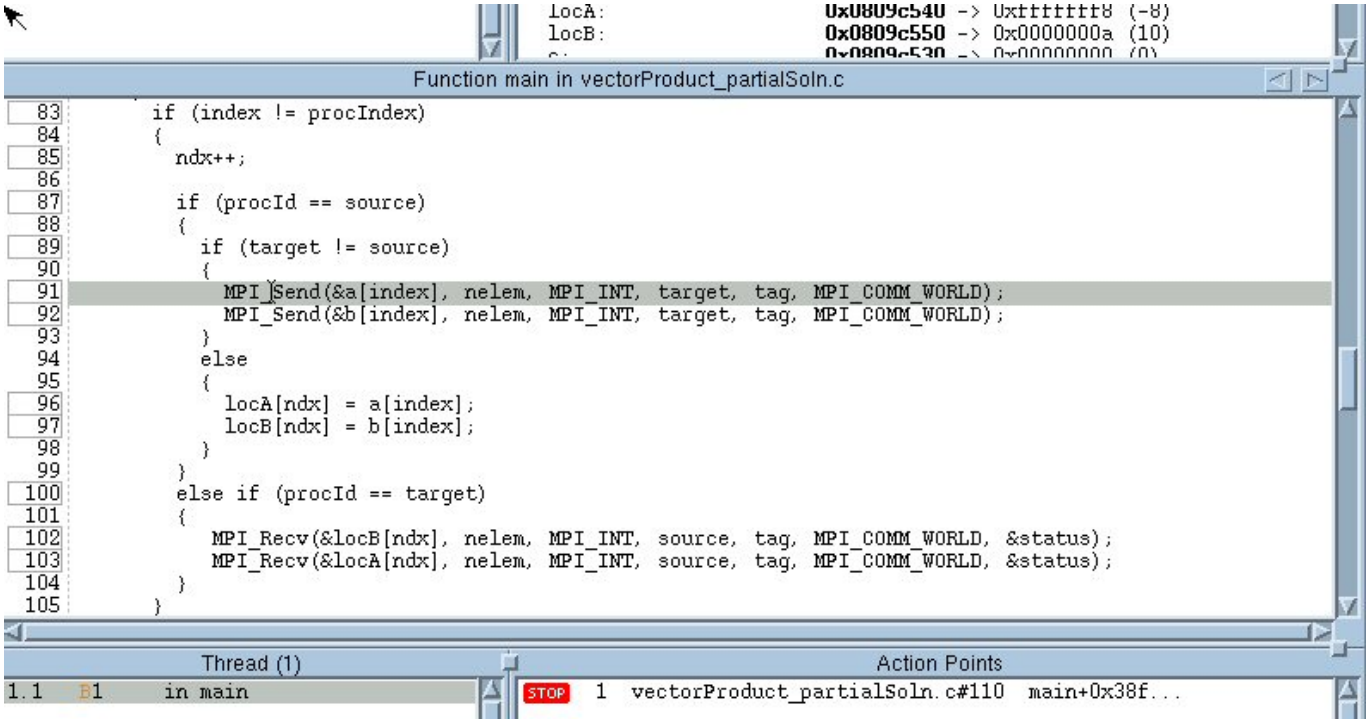
a --> locB ; b --> locA

on proc 1 and proc 2 instead of the correct order,

a --> locA ; b --> locB

Therefore, our coding error must be somewhere in our point-to-point communications call. Since the correct data elements are being sent to each process but the data is being stored in the wrong vectors, a logical guess is that the error is in our call to MPI_RECV( ). Note that we have two sequential calls to MPI_RECV( ) on the rank 1 and rank 2 processes; one to receive data into the buffer *locA* and one to receive data into the buffer *locB*. Since these buffers appear to be receiving data from the global vectors *a* and *b* in reverse order, perhaps our two calls to are in the wrong order.

If we look at these two calls to MPI_SEND( ) on lines 91 and 92 in the figure below, we see that we are sending data from proc 0 to proc 1 and proc 2 in the order a --> b .



However, on lines 102 and 103, we see that proc 1 and proc 2 are receiving data from proc 0 in the order locB --> locA; i.e., in the reverse order in which the data is being sent.

Our first thought on how to correct our coding error might be to simply reverse the order of our calls to MPI_RECV( ) on lines 102 and 103 of our code. The problem with this is that the order in which data is sent from one process is not necessarily the order in which it is received by another process. Therefore, if we reverse the order of our calls to MPI_RECV( ), but data from the second send arrives at proc 1 and/or proc 2 before the data from the first send, we would again end up with the wrong data stored in the wrong local vectors.

In order to guarantee that data from vector *a* (*b*) is received by *locA* (*locB*), we somehow need to differentiate these two different sets of data so that *locA* (*locB*) knows to receive only data from *a* (*b*). Of course we already know exactly how to do this. We simply pass a unique value for the argument tag to each pair of send/receive calls in our program. If we look at our code at the beginning of the for( ) loop on line 73 of our code and then at the code for our two calls to MPI_SEND( ) and MPI_RECV( ),

```
for (procIndex = 0; procIndex {
  source = root;
  target = procIndex;
  tag = target;

  .  .  .

        MPI_Send(&a[index], nelem, MPI_INT, target, tag, MPI_COMM_WORLD);
        MPI_Send(&b[index], nelem, MPI_INT, target, tag, MPI_COMM_WORLD);

  .  .  .

        MPI_Recv(&locB[ndx], nelem, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
        MPI_Recv(&locA[ndx], nelem, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
```

we see that we are passing exactly the value for the argument tag in both pairs of send/receive calls. Consequently, both *locA* and *locB* may receive data from either *a* or *b* independent of the order in which we place our calls to MPI_RECV( ).

Therefore, to correct our coding error we need to modify the value of the argument tag in one of these two pairs of send/receive calls. For example, we could simply change the value of one of these arguments from *tag* to (*tag + 1*). Of course, we want to make sure that we make this change in the correct pair of tags so that *locA* (*locB*) receives data from the vector *a* (*b*). Otherwise, we guarantee that data on proc 1 and proc 2 is always stored in the *locA* and *locB* in reverse order.

The correction that we need to make to our code is as follows:

```
for (procIndex = 0; procIndex < nproc; procIndex++)
{
  source = root;
  target = procIndex;
  tag = target;

  ndx = -1;

  for (index = 0; index < nelem_in_gblArray; index++)
  {
    if (index != procIndex)
    {
      ndx++;

      if (procId == source)
      {
        if (target != source)
        {
          MPI_Send(&a[index], nelem, MPI_INT, target, tag, MPI_COMM_WORLD);
          MPI_Send(&b[index], nelem, MPI_INT, target, (tag + 1), MPI_COMM_WORLD);
        }
        else
        {
          locA[ndx] = a[index];
          locB[ndx] = b[index];
        }
      }
      else if (procId == target)
      {
         MPI_Recv(&locB[ndx], nelem, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
         MPI_Recv(&locA[ndx], nelem, MPI_INT, source, (tag + 1), MPI_COMM_WORLD, &status);
      }
    }
  }
}
```

When we make this correction to our code and then recompile and run our corrected code on three processes, we get the following correct output.

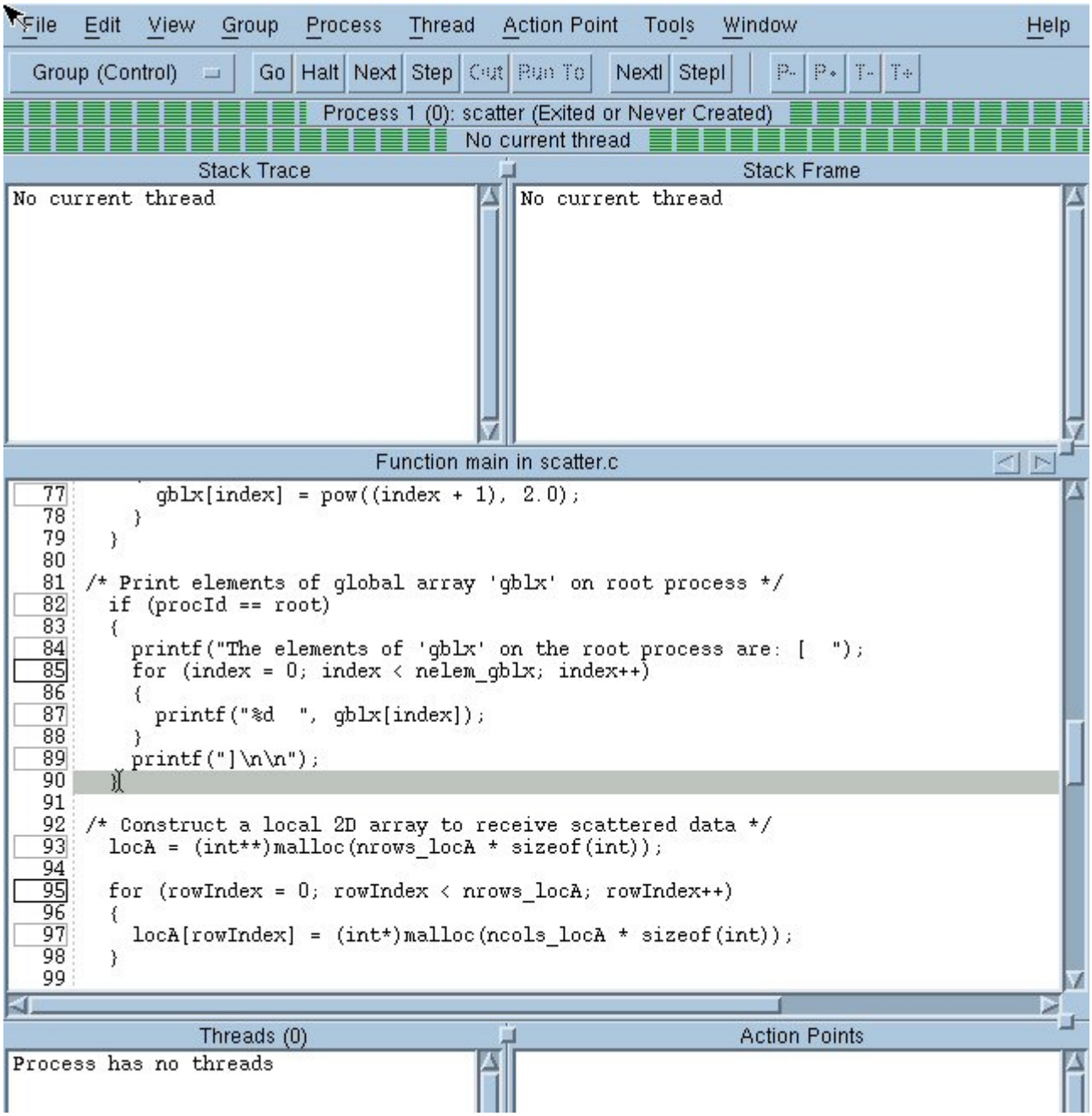
```
$ mpicc -g -lm -o vectorProduct vectorProduct.c

$ mpirun -np 3 vectorProduct

The elements of the global arrays are a = [  5  -8  2  ] ,  b = [  -25  10  -4  ]

The components of the cross-product on the root process are c = [  12   -30  -150  ]

The dot product of vector a and vector c = 0

The dot product of vector b and vector c = 0
```

Exercise 2:

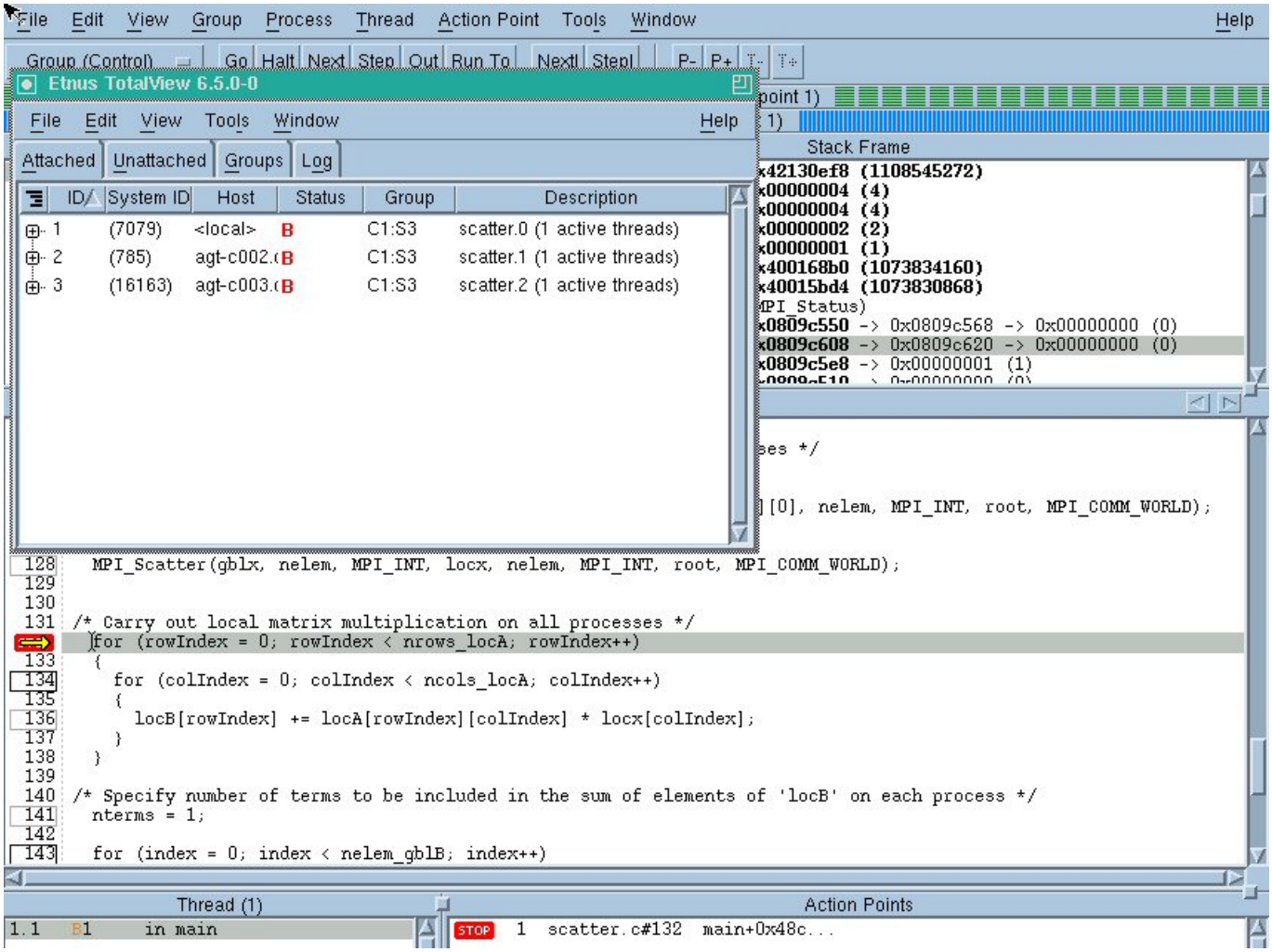
To identify the error(s) in this code, we rerun our program in TotalView® using the command:

```
$ mpirun -dbg=totalview -np 3 scatter
```

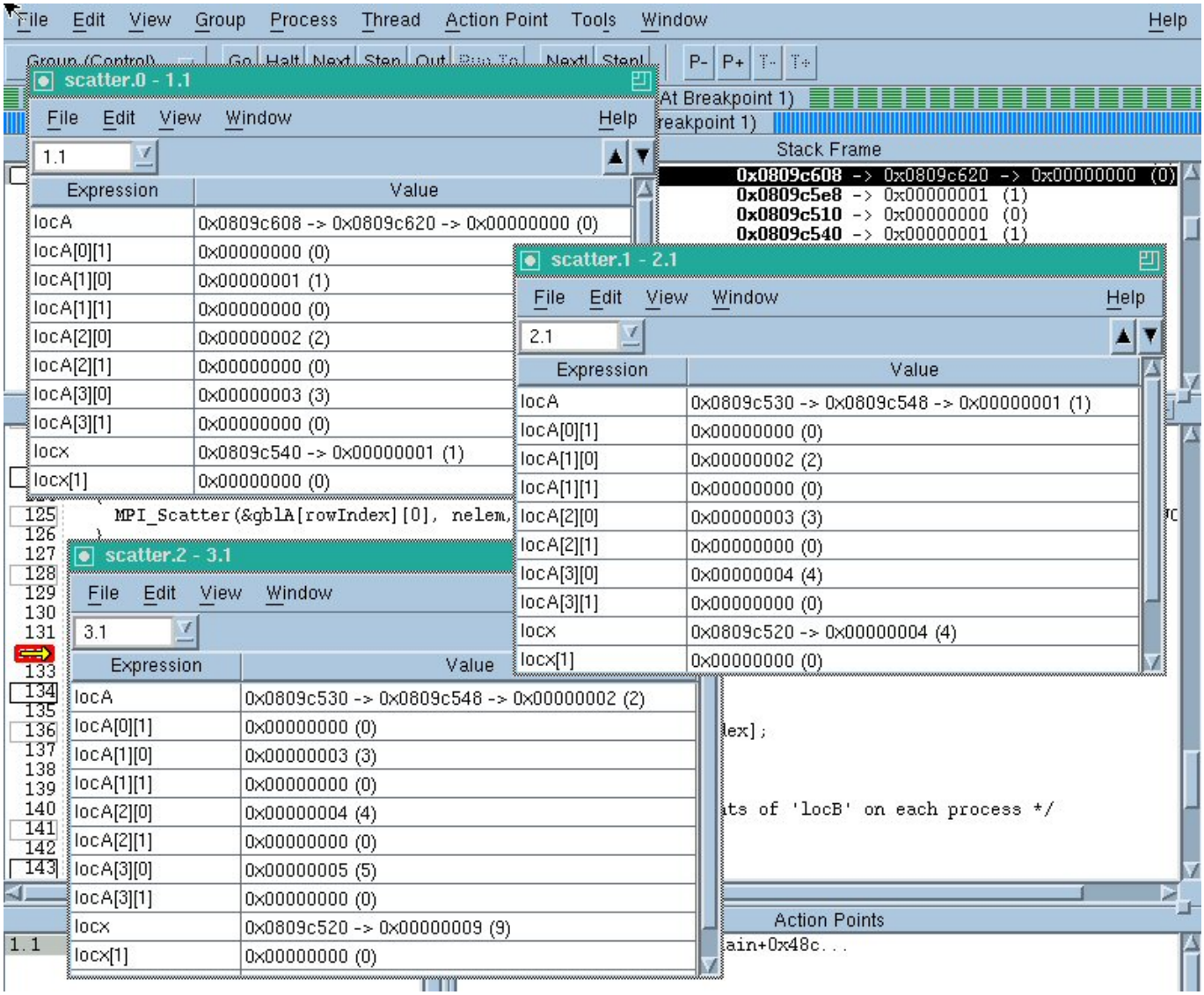
Since *gblA* and *gblx* are correct, our coding error(s) must be somewhere below line 90 in our code.



Therefore, we set our initial breakpoint below this line of code. The next section of code below this initial breakpoint where we might anticipate a coding error is in the collective communication calls to MPI_SCATTER( ). Consequently, we set our initial breakpoint on line 132, start our run in TotalView®, and examine the values for the local data that was distributed across our three processes during execution of the code immediately preceding line 132. We then start our run in TotalView®. As we can see from the information contained in the Root Window, all three of our processes have paused at the breakpoint on line 132.



However, when we examine the element values in the local arrays *locA* and *locx*, we find that something simply does not look right at all. If we focus on the element values in *locx*, we see that the second element of this vector is zero on each of our three processes as shown in the following figure:



But this can not be correct, because none of the six elements in the global vector *gblx* are zero. We also notice that the first element in *locx* on the rank *k* process is the element *gblx[k]*. This tells us that only one element from *gblx* is being distributed to each process and, since we only are running three processes, then only half of the elements in *gblx* are being distributed across these processes. Consequently, there appears to be an error in our second call on line 128 of our code.

Now if we also look at the elements in *locA[0][k]* on each of our three processes, we also see something that does not look correct.

```
proc 0:   loc A  =  [ 0 0 ]
proc 1:   loc A  =  [ 1 0 ]
proc 2:   loc A  =  [ 2 0 ]
```

As in the case with *locx*, it appears that only one element from the first row of *gblA* is being distributed to each process. And,as we can see from the figure below, it also appears that only one element from all of the other rows of *gblA* is being distributed to each process.

Clearly, only half of the data in each of our two global arrays, *gblA* and *gblx*, are being distributed across our three processes, whereas the remaining data is not being distributed at all. Consequently, an error must be occurring in each of our calls to MPI_SCATTER( ). In order to have all six columns of *gblA* and all six elements of gblx correctly distributed over three processes, two columns of data from *gblA* and two elements from *gblx* would have to be distributed to each process. This tells us that the value of the argument *nelem* being passed to MPI_SCATTER( ) must be *nelem = 2*. However, when we examine the value of *nelem* that is actually being passed during each of our calls to MPI_SCATTER( ), we find, instead, that the value of this argument being passed is actually nelem = 1, as shown in the following figure.

Clearly, our coding error appears to involve the incorrect initialization of the variable *nelem*. If we look back through our code to find where this variable was initialized, we see that on line 119 we have defined *nelem* by the expression

nelem = nrows_gblA/nproc;

Since our global matrix has four rows and we are running our program on three processes then

nelem = 4 / 3 = 1

since *nelem* has an integer data type. Obviously, if we have a matrix containing *R* rows and *C* columns, and we want to scatter the elements of this matrix evenly across *nproc* processes, then we want to distribute *C/nproc* elements, not *R/nproc* elements, to each process. Therefore, we can correct our code by changing our initialization of *nelem* on line 119 to

```
nelem = ncols_gblA/nproc;
```

When we make this correction to our code and then recompile and run our corrected code on three processes, we get the following correct output:

```
$ mpicc -g -lm -o scatter scatter.c

$ mpirun -np 3 scatter

 0   1   2   3   4   5
 1   2   3   4   5   6
 2   3   4   5   6   7
 3   4   5   6   7   8

The elements of 'gblx' on the root process are: [  1  4  9  16  25  36  ]

The transpose of the global array B=Ax on the root process is: [  350   441   532   623  ]
```

3.9 Hung Processes in Message Passing

3.9.1 Hung Processes in Message Passing

Introduction

In parallel codes using message passing, processes are typically performing independent tasks simultaneously. When the time comes to send and receive messages, certain conditions must be met in order to successfully transfer the data. One of these conditions involves blocking vs. *nonblocking* sends and receives.

In a blocking send, the function or subroutine does not return until the "buffer" (the message being sent) is reusable. This means that the message either has been safely stored in another buffer or has been successfully received by another process.

As an example, suppose the array *x* is being sent from process 0 to process 1, and immediately after the send, *x* is modified. Here is the MPI call and the modification of *x*:

```
MPI_Send(x,n,MPI_INT,1,1,MPI_COMM_WORLD);
x[1] = 10;
```

The MPI_SEND function will not return until the original *x* is either buffered or received by process 1. This prevents the original data from being overwritten in the line following the send.

Buffering of the message is system-dependent and on a given system depends on the size of the message. There is generally a maximum allowable buffer size. If the message exceeds this size, it must be received by the complimentary call (e.g., MPI_RECV) before the send function returns. This has the potential to cause processes to hang if the message passing is not handled carefully. A particularly pernicious aspect of this kind of bug is that the code may work fine in one case, where the message size happens to be smaller than the buffer limit, and then fail in another, where the message size is larger than the buffer limit. It could also work fine on one platform, and the identical run could fail on another platform with different buffer behavior.

There are of course other errors that can cause a hung process. The group of arguments containing the source, destination, tag, and communicator is known as the message envelope. If there is an error in any argument that is part of the message envelope, a hung process could result. Some examples are the specification of a non-existent destination, or a different tag in the send and receive. In these cases, the debugging procedure would be similar to that discussed in this lesson. The first step is to determine the lines at which the hang is occurring, and then to deduce the cause.

Objectives

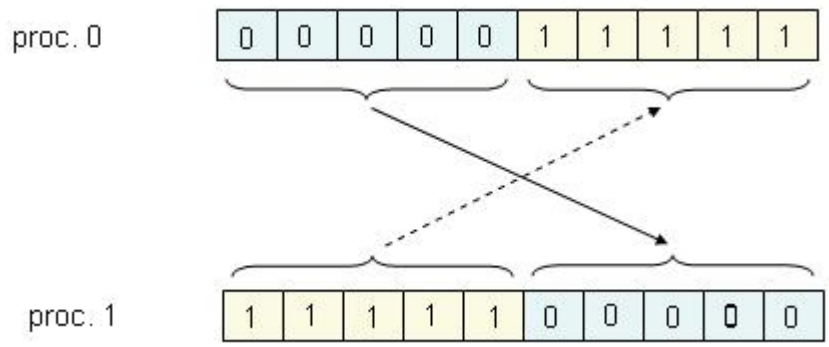
In this lesson, you will learn how message passing errors can cause hung processes and how to debug code with a hung process using the AIX debugger pdbx. A sample program is provided that you may download and debug on your own while following along with the procedure described here.

3.9.2 Sample Code with Hung Process

The following code is designed to run on exactly two processors. An array is filled with process numbers. The first half of the array is filled with the local process number, and the second half of the array is filled with the other process number. The second halves of the local arrays are filled by message passing.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void main(int argc, char *argv[]){

      int nvals, *array, myid, i;
      MPI_Status status;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);

      nvals = atoi(argv[1]);

      array = (int *) malloc(nvals*sizeof(int));
      for(i=0; i<nvals/2; i++);
            array[i&#93 = myid;

      if(myid == 0){
            MPI_Send(array,        nvals/2,MPI_INT,1,1,MPI_COMM_WORLD);
            MPI_Recv(array+nvals/2,nvals/2,MPI_INT,1,1,MPI_COMM_WORLD,&status);
         } else {
            MPI_Send(array,        nvals/2,MPI_INT,0,1,MPI_COMM_WORLD);
            MPI_Recv(array+nvals/2,nvals/2,MPI_INT,0,1,MPI_COMM_WORLD,&status);
         }

      printf("myid = %d:  array[nvals-1] = %dn",myid,array[nvals-1]);

      MPI_Finalize();

   }
```

A diagram that depicts how the arrays are filled is shown below. Each processor fills the left half of the local copy of the array with its own process number. It then passes these values to the other processor, filling the right half of the local array.



Compiling and Running

The code can be compiled and linked to the MPI library, for example under AIX:

```
% mpxlc -o hung_proc hung_proc.c
```

To run the code, a command-line argument is required specifying the size of the one-dimensional array. The size of the array at which the code fails is system-dependent, so if you want to run it on your own system you will have to experiment to find the failure point, which will always occur at some value. If the code runs successfully, it will print the last value of the array on each processor.

Now we specify increasing array sizes under AIX:

- array size 100 `% hung_proc 100 -procs 2`
  ```
  myid = 0:  array[nvals-1] = 1
  myid = 1:  array[nvals-1] = 0
  ```
- array size 1000: `% hung_proc 1000 -procs 2`
  ```
  myid = 0:  array[nvals-1] = 1
  myid = 1:  array[nvals-1] = 0
  ```
- array size 10000: `% hung_proc 10000 -procs 2`

At array size 10000, the code hangs indefinitely. (You can get back to the prompt with a CTL-c under Unix.) Again, the argument value may differ on your system. Next, we use a debugger to determine what the error is in the program.

3.9.3 Debugging the Sample Code

To aid in debugging our sample code we will use the AIX debugger pdbx, a parallel version of the common dbx debugger. This is a command-line debugger.

First we need to recompile the code with the addition of the '-g' flag, which builds a symbol table for use by the debugger. Then we can start running the code under pdbx:

```
% pdbx hung_proc 10000 -procs 2
pdbx Version 3, Release 2 -- Feb 23 2003 15:55:50

   0:Core file "
   0:" is not a valid core file (ignored)
   1:Core file "
   1:" is not a valid core file (ignored)
   0:reading symbolic information ...
   1:reading symbolic information ...
   1:[1] stopped in main at line 10 ($t1)
   1:   10         MPI_Init(&argc, &argv);
   0:[1] stopped in main at line 10 ($t1)
   0:   10         MPI_Init(&argc, &argv);
0031-504  Partition loaded ...

pdbx(all)
```

The warnings about the core file occur when a core file is not being used, and may be ignored. Each line of output starts with the process number. Using pdbx the code is automatically run and then stopped at the first executable statement, and hence both processes are seen to have stopped at the call to MPI_INIT. Control is then returned to the user at the pdbx(all) prompt, with "all" indicating that all processes will be affected by the commands.

Debugging Procedure

A debugger is very helpful in finding bugs, but remember that you will save time by thinking through the problem before starting the debugging process. In our example the code is hanging, so we cannot be sure which region of the code is causing the problem. However, we can be pretty confident the problem involves message passing since it is a common cause of a hung process.

First, we list the source code to determine where we should set a breakpoint. In parallel debugging, the amount of

information written to the screen can be overwhelming and should be minimized where possible. We do not need source listings from every process, so for now we restrict ourseles to a single processor, say processor 0. Under pdbx a process can be specified by prefixing a command with "on procno," where *procno* is the process number. Since we anticipate wanting to restrict source listings to process 0 frequently, we define an alias for it:

```
pdbx(all) alias l0 on 0 list
```

The command "on 0 list" will provide a listing from process 0 only, and we have defined this command as "l0" (letter "l"-zero) to minimize typing.

```
pdbx(all) l0
   0:   11         MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   0:   12
   0:   13         nvals = atoi(argv[1]);
   0:   14
   0:   15         array = (int *) malloc(nvals*sizeof(int));
   0:   16         for(i=0; i<nvals/2; i++)
   0:   17               array&#91:i&#93: = myid;
   0:   18
   0:   19         if(myid == 0){
   0:   20               MPI_Send(array,        nvals/2,MPI_INT,1,1,MPI_COMM_WORLD);
```

Let us look at a few more lines:

```
pdbx(all) l0
   0:   21               MPI_Recv(array+nvals/2,nvals/2,MPI_INT,1,1,MPI_COMM_WORLD,&status);
   0:   22            } else {
   0:   23               MPI_Send(array,        nvals/2,MPI_INT,0,1,MPI_COMM_WORLD);
   0:   24               MPI_Recv(array+nvals/2,nvals/2,MPI_INT,0,1,MPI_COMM_WORLD,&status);
   0:   25            }
   0:   26
   0:   27         printf("myid = %d:  array[nvals-1] = %dn",myid,array[nvals-1]);
   0:   28
   0:   29         MPI_Finalize();
```

The message-passing block starts at line 19, so that is where we set our breakpoint:

```
pdbx(all) stop at 19
all:[0] stop at "hung_proc.c":19

pdbx(all)
```

The word "all" followed by a colon indicates that all processes will stop at this breakpoint. Now we continue to this point:

```
pdbx(all) cont
   0:[6] stopped in main at line 19 ($t1)
   0:   19         if(myid == 0){
   1:[6] stopped in main at line 19 ($t1)
   1:   19         if(myid == 0){

pdbx(all)
```

Now that we are in the region of the code we want to examine, we step through the code one line at a time:

```
pdbx(all) next
   0:stopped in main at line 20 ($t1)
   0:   20               MPI_Send(array,nvals/2,MPI_INT,1,1,MPI_COMM_WORLD);
   1:stopped in main at line 23 ($t1)
   1:   23               MPI_Send(array,nvals/2,MPI_INT,0,1,MPI_COMM_WORLD);

pdbx(all) next
```

The first next command advanced each process one line. Note that process 0 stopped at line 20, while process 1 stopped at line 23 due to the if(myid == 0) conditional. The second next command did *not* result in a pdbx prompt, so the code appears to be hanging at line 20 in process 0 and/or line 23 in process 1. Both of these lines contain blocking sends so this gives us a clue as to location of the problem. A blocking send may or may not return prior to the posting of the corresponding receive, depending on the size of the message and how the messages are handled by the system. This is a classic deadlock situation, with both sends waiting for receives, but neither process being able to move on to the receive

until the send has completed. There are several ways to correct this situation, the simplest being to reverse the order of the send and receive on one processor only. If lines 23 and 24 are reversed to read

```
MPI_Recv(array+nvals/2,nvals/2,MPI_INT,0,1,MPI_COMM_WORLD,&status);
MPI_Send(array,        nvals/2,MPI_INT,0,1,MPI_COMM_WORLD);
```

the code works fine for any size array.

3.9.4 Debugging Exercise

Hung Process Debugging Exercise

Many of us have been in the situation where we have struggled for an interminable length of time to find an obscure bug in our code. We finally found the problem and think we fixed it but the problem did not go away. Now what? We can use our newly-acquired debugging skills to determine what went wrong.

Here we continue with the same example we examined in the previous section. We found the bug, reordered the second send/receive pair, and came up with the following code.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void main(int argc, char *argv[]){

      int nvals, *array, myid, i, tag, dest, source;
      MPI_Status status;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);

      nvals = atoi(argv[1]);

      array = (int *) malloc(nvals*sizeof(int));
      for(i=0; i<nvals/2; i++)
            array[i] = myid;

      if(myid == 0){
            dest = 1;
            tag = 1;
            MPI_Send(array,        nvals/2,MPI_INT,dest,tag,MPI_COMM_WORLD);
            MPI_Recv(array+nvals/2,nvals/2,MPI_INT,dest,tag,MPI_COMM_WORLD,&status);
         } else {
            source = 0;
            tag = 0;
            MPI_Recv(array+nvals/2,nvals/2,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
            MPI_Send(array,        nvals/2,MPI_INT,source,tag,MPI_COMM_WORLD);
         }

      printf("myid = %d:  array[nvals-1] = %d\n",myid,array[nvals-1]);

      MPI_Finalize();

   }
```

Upon testing this code, we find that things have gotten even worse - it hangs every time, no matter what size we choose for the array! Use your favorite debugger to solve the new problem. (Hint: Follow the same procedure used in the previous section to determine where the code is hanging, and examine the arguments of the hung routines.)

3.9.5 Solution to Exercise

The send and receive calls had been reordered correctly, but an error was introduced in the message tag on the receive end. Processor 0 is sending a message with a tag of 1, and processor 1 is trying to receive a message with a tag of 0. Processor 1 is patiently waiting for a message with the specified tag, and no message with that tag was ever sent, hence the hung processes.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <mpi.h>

void main(int argc, char *argv[]){

      int nvals, *array, myid, i, tag, dest, source;
      MPI_Status status;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);

      nvals = atoi(argv[1]);

      array = (int *) malloc(nvals*sizeof(int));
      for(i=0; i<nvals/2; i++)
            array[i&#93: = myid;

      if(myid == 0){
            dest = 1;
            tag = 1;
            MPI_Send(array,        nvals/2,MPI_INT,dest,tag,MPI_COMM_WORLD);
            MPI_Recv(array+nvals/2,nvals/2,MPI_INT,dest,tag,MPI_COMM_WORLD,&status);
         } else {
            source = 0;
            tag = 1;
            MPI_Recv(array+nvals/2,nvals/2,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
            MPI_Send(array,        nvals/2,MPI_INT,source,tag,MPI_COMM_WORLD);
         }

      printf("myid = %d:  array[nvals-1] = %d\n",myid,array[nvals-1]);

      MPI_Finalize();

   }
```

3.10 Dynamic Memory Allocation Errors

3.10.1 Dynamic Memory Allocation Errors

Introduction

Frequently, computer programs are designed to handle a variety of applications that may require the declaration of varying array sizes to accommodate the applications. Dynamic memory allocation is commonly used to facilitate these size variations in order to allocate the proper array size. The utility tools for dynamic memory allocation vary according to the specific language: *malloc* and *calloc* are the standard for C while *allocate* is used for Fortran 9x. In parallel processing, dynamic memory allocation is very useful for efficient array layouts since an array's size can change from processor to processor.

Several of the common errors associated with dynamic memory allocation are:

1. Allocatable array not allocated
2. Array allocated, but with zero size
3. Array allocated, but with wrong size

Compilers behave quite differently when confronted with bugs of this nature. Some may warn you about the mismatches; others do not. Similarly, at runtime some executables throw a segmentation fault while others may simply give wrong answers. We examine the three types of error listed above individually in the following sections.

Objectives

In this lesson, you will learn about dynamic memory allocation and the common errors associated it. A sample program is given to illustrate a simple dynamic memory allocation process. The sample program is then modified to show some of the common errors.

3.10.2 Sample Dynamic Allocation Code

The following example program, allocated_example.c, is written for 2 processes: process 0 sends the content of an array to process 1; concurrently, process 1 probes process 0 for a message, determines its size, dynamically allocates the memory, then receives the message. The dynamic array (or pointer to an array), *rdata* is declared. The dynamic allocation of *rdata* is

done via *calloc*.

```
#include &#139mpi.h&#155
#include &#139math.h&#155
#include &#139stdio.h&#155

void main(int argc, char* argv[])
{
      double sdata[55];
      double *rdata;
      int p, i, count, myid, n;
      MPI_Status status;

      MPI_Init(&argc, &argv);                /* starts MPI */
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get current process id */
      MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */

      if (myid == 0) {
        for(i=0;i
```

3.10.3 Debugging Sample Codes

3.10.3.1 Debugging Sample Codes

In the following sections, we debug some sample codes.

3.10.3.2 Allocatable Array Not Allocated

In this example, the statement that dynamically allocates *rdata* via *calloc*

rdata = (double*) calloc(count,n);

is commented out of the sample program, allocated_example.c, to create a bug caused by an allocatable array not being allocated.

```
#include &#139mpi.h&#155
#include &#139math.h&#155
#include &#139stdio.h&#155

void main(int argc, char* argv[])
{
      double sdata[55];
      double *rdata;
      int p, i, count, myid, n;
      MPI_Status status;

      MPI_Init(&argc, &argv);                /* starts MPI */
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get current process id */
      MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */

      if (myid == 0) {
        for(i=0;i
```

3.10.3.3 Allocatable Array Allocated with Zero Length

In this example, the statement

/* rdata = (double*) calloc(count, &n); */

of the preceding example, not_allocated_example.c, is replaced with

```
count2 = 0;
rdata = (double*) calloc(count2, &n);
```

to create the situation where *count*, the size of the dynamic array, is zero.

```
#include &#139mpi.h&#155
#include &#139math.h&#155
#include &#139stdio.h&#155
```

```
void main(int argc, char* argv[])
{
      double sdata[55];
      double *rdata;
      int p, i, count, myid, n, count2;
      MPI_Status status;

      MPI_Init(&argc, &argv);                /* starts MPI */
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get current process id */
      MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */

      if (myid == 0) {
        for(i=0;i
```

3.10.3.4 Allocatable Array Allocated with Wrong Size

In this example, the statement

rdata = (double*) calloc(count, &n);

of the first example (not_allocated_example.c) is replaced with

```
count2 = count*0.5;
rdata = (double*) calloc(count2, &n);
```

to create the situation where *count2* is only half the intended size.

```
#include &#139mpi.h&#155
#include &#139math.h&#155
#include &#139stdio.h&#155

void main(int argc, char* argv[])
{
      double sdata[55];
      double *rdata;
      int p, i, count, myid, n, count2;
      MPI_Status status;

      MPI_Init(&argc, &argv);                /* starts MPI */
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get current process id */
      MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */

      if (myid == 0) {
        for(i=0;i
```

3.10.4 Exercises

1. Use your favorite debugger to test the not_allocated_example.c (or not_allocated_example.f90).
2. Use your favorite debugger to test the zero_length_example.c (or zero_length_example.f90).
3. Use your favorite debugger to test the wrong_size_example.c (or wrong_size_example.f90).
4. What if you provide an inconsistent data type between MPI_SEND and MPI_RECV? Modify one of the codes in previous sections to test it. (The modified program for F90 is provided here)

3.10.5 Solutions

1. Use your favorite debugger to test the not_allocated_example.c (or not_allocated_example.f90). This was a practice exercise so there is no real solution. If you need help, see the section "Allocatable Array Not Allocated."
2. Use your favorite debugger to test the zero_length_example.c (or zero_length_example.f90). This was a practice exercise so there is no real solution. If you need help, see the section "Allocatable Array Allocated with Zero Length."
3. Use your favorite debugger to test the wrong_size_example.c (or wrong_size_example.f90). This was a practice exercise so there is no real solution. If you need help, see the section "Allocatable Array Allocated with Wrong Size."
4. What if you provide an inconsistent data type between MPI_SEND and MPI_RECV? Modify one of the codes in previous sections to test it. (The modified program for F90 is provided here) This was a practice exercise so there is no real solution. The approach is similar to the first three exercises.

3.11 Debugging Collective Communications

3.11.1 Debugging Collective Communications

In this lesson you will learn about the most commonly used collective communications in MPI and the possible coding errors associated with them. A sample program containing a collective communications error is debugged using the Totalview® debugger. Two exercises are provided at the end of the lesson to test your debugging skills.

A collective communication event is a communications event that involves all processes contained within a given communicator. Since any communicator will usually contain more than a single process, collective communications almost always involve multiple processes. The most commonly used collective communications in MPI are:

- MPI_BCAST()
- MPI_SCATTER()
- MPI_GATHER() , MPI_ALLGATHER()
- MPI_REDUCE() , MPI_ALLREDUCE()

We now explain each of these communication events and describe possible coding errors associated with them.

MPI_BCAST()

A broadcast communication event is one in which a single specified process (the root process) sends the identical data to all processes contained within a given communicator. In MPI, a broadcast is requested by issuing the call

```
MPI_Bcast(&data, nelem, datatype, root, comm)
```

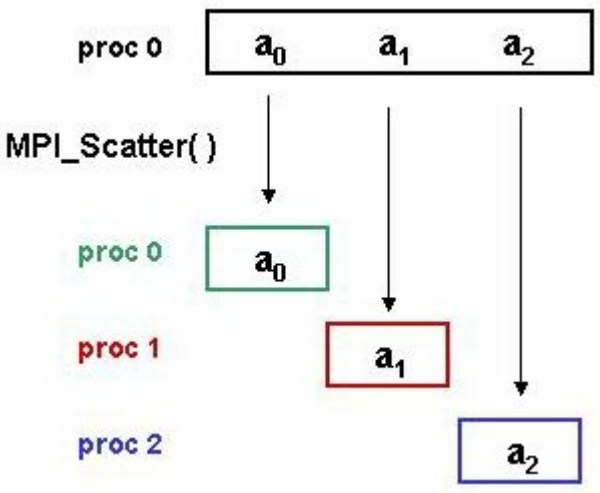
Following this call, a copy of the information contained in data stored on the rank root process is sent to each process in the communicator *comm*. This call must be issued by all processes in the communicator *comm*, and all processes issuing this call must have the same values for both *root* and *comm*. The combination of the parameters *nelem* (the number of elements being broadcast to all processes) and *datatype* (the datatype of the *nelem* of data being broadcast) determine the amount of memory needed to store the data on each process.

There are a number of possible coding errors within an MPI program that can cause a broadcast communications event to fail. These errors include:

- calling MPI_BCAST() on only a subset of processes that are contained within a given communicator, *comm*
- attempting to broadcast data that does not exist on the root process
- failing to define the parameter data on all processes in *comm*
- specifying different values for *nelem* or datatype on different processes in *comm*
- failing to specify the same value for *root* on each process in *comm*

MPI_SCATTER()

In a scatter communication event, elements of a global array of data, *gblArray*, located on the root process are distributed across all processes contained within a given communicator.



In MPI, a scatter is requested by issuing the call

```
MPI_Scatter(&gblArray, snelem, sdatatype, &locArray, rnelem, rdatatype, root, comm)
```

As with all collective communication calls, MPI_SCATTER() must be called by all processes in the communicator *comm* and each of these processes must have the same value for the parameter *root*.

MPI_SCATTER() splits the data in an *n x m* global array *gblArray*, stored on the rank root process, into *nproc* equal segments with each segment containing *snelem = m/nproc* elements of type *sdatatype*. The k^{th} segment of data is then sent to the rank *(k□1)* process where *rnelem* of data of type *rdatatype* are stored in the local array *locArray*. In almost all cases *snelem = rnelem* and *sdatatype = rdatatype*.

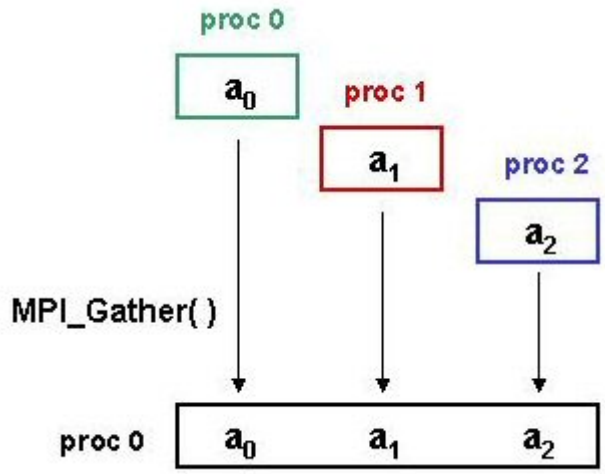
There are a number of possible coding errors within an MPI program that can cause a scatter communications event to fail. These errors include:

- calling MPI_SCATTER() on only a subset of processes that are contained within a given communicator, *comm*
- attempting to scatter data that does not exist on the root process
- failing to allocate memory for *locArray* on all processes in *comm*
- splitting the data in *gblArray* incorrectly on the root process by incorrectly specifying *snelem*
- failing to specify the same value for root on each process in *comm*

For an example that illustrates debugging a program containing a coding error in a call to MPI_SCATTER(), see the lesson titled "Incorrect Local Memory Distribution" in this tutorial.

MPI_GATHER()

In a gather communication event, elements of local data are collected from each process contained in a given communicator and stored in a global array *gblArray*, on a single process. MPI_GATHER() is, in essence, the inverse of MPI_SCATTER().



In MPI, a gather is requested by issuing the following call:

```
MPI_GATHER(&locArray, snelem, sdatatype, &gblArray, rnelem, rdatatype, root, comm)
```

MPI_GATHER() collects *snelem* of data of type *sdatatype* contained in the local array *locArray* and stores *rnelem* of that data in rank process order on the rank root process. Note that since data gathered by a call is stored in process rank order, the sequence of data in the global array on the root process will be *a0*, *a1*, and *a2*, where *ak* is the data collected from the rank *k* process.

As with MPI_SCATTER(), MPI_GATHER() must be called by all processes in the communicator *comm* and each of these processes must have the same value for the parameter root. In addition, as with MPI_SCATTER(), it is almost always the case that *snelem = rnelem* and *sdatatype = rdatatype*.

The same types of coding errors that can cause MPI_SCATTER() to fail can cause MPI_GATHER() to fail. In addition to these types of errors the following error also can cause MPI_GATHER() to fail :

- failing to allocate memory for the global array *gblArray* on all processes in the communicator *comm*

MPI_ALLGATHER()

In contrast to MPI_GATHER(), in an allgather communication event elements of local data are collected from each process in a communicator and are stored in a global data array on all of the processes in the communicator.

The same types of coding errors that can cause MPI_GATHER() to fail can cause MPI_ALLGATHER() to fail. In addition, the following error can also cause MPI_ALLGATHER() to fail,

- failing to allocate memory for the global array *gblArray* on all processes in the communicator comm

.

MPI_REDUCE()

In a reduction communication event, an operand or set of *nelem* operands stored on each process in a given communicator is combined using a specified binary mathematical operation that is successively applied to the operand(s) on each process. The result of this combined binary operation is then stored in the variable result on the rank root process.

In MPI, a reduction is requested by issuing the call

```
MPI_REDUCE(&operand, &result, nelem, datatype, operator, root, comm)
```

There are two types of operators that can be passed to MPI_REDUCE(). The first is one of the predefined MPI reduction operators. A second type is a user-defined reduction operator. Consequently, one type of error that can cause MPI_REDUCE() to fail is the use of a user-defined reduction operator that has been defined incorrectly.

As with all collective communications events, MPI_REDUCE() must be called by all processes in the communicator *comm* and each of these processes must have the same value for the parameter *root*. Not doing so will result in a failure of the call to MPI_REDUCE().

Note also that in the above call to MPI_REDUCE(), we have used a different parameter name for operand and result. Attempting to use the same name for both parameters, which is termed aliasing parameters, is illegal in MPI and will result in an error. An example of this is shown below.

If we change the call to MPI_REDUCE() on line 149 of our scatter.c code example (see Section 4.6.2.2, Collective Communications Call Error) debugged for MPI_SCATTER() from

```
MPI_REDUCE(&locB[index], &gblB[index], nterms, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
```

to

```
MPI_REDUCE(&locB[index], &locB[index], nterms, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
```

we get the following output:

```
$ mpirun -np 3 scatter
 0   1   2   3   4   5
 1   2   3   4   5   6
 2   3   4   5   6   7
 3   4   5   6   7   8



The elements of 'gblx' on the root process are: 1 4 9 16 25 36


0 - MPI_REDUCE : Invalid buffer pointer: Arguments must specify different buffers (no aliasing)
[0]  Aborting program !
[0] Aborting program!
p0_24640:  p4_error: : 8641
Killed by signal 2.
Killed by signal 2.
```

MPI_ALLREDUCE()

In contrast to MPI_REDUCE(), in an allreduce communication event, the result of the combined binary operation stored in *result*, the second parameter passed to MPI_ALLREDUCE(), is stored on all processes contained in the communicator *comm*.

The same types of coding errors that can cause MPI_REDUCE() to fail can cause MPI_ALLREDUCE() to fail. In addition, the following error can also cause MPI_ALLREDUCE() to fail,

- failing to allocate memory for the parameter result on all processes in the communicator *comm*

3.11.2 Sample Collective Communications Code

The code shown below, broadcast.c, is a C program for computing the elements of the vector B for the equation

A x = B

where A is an *n x m* matrix, *x* is an *m x 1* column vector, and B is an *n x 1* column vector.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

int main( int argc, char *argv[])
{
  int root;
  int nelem, nterms, ndim, ncount;
  int nproc, procId, index, ndx, procIndex, source, target, tag;
  int rowIndex, colIndex;
  int **gblA, *gblx, *gblB;
  int *locA, *locx, *locB;
  int nrows_gblA, ncols_gblA;
  int nelem_locA, nelem_gblx, nelem_gblB;
  int nelem_locx, nelem_locB;

  MPI_Status status;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&procId);

/* Define dimensions of global arrays for Ax=B */
  nrows_gblA = 4, ncols_gblA = 6;
  nelem_gblx = ncols_gblA;
  nelem_gblB = nrows_gblA;

/* Define dimensions of local arrays for Ax=B */
  nelem_locA = ncols_gblA;
  nelem_locx = ncols_gblA;
  nelem_locB = nrows_gblA;

/* Specify root as the rank 0 process */
  root = 0;

/* Dynamically allocate memory for the global arrays on root process */
  if (procId == root)
  {
  /* Construct 'gblA' */
    gblA = (int**)malloc(nrows_gblA * sizeof(int));

    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      gblA[rowIndex] = (int*)malloc(ncols_gblA * sizeof(int));
    }

  /* Initialize 'gblA' */
    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      for (colIndex = 0; colIndex < ncols_gblA; colIndex++)
      {
        gblA[rowIndex][colIndex] = (((nrows_gblA - 1) * rowIndex) + colIndex) + 1;
      }
    }

  /* Construct 'gblx' and 'gblB' */
    gblx = malloc(nelem_gblx * sizeof(int));
    gblB = malloc(nelem_gblB * sizeof(int));

  /* Initialize 'gblx' and 'gblB' */
    for (index = 0; index < nelem_gblx; index++)
```

```
    {
      gblx[index] = index + 1;
    }

    for (index = 0; index < nelem_gblB; index++)
    {
      gblB[index] = 0;
    }
  }

/* Dynamically allocate memory for all local arrays on all processes */
  locA = malloc(nelem_locA * sizeof(int));
  locx = malloc(nelem_locx * sizeof(int));
  locB = malloc(nelem_locB * sizeof(int));

  for (index = 0; index < nelem_locA; index++)
  {
    locA[index] = 0;
  }

  for (index = 0; index < nelem_locx; index++)
  {
    locx[index] = 0;
  }

  for (index = 0; index < nelem_locB; index++)
  {
    locB[index] = 0;
  }


/* Print the global arrays 'gblA' and 'gblx' on the root process */
  if (procId == root)
  {
  /* 'gblA' */
    printf("The elements of the global array 'gblA' are:\n");

    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      for (colIndex = 0; colIndex < ncols_gblA; colIndex++)
      {
        printf("  %d", gblA[rowIndex][colIndex]);
      }
      printf("\n");
    }
      printf("\n");

  /* 'gblx' */
    printf("The elements of the global array 'gblx' are: [ ");

    for (index = 0; index < nelem_gblx; index++)
    {
      printf("%d ", gblx[index]);
    }
    printf("]\n\n");
  }

  MPI_Datatype vtype;
  MPI_Type_contiguous(ncols_gblA, MPI_INT, &vtype);
  MPI_Type_commit(&vtype);

/* Specify source as root process */
  source = root;

  for (procIndex = 0; procIndex < nproc; procIndex++)
  {
    target = procIndex;
    tag = target;
    nelem = 1;
```

```
    if (target != source)
    {
      if (procId == source)
      {
        MPI_Send(&gblA[target][0], nelem, vtype, target, tag, MPI_COMM_WORLD);
      }
      else if (procId == target)
      {
        MPI_Recv(locA, nelem, vtype, source, tag, MPI_COMM_WORLD, &status);
      }
    }
    else if (procId == source)
    {
      locA = &gblA[target][0];
    }
  }

/* Point locx to the address of gblx */
  if (procId == root)
  {
    locx = gblx;

/* Broadcast local array 'locx' to all processes */
    MPI_Bcast(locx, nelem_locx, MPI_INT, root, MPI_COMM_WORLD);
  }

  /* Carry out local matrix multiplication, Ax=B, on each process */
    for (index = 0; index < nelem_locx; index++)
    {
      locB[procId] += locA[index] * locx[index];
    }

    nterms = 1;

  /* Sum elements of 'locB' on all processes and store results on root process */
    for (index = 0; index < nelem_locB; index++)
    {
      MPI_Reduce(&locB[index], &gblB[index], nterms, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
    }

  /* Print elements of 'gblB' on root process */
    if (procId == root)
    {
      printf("The elements of 'gblB' on the root process is: [ ");
      for (index = 0; index < nelem_gblB; index++)
      {
        printf("%d ", gblB[index]);
      }
      printf("]\n\n");
    }


  MPI_Finalize();

  return 0;
}
```

When we compile and run this code on four processes, we obtain the following output:

```
$ mpicc -g -o broadcast broadcast.c

$ mpirun -np 4 broadcast
```

The elements of the global matrix *gblA* on the root process are

```
 1    2    3    4    5    6
 4    5    6    7    8    9
 7    8    9   10   11   12
10   11   12   13   14   15
```

The elements of the global array *gblx* are: [1 2 3 4 5 6]

The elements of *gblB* on the root process is: [91 0 0 0]

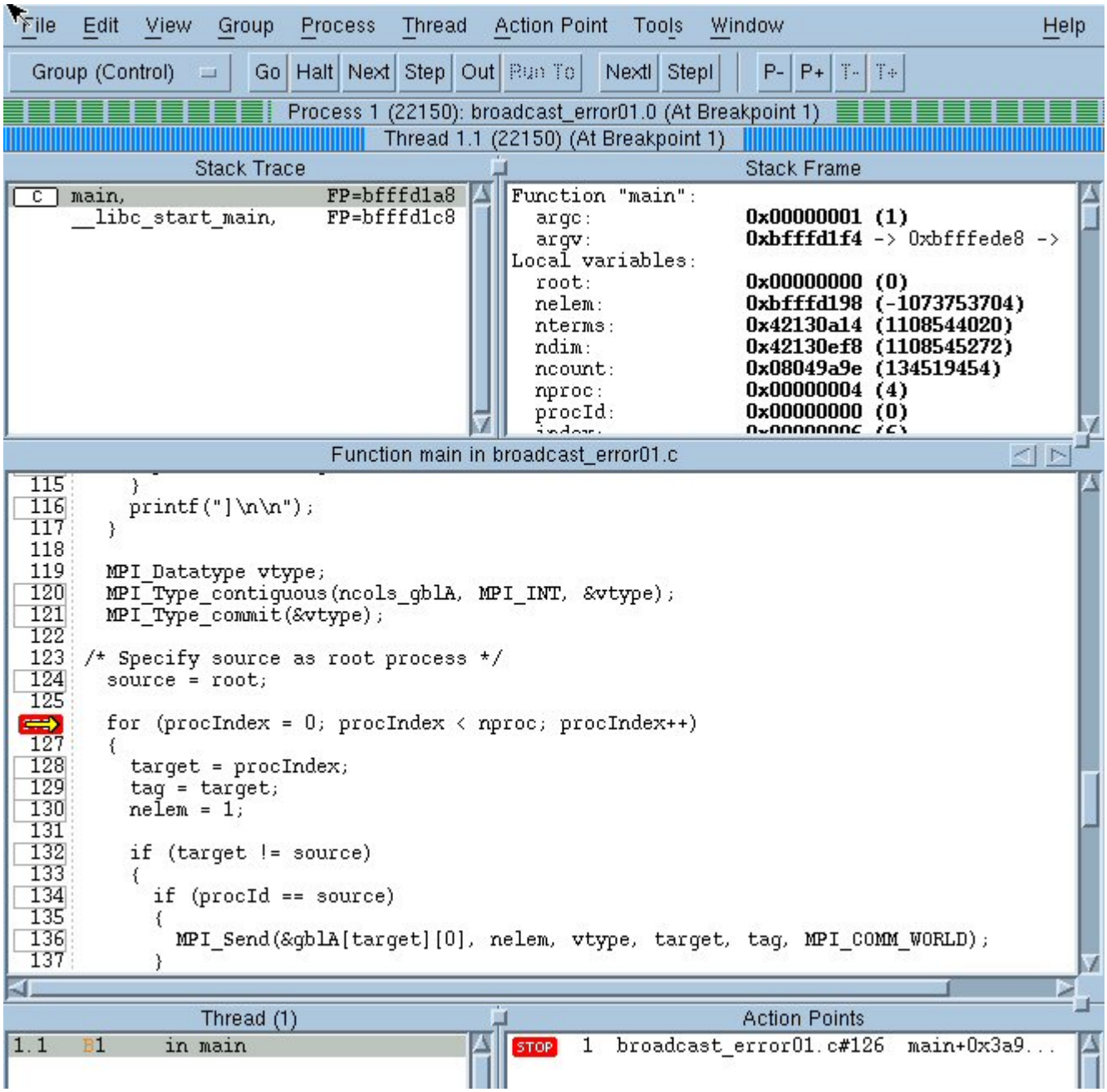
Clearly, we have a coding error somewhere in our program. In order to identify this error, we will debug our code using the TotalView® debugger.

3.11.3 Debugging the Sample Code

To debug the sample code using Totalview®, we enter the following command:

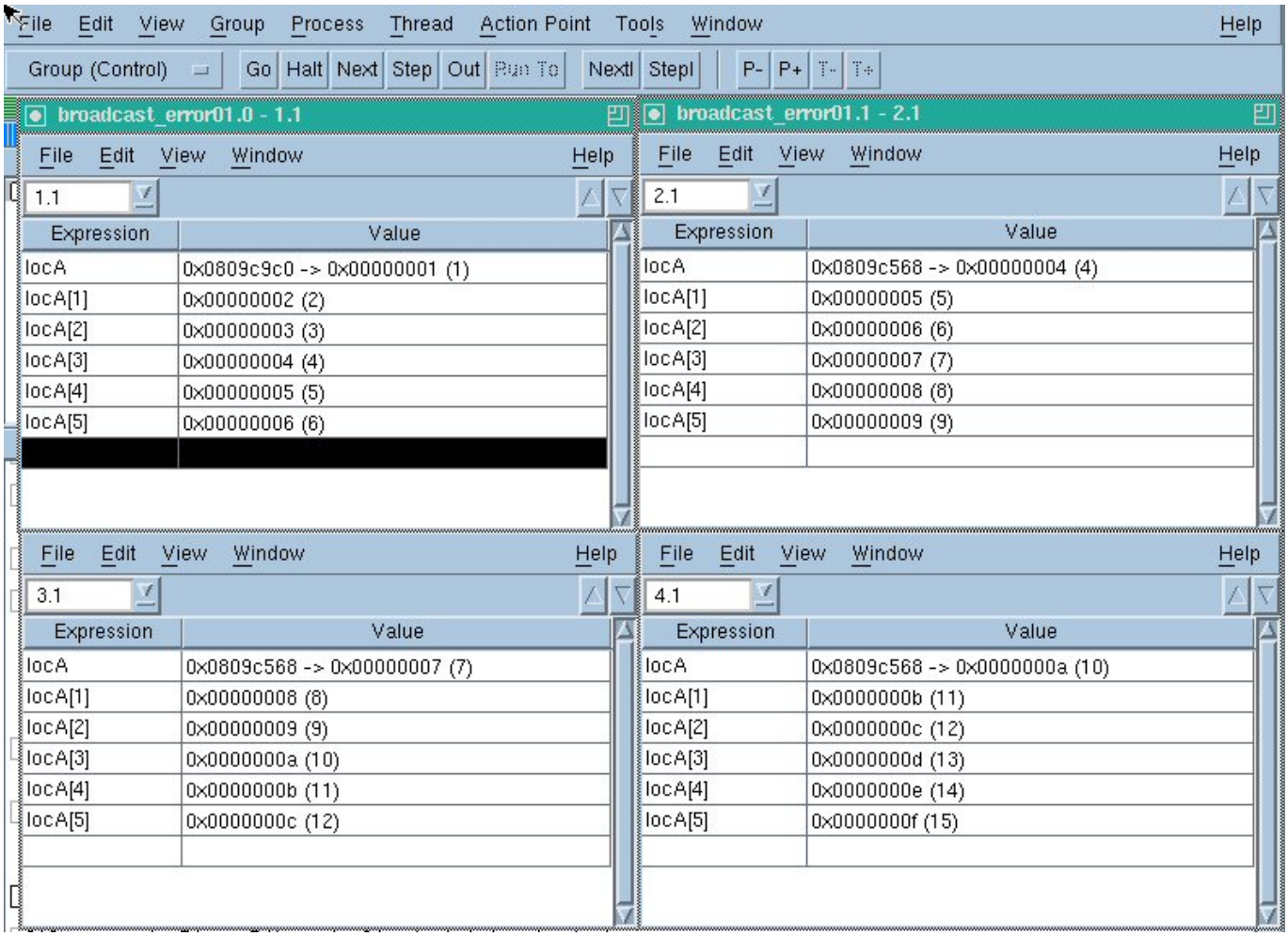
```
$ mpirun dbg=totalview -np 4 broadcast
```

Obviously, our error occurs somewhere after the section of code that prints out the elements of the global vector *gblx* on the root process, so we will set our first breakpoint somewhere following this section of code. As shown in the illustration below, the code fragment that prints *gblx* ends on line 117, so we will set a breakpoint on line 126 at which point we enter the section of our program that executes a set of *nproc* point-to-point communication events. Then we start our run in TotalView® by clicking on the 'Go' button in the second toolbar across the top of the Process Window.

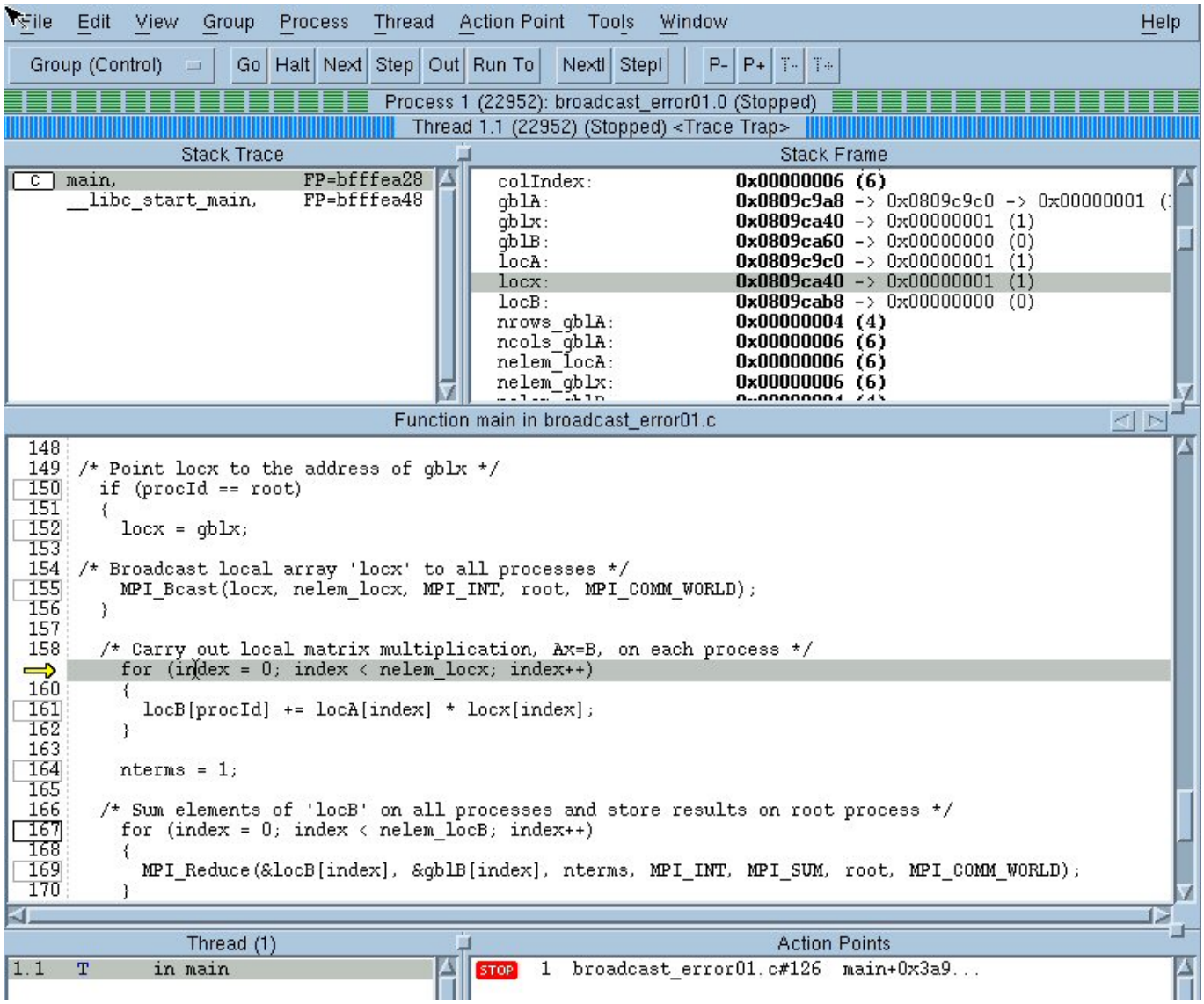


To see if our error is located in this part of our program, we traverse this for loop *nproc* times and then look at the values

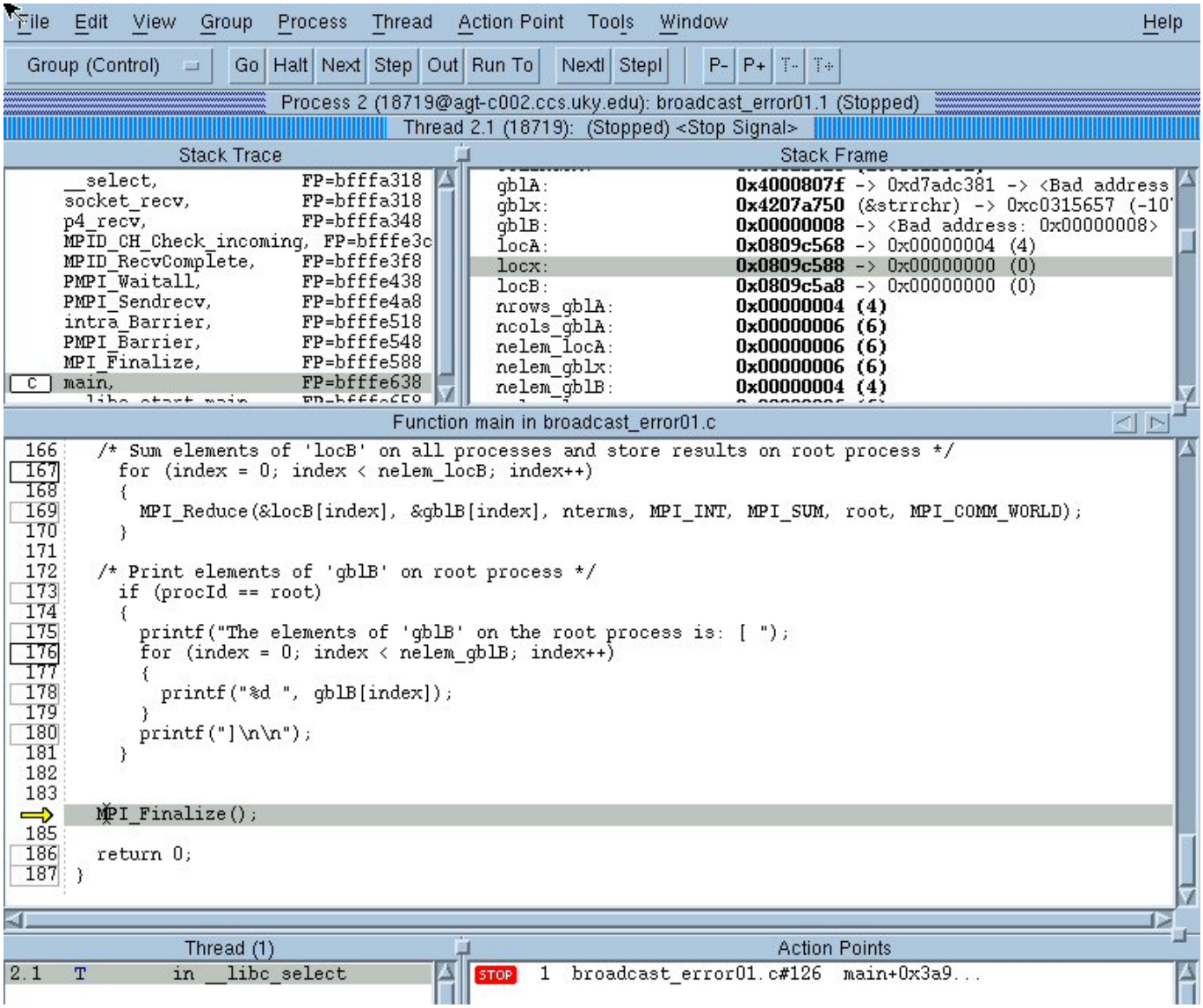
of the elements of the local matrix *locA* on each of our four processes. If this is where our error is located, *locA* would likely help us identify exactly what this error might be. After exiting this for loop, we see that the element values for *locA* are correct on all four processes as illustrated in the figure below.



Since there appears to be no errors in the coding of our point-to-point communications event, we continue to move through our code to see if we can find anything that looks unusual. As soon as we exit the point-to-point communications code, we immediately initialize the array *locx* on the root process on line 152 and then execute a collective communications call to broadcast the global vector *gblx* to all processes. However, when we execute our call to MPI_BCAST() on line 155, we immediately see something that does not look right. On Process 1 (our root process), the pointer in the Function pane of the Process Window moves to line 159 as shown in the following figure.



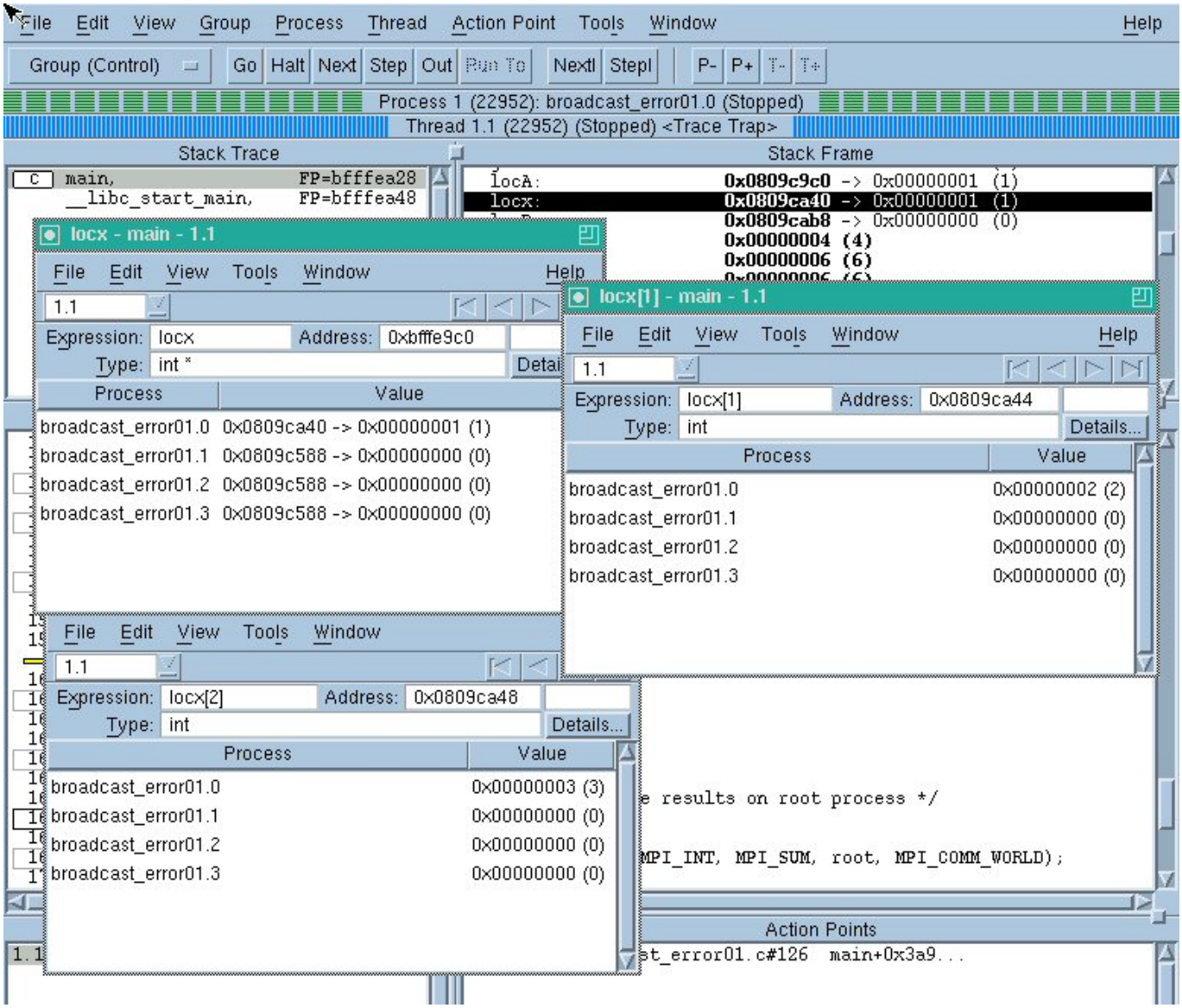
However, in the Function pane of the same window on our other three processes (process 2, 3, and 4), the pointer moves to line 184 at which point the program executes a call to MPI_FINALIZE() to abort the run on these processes.



Consequently, our error appears to be in the collective communication event on line 155 of our code:

```
MPI_Bcast(gblx, nelem_gblx, MPI_INT, root, MPI_COMM_WORLD);
```

To identify our error we look at the values for each of the arguments passed to MPI_BCAST() to see if anything looks incorrect. When we look at the value of the first argument in our broadcast function, *locx*, we immediately identify the problem. Examining the values of the elements of *locx* on each process, we find that these values are correct on the root process but are all incorrect on each of the other processes. This is illustrated below for the first three elements of *locx*. As we can see, the elements of *locx* are all zero on every process except the root process.



This suggests that

1. MPI_BCAST() is not being called by each of the processes on which *locx* is zero, or;
2. one or more arguments passed to MPI_BCAST() are being passed incorrectly.

Since our program pauses at line 155 only on proc 0 but skips to the end of the program on all other processes in our communicator, it appears that the our first suggestion may be correct. That is, the only process that is calling MPI_BCAST() is our rank 0 process. What is causing all processes other than the rank 0 process to be excluded from this collective communications call? A logical guess would be that MPI_BCAST() is being called from inside an if statement such as

```
if (procId == 0)
{
   MPI_BCAST( .  .  . );
 }
```

If we carefully examine our code, we see that this is exactly what is happening - we have placed our call to MPI_BCAST () on line 155 inside the body of the if statement that begins on line 150,

```
if (procId == root)
{
  locx = gblx;

/* Broadcast local array 'locx' to all processes */
  MPI_Bcast(locx, nelem_locx, MPI_INT, root, MPI_COMM_WORLD);
}
```

As a consequence of our coding error, MPI_BCAST() is not being called by all of the processes in our communicator but, instead, is being called only by the root processes, which is our rank 0 process.

To correct our coding error we need to move our broadcast call outside of this if statement so that MPI_BCAST() will be called by all processes as required in MPI.

```
if (procId == root)
{
  locx = gblx;
}

/* Broadcast local array 'locx' to all processes */
  MPI_Bcast(locx, nelem_locx, MPI_INT, root, MPI_COMM_WORLD);
```

When we make this correction and then recompile and rerun our code, we get the following correct output for our computation.

```
$ mpicc -g -o broadcast broadcast.c

$ mpirun -np 4 broadcast
```

The elements of the global matrix *gblA* on the root process are

```
 1   2   3   4   5   6
 4   5   6   7   8   9
 7   8   9  10  11  12
10  11  12  13  14  15
```

The elements of the global array *gblx* are: [1 2 3 4 5 6]

The elements of *gblB* on the root process is: [91 154 217 280]

3.11.4 Debugging Exercises

Exercise 1

The code shown below, broadcast.c, is a C program for computing the elements of the vector B for the equation

A x = B

where A is an *n x m* matrix, *x* is an *m x 1* column vector, and B is an *n x 1* column vector.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

int main( int argc, char *argv[])
{
  int root;
  int nelem, ndim, ncount;
  int nproc, procId, index, ndx, procIndex, source, target, tag;
  int rowIndex, colIndex;
  int **gblA, *gblx, *gblB;
  int *locA, *locx, *locB;
  int nrows_gblA, ncols_gblA;
  int nelem_locA, nelem_gblx, nelem_gblB;
  int nelem_locx, nelem_locB;

  MPI_Status status;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&procId);

/* Define dimensions of global arrays for Ax=B */
  nrows_gblA = 4, ncols_gblA = 6;
  nelem_gblx = ncols_gblA;
```

```
  nelem_gblB = nrows_gblA;

/* Define dimensions of local arrays for Ax=B */
  nelem_locA = ncols_gblA;
  nelem_locx = ncols_gblA;
  nelem_locB = nrows_gblA;

/* Define rank 0 process as root */
  root = 0;

/* Dynamically allocate memory for the global arrays on root process */
  if (procId == root)
  {
    /* Construct 'gblA' */
    gblA = (int**)malloc(nrows_gblA * sizeof(int));

    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      gblA[rowIndex] = (int*)malloc(ncols_gblA * sizeof(int));
    }

  /* Initialize 'gblA' */
    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      for (colIndex = 0; colIndex < ncols_gblA; colIndex++)
      {
        gblA[rowIndex][colIndex] = (((nrows_gblA - 1) * rowIndex) + colIndex) + 1;
      }
    }

  /* Construct 'gblx' and 'gblB' */
    gblx = malloc(nelem_gblx * sizeof(int));
    gblB = malloc(nelem_gblB * sizeof(int));

  /* Initialize 'gblx' and 'gblB' */
    for (index = 0; index < nelem_gblx; index++)
    {
      gblx[index] = index + 1;
    }

    for (index = 0; index < nelem_gblB; index++)
    {
      gblB[index] = 0;
    }
  }

/* Dynamically allocate memory for all local arrays on all processes */
  locA = malloc(nelem_locA * sizeof(int));
  locx = malloc(nelem_locx * sizeof(int));
  locB = malloc(nelem_locB * sizeof(int));

  for (index = 0; index < nelem_locA; index++)
  {
    locA[index] = 0;
  }

  for (index = 0; index < nelem_locx; index++)
  {
    locx[index] = 0;
  }

  for (index = 0; index < nelem_locB; index++)
  {
    locB[index] = 0;
  }

/* Print the global array 'gblA' on the root process */
  if (procId == root)
  {
```

```
    printf("The elements of the global array 'gblA' are:\n");

    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      for (colIndex = 0; colIndex < ncols_gblA; colIndex++)
      {
        printf("  %d", gblA[rowIndex][colIndex]);
      }
      printf("\n");
    }
      printf("\n\n");
  }

/* Print the global vector 'gblx' on the root process */
  if (procId == root)
  {
    printf("The elements of the global vector 'gblx' are: [ ");
    for (index = 0; index < nelem_gblx; index++)
    {
      printf("%d ", gblx[index]);
    }
    printf("]\n\n");
  }

/* Define source as root process */
  source = root;

  for (procIndex = 0; procIndex < nproc; procIndex++)
  {
    target = procIndex;
    tag = target;

    if (target != source)
    {
      if (procId == source)
      {
        MPI_Send(&gblA[target][0], nelem_locA, MPI_INT, target, tag, MPI_COMM_WORLD);
      }
      else if (procId == target)
      {
        MPI_Recv(locA, nelem_locA, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
      }
    }
    else if (procId == source)
    {
      locA = &gblA[target][0];
    }
  }

/* Broadcast local array 'gblx' to all processes */
  MPI_Bcast(gblx, nelem_gblx, MPI_INT, root, MPI_COMM_WORLD);

/* Carry out local matrix multiplication, Ax=B, on each process */
  for (index = 0; index < nelem_gblx; index++)
  {
    locB[procId] += locA[index] * gblx[index];
  }


  nelem = 1;

/* Sum elements of 'locB' on all processes and store results on root process */
    for (index = 0; index < nelem_locB; index++)
    {
      MPI_Reduce(&locB[index], &gblB[index], nelem, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
    }

/* Print elements of 'gblB' on root process */
  if (procId == root)
```

```
  {
    printf("The elements of 'gblB' on the root process is: [ ");
    for (index = 0; index < nelem_gblB; index++)
    {
      printf("%d ", gblB[index]);
    }
    printf("]\n\n");
  }

  MPI_Finalize();

  return 0;
}
```

When we compile this code and run it on four processes, we get the following output.

```
$ mpicc -g -o broadcast broadcast.c

$ mpirun -np 4 broadcast
```

The elements of the global array *gblA* are:

```
  1    2    3    4    5    6
  4    5    6    7    8    9
  7    8    9   10   11   12
 10   11   12   13   14   15
```

The elements of the global array *gblx* are: [1 2 3 4 5 6]

```
p1_25656:  p4_error: interrupt SIGSEGV: 11
bm_list_6926: (4.981998) wakeup_slave: unable to interrupt slave 0 pid 6925
p3_14453:  p4_error: net_recv read:  probable EOF on socket: 1
p2_11141:  p4_error: interrupt SIGSEGV: 11
bm_list_6926: (4.982444) wakeup_slave: unable to interrupt slave 0 pid 6925
$ Broken pipe
Broken pipe
Broken pipe
```

Clearly, we have a coding error somewhere in our program. To identify this error, debug the code using TotalView®.

```
$ mpirun -dbg=totalview -np 4 broadcast
```

Exercise 2

The code shown below, reduce.c, is a C program for computing the elements of the vector B for the equation

A x = B

where A is an *n x m* matrix, *x* is an *m x 1* column vector, and B is an *n x 1* column vector.

```
#include <stdio.h>
#include <stdlib.hh>
#include <math.hh>
#include "mpi.h"

int main( int argc, char *argv[])
{
  int nproc, procId, root;
  int procIndex, index, rowIndex, colIndex;
  int nelem, nelem_in_gblArray, nelem_in_locArray;
  MPI_Status status;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&procId);

  int **gblA, **locA;
  int *gblx, *gblB, *locx, *locB;
```

```
  int nrows_gblA = 4;
  int ncols_gblA = 6;
  int nterms;

  int nrows_locA = nrows_gblA;
  int ncols_locA = ncols_gblA/nproc;
  int nelem_gblx = ncols_gblA;
  int nelem_locx = ncols_locA;
  int nelem_gblB = nrows_gblA;
  int nelem_locB = nelem_gblB;

  gblB = malloc(nelem_gblB*sizeof(int));
  locB = malloc(nelem_locB*sizeof(int));
  locx = malloc(nelem_locx*sizeof(int));

/* Define proc 0 as root process */
  root = 0;

  if (procId == root)
  {
  /* Construct a 2D array on the root process */
    gblA = (int**)malloc(nrows_gblA * sizeof(int));

    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      gblA[rowIndex] = (int*)malloc(ncols_gblA * sizeof(int));
    }

  /* Initialize the 2D array */
    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      for (colIndex = 0; colIndex < ncols_gblA; colIndex++)
      {
        gblA[rowIndex][colIndex] = rowIndex + colIndex;
      }
    }
  }

/* Print elements of the global array 'gblA' on the root process */
  if (procId == root)
  {
    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      for (colIndex = 0; colIndex < ncols_gblA; colIndex++)
      {
        printf(" %d  ", gblA[rowIndex][colIndex]);
      }
      printf(" \n");
    }
    printf(" \n\n");
  }

  if (procId == root)
  {
    gblx = malloc(nelem_gblx*sizeof(int));
    for (index = 0; index < nelem_gblx; index++)
    {
      gblx[index] = pow((index + 1), 2.0);
    }
  }

/* Print elements of global array 'gblx' on root process */
  if (procId == root)
  {
    printf("The elements of 'gblx' on the root process are: ");
    for (index = 0; index < nelem_gblx; index++)
    {
      printf("%d ", gblx[index]);
    }
```

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

/* Construct a local 2D array to receive scattered data */
  locA = (int**)malloc(nrows_locA * sizeof(int));

  for (rowIndex = 0; rowIndex < nrows_locA; rowIndex++)
  {
    locA[rowIndex] = (int*)malloc(ncols_locA * sizeof(int));
  }

/* Initialize the local arrays 'locA', 'locx', and 'locB'  */
  for (rowIndex = 0; rowIndex < nrows_locA; rowIndex++)
  {
    for (colIndex = 0; colIndex < ncols_locA; colIndex++)
    {
      locA[rowIndex][colIndex] = 0;
    }
  }

  for (index = 0; index < nelem_locx; index++)
  {
    locx[index] = 0;
  }

  for (index = 0; index < nelem_locB; index++)
  {
    locB[index] = 0;
  }

/* Specify number of elements scattered to each separate process */
  nelem = ncols_gblA/nproc;

/* Scatter elements of 'gblA' and 'gblx' across all 'nproc' processes */
  for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
  {
    MPI_Scatter(gblA, nelem, MPI_INT, locA, nelem, MPI_INT, root, MPI_COMM_WORLD);
  }

  MPI_Scatter(gblx, nelem, MPI_INT, locx, nelem, MPI_INT, root, MPI_COMM_WORLD);

/* Carry out local matrix multiplication on all processes */
  for (rowIndex = 0; rowIndex < nrows_locA; rowIndex++)
  {
    for (colIndex = 0; colIndex < ncols_locA; colIndex++)
    {
      locB[rowIndex] += locA[rowIndex][colIndex] * locx[colIndex];
    }
  }

/* Specify number of terms to be included in the sum of elements of 'locB' on each process */
  nterms = 1;

  MPI_Reduce(locB, gblB, nterms, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);

/* Print elements of the global array B=Ax on the root process */
  if (procId == root)
  {
    printf("The transpose of the global array B=Ax on the root process is: [ ");
    for (index = 0; index < nelem_gblB; index++)
    {
      printf("%d ", gblB[index]);
    }
    printf("]\n\n");
  }

  MPI_Finalize();

  return 0;
```

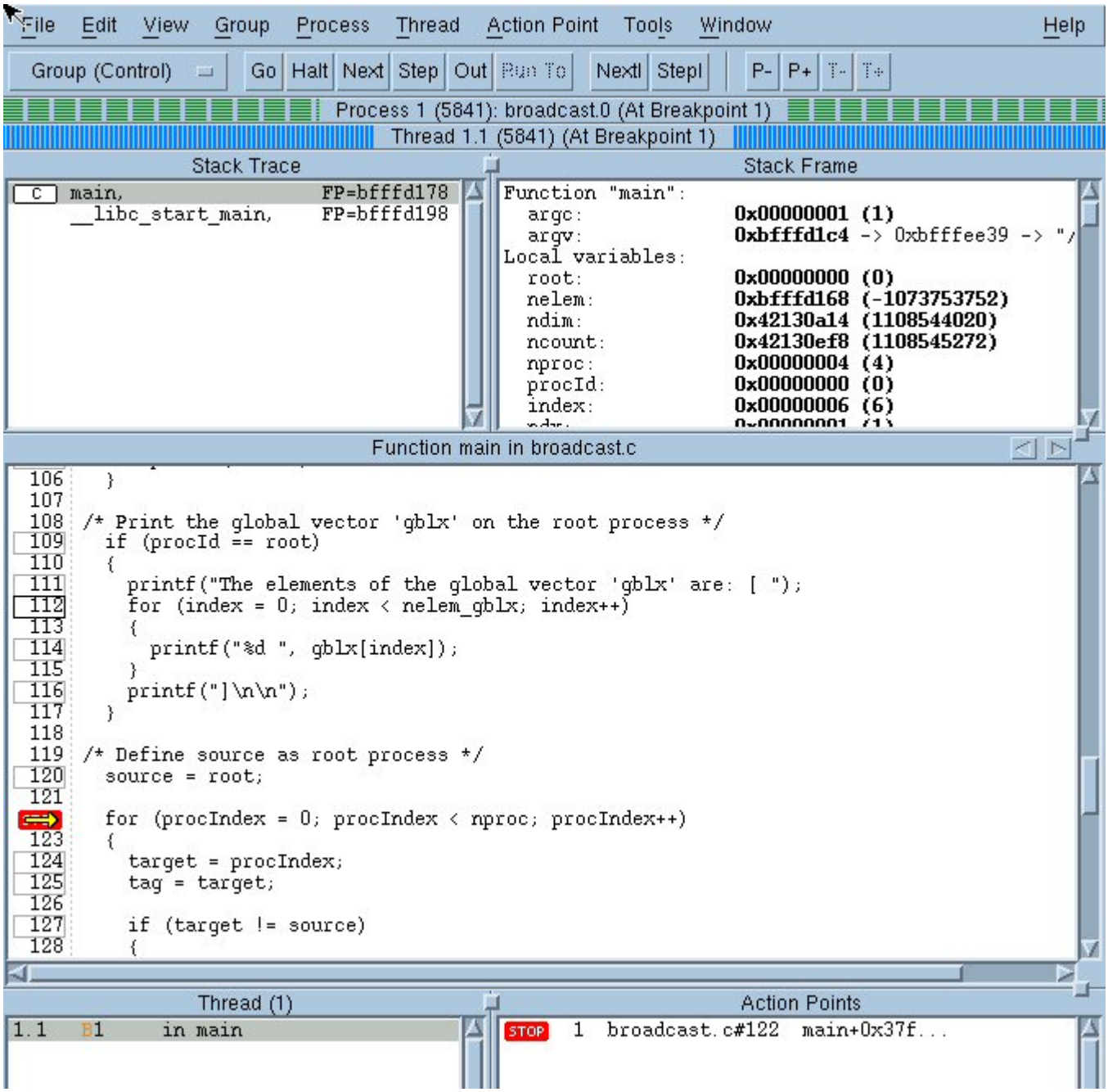
}

Run this code in parallel on three processes and examine the output to see if the results of this computation suggest that there may be one or more errors in this code. If the output does suggest the presence of a coding error, debug this program using TotalView® or some alternative parallel debugger of your choice.

3.11.5 Solutions to Exercises

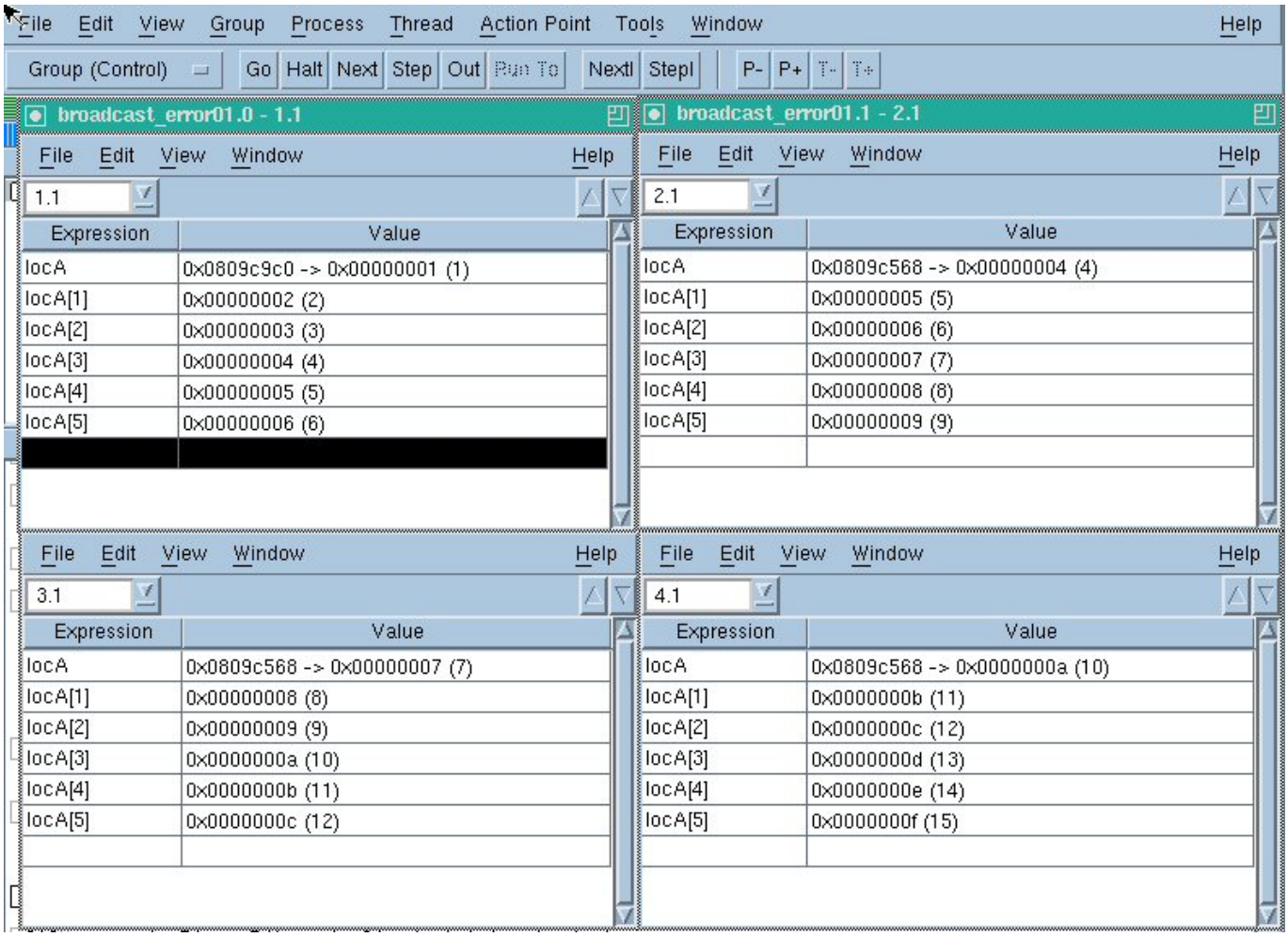
Exercise 1:

Obviously, our error occurs after the section of code that prints out the elements of the global vector *gblx* on the root process. Therefore, we set our first breakpoint somewhere below this section of code. As illustrated in the figure below, the code fragment that prints *gblx* ends on line 117, so we set our breakpoint at line 122 and start our run in TotalView®. Line 122 is the entry point into a for loop that executes a set of *nproc* point-to-point communication events that distributes the global array *gblA* across our four processes.

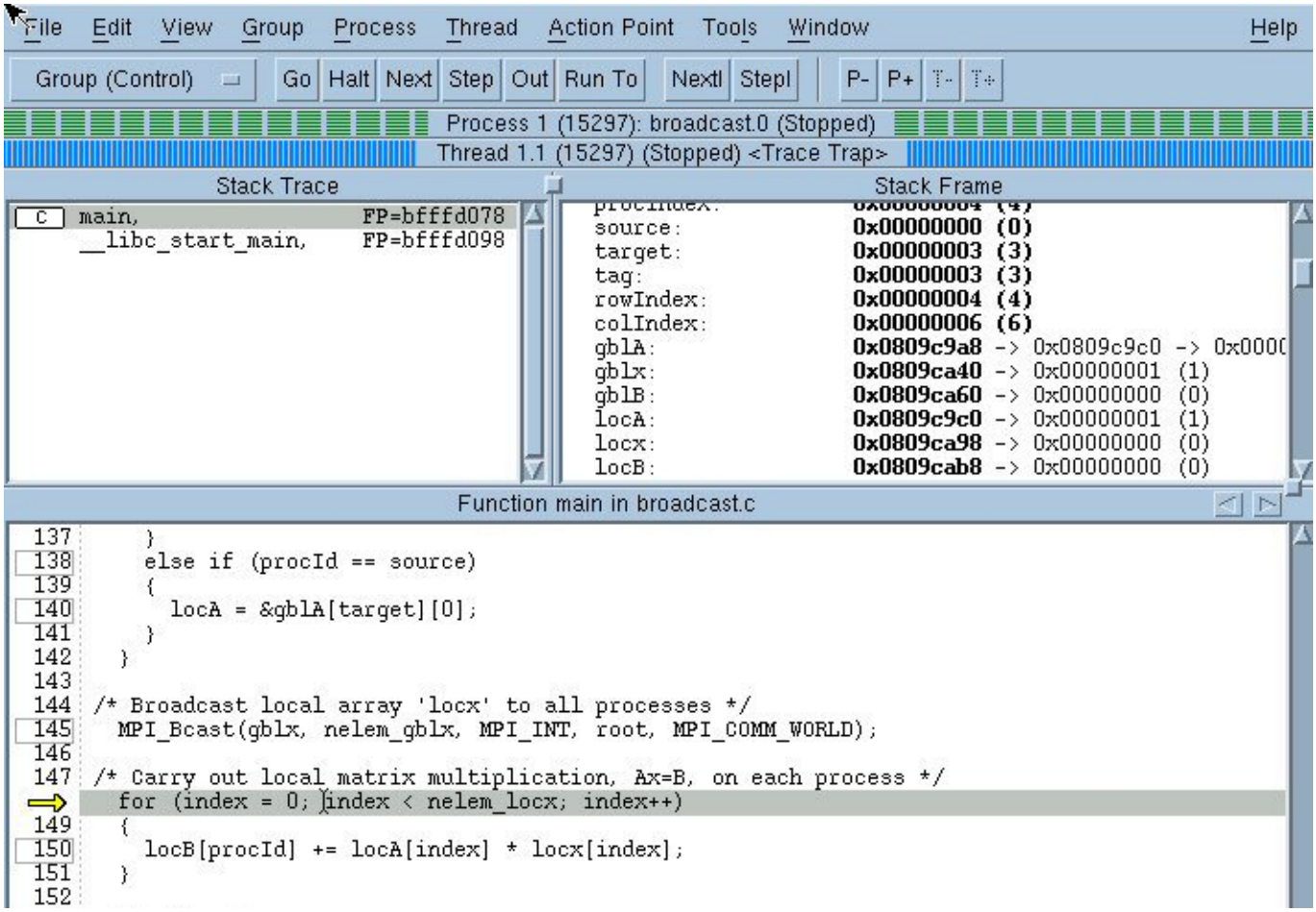


To see if our error is located in this part of our code, we traverse this for loop *nproc* times and then examine the values of the elements of the local array *locA* on each of the four processes. After exiting this loop we see that the values of the

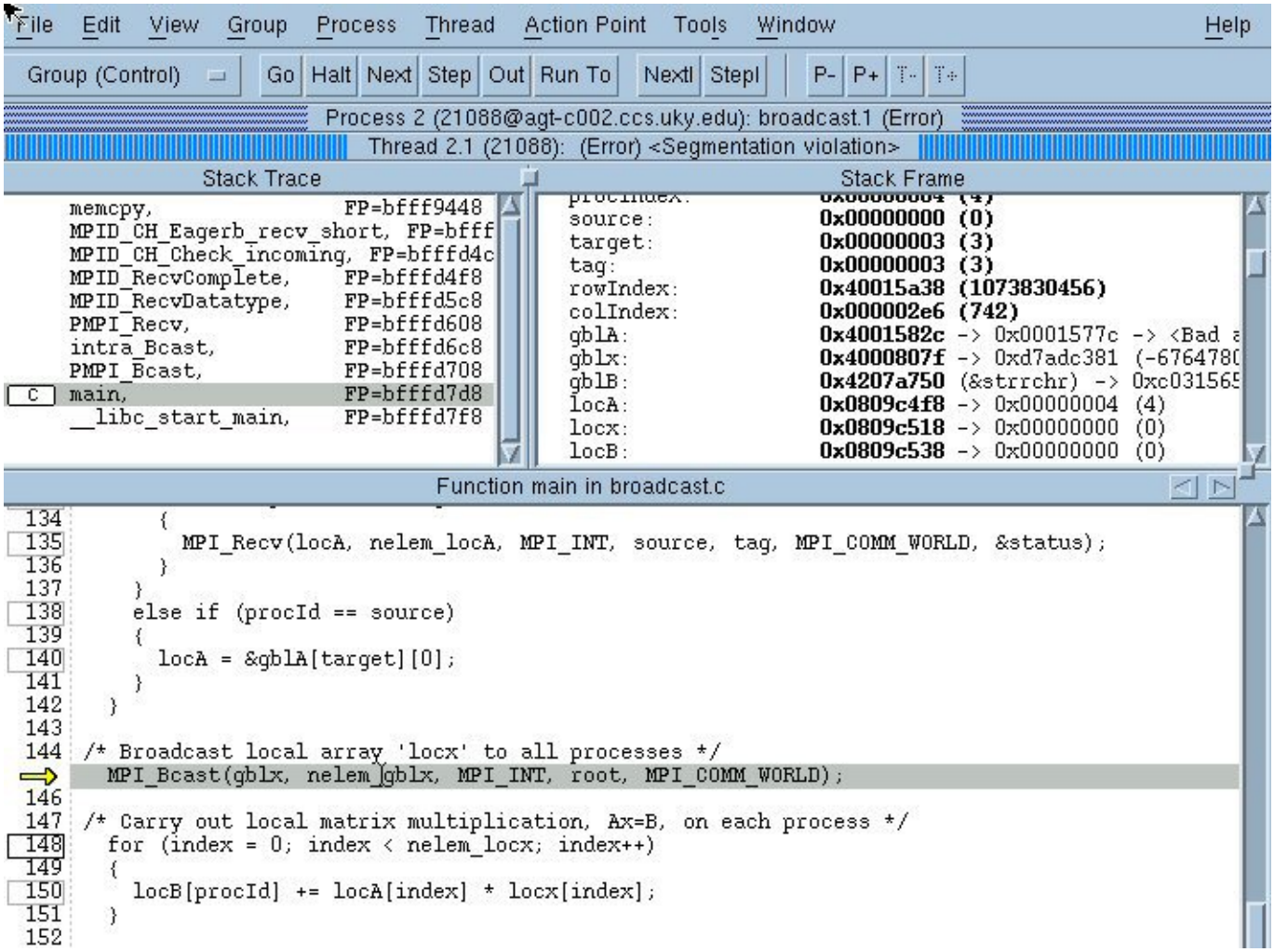
elements of *locA* are all correct on each of our four processes as shown in the following figure. Note that these results are identical to those from the example program in this lesson.



Since there are no errors in the coding of our point-to-point communications calls, we continue to move through our code to see if we can find anything that looks unusual. After exiting the for loop, we immediately move to a call to broadcast the global vector *gblx* on line 145. However, when we attempt to continue beyond this call to MPI_BCAST(), we immediately see something that does not look right. On Process 1 (our rank 0 or root process) the pointer in the Function pane of the Process Window moves to line 148 as shown in the following figure.



However, in the Function pane of the same window on our other three processes (Processes 2, 3, and 4) the pointer does not move beyond line 145 as shown for Process 2 (our rank 1 process) in the figure below.

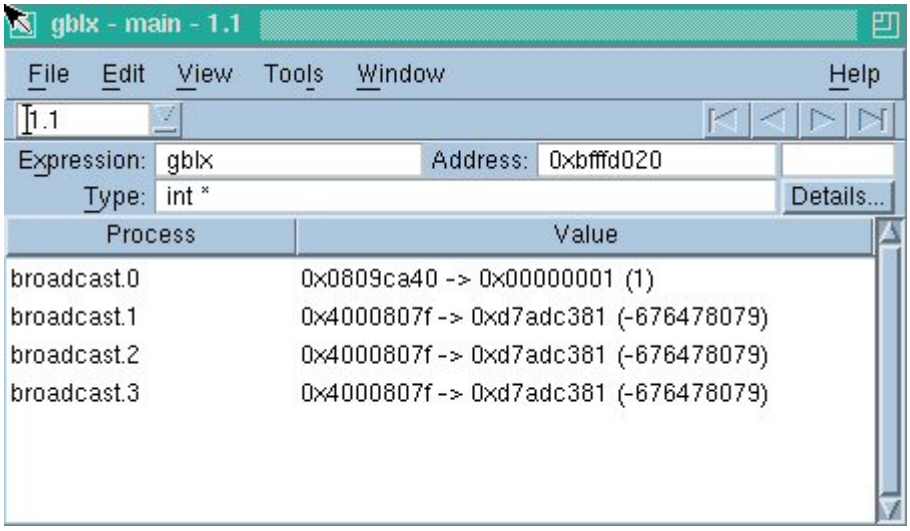


In addition the line above the Stack Trace and Stack Frame panes in this Process Window states that there is an error in the broadcast event on Process 2.

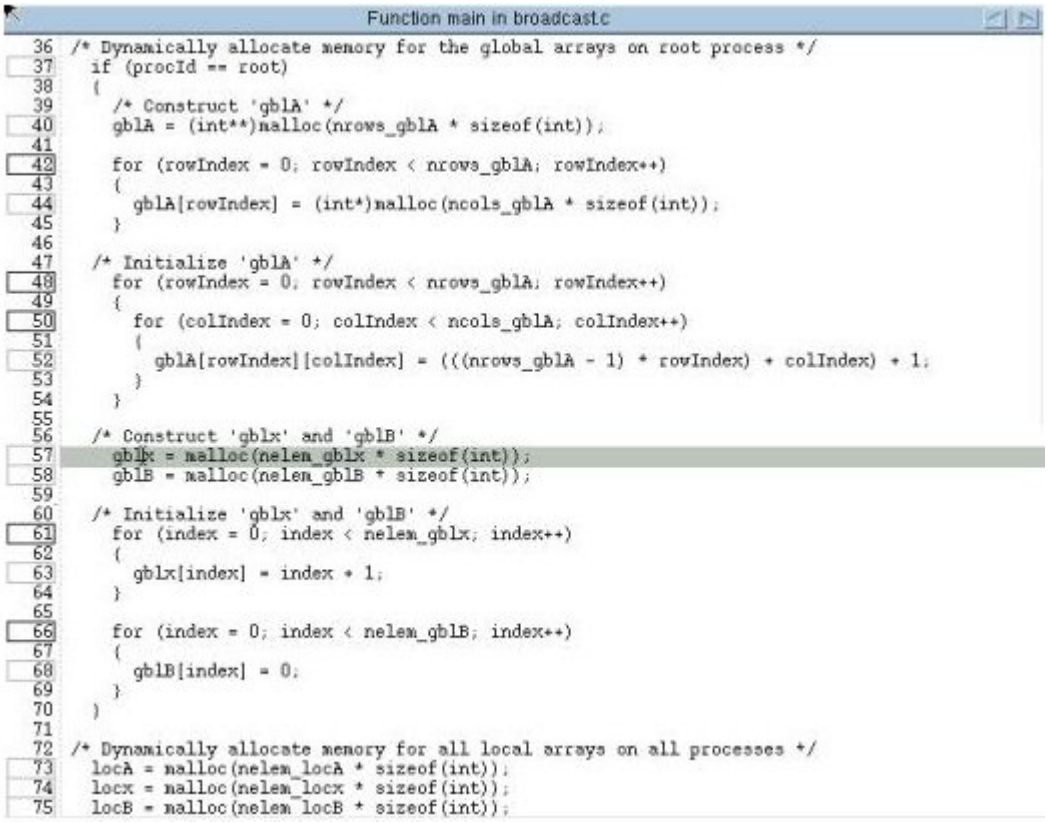
Consequently, our error appears to be in the collective communication call on line 145 of our code:

```
MPI_Bcast(gblx, nelem_gblx, MPI_INT, root, MPI_COMM_WORLD);
```

To identify our error we examine the values for each of the arguments passed to MPI_BCAST() to see if anything looks wrong. When we examine the value of the first argument passed to this function, *gblx*, we can immediately identify the problem. As shown in the following figure, *gblx* is defined on proc 0 but is undefined on our other three processes.



If we look back through our code, we see exactly why this is the case. In lines 37 - 70 of our code we are dynamically allocating memory for the global array *gblA* (lines 40 - 45), *gblx* (line 57), and *gblB* (line 58) only on the root process (proc 0) as illustrated in the figure below.



In order to correct our coding error, we must allocate the memory for *gblx* outside of the ifstatement on line 37. That is, we need to change our code to read as follows:

```
gblx = malloc(nelem_gblx * sizeof(int));

if (procId == root)
  {
    /* Construct 'gblA' */
    gblA = (int**)malloc(nrows_gblA * sizeof(int));

    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      gblA[rowIndex] = (int*)malloc(ncols_gblA * sizeof(int));
```

```
    }

    .  .  .

  /* Construct 'gblB' */
    gblB = malloc(nelem_gblB * sizeof(int));

  /* Initialize 'gblx' and 'gblB' */
    for (index = 0; index < nelem_gblx; index++)
    {
      gblx[index] = index + 1;
    }

    for (index = 0; index < nelem_gblB; index++)
    {
      gblB[index] = 0;
    }
  }
```

By allocating memory for *gblx* outside of the if statement on line 37, we are allocating memory for this vector on all processes in our communicator.

There also is a second way to correct this code. We could allocate memory for *gblx* on the root process only and broadcast a local copy of *gblx* to each process in our communicator. That is, copy *gblx* to *locx* on the root process and then call

```
MPI_BCAST(locx, nelem_gblx, MPI_INT, root, MPI_COMM_WORLD);
```

on all processes in MPI_COMM_WORLD. However, we would then need to modify our expression for *locB* on line 150 to read

```
locB[procId] += locA[index] * locx[index];
```

As you probably already realize, this would simply reproduce our code from the example problem, broadcast.c, in this lesson.

The C code shown below uses the first correction presented above.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

int main( int argc, char *argv[])
{
  int root;
  int nelem, ndim, ncount;
  int nproc, procId, index, ndx, procIndex, source, target, tag;
  int rowIndex, colIndex;
  int **gblA, *gblx, *gblB;
  int *locA, *locx, *locB;
  int nrows_gblA, ncols_gblA;
  int nelem_locA, nelem_gblx, nelem_gblB;
  int nelem_locx, nelem_locB;

  MPI_Status status;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&procId);

/* Define dimensions of global arrays for Ax=B */
  nrows_gblA = 4, ncols_gblA = 6;
  nelem_gblx = ncols_gblA;
  nelem_gblB = nrows_gblA;

/* Define dimensions of local arrays for Ax=B */
  nelem_locA = ncols_gblA;
  nelem_locx = ncols_gblA;
  nelem_locB = nrows_gblA;
```

```
/* Define rank 0 process as root */
  root = 0;

/* Allocate memory for the vector 'gblx' on all processes */
  gblx = malloc(nelem_gblx * sizeof(int));

/* Dynamically allocate memory for the global arrays on root process */
  if (procId == root)
  {
    /* Construct 'gblA' */
    gblA = (int**)malloc(nrows_gblA * sizeof(int));

    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      gblA[rowIndex] = (int*)malloc(ncols_gblA * sizeof(int));
    }

  /* Initialize 'gblA' */
    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      for (colIndex = 0; colIndex < ncols_gblA; colIndex++)
      {
        gblA[rowIndex][colIndex] = (((nrows_gblA - 1) * rowIndex) + colIndex) + 1;
      }
    }

  /* Allocate memory for 'gblB' */
    gblB = malloc(nelem_gblB * sizeof(int));

  /* Initialize 'gblx' and 'gblB' */
    for (index = 0; index < nelem_gblx; index++)
    {
      gblx[index] = index + 1;
    }

    for (index = 0; index < nelem_gblB; index++)
    {
      gblB[index] = 0;
    }
  }

/* Dynamically allocate memory for all local arrays on all processes */
  locA = malloc(nelem_locA * sizeof(int));
  locx = malloc(nelem_locx * sizeof(int));
  locB = malloc(nelem_locB * sizeof(int));

  for (index = 0; index < nelem_locA; index++)
  {
    locA[index] = 0;
  }

  for (index = 0; index < nelem_locx; index++)
  {
    locx[index] = 0;
  }

  for (index = 0; index < nelem_locB; index++)
  {
    locB[index] = 0;
  }

/* Print the global array 'gblA' on the root process */
  if (procId == root)
  {
    printf("The elements of the global array 'gblA' are:\n");

    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      for (colIndex = 0; colIndex < ncols_gblA; colIndex++)
```

```
      {
        printf("  %d", gblA[rowIndex][colIndex]);
      }
      printf("\n");
    }
    printf("\n");
  }

/* Print the global vector 'gblx' on the root process */
  if (procId == root)
  {
    printf("The elements of the global vector 'gblx' are: [ ");
    for (index = 0; index < nelem_gblx; index++)
    {
      printf("%d ", gblx[index]);
    }
    printf("]\n\n");
  }

/* Define source as root process */
  source = root;

  for (procIndex = 0; procIndex < nproc; procIndex++)
  {
    target = procIndex;
    tag = target;

    if (target != source)
    {
      if (procId == source)
      {
        MPI_Send(&gblA[target][0], nelem_locA, MPI_INT, target, tag, MPI_COMM_WORLD);
      }
      else if (procId == target)
      {
        MPI_Recv(locA, nelem_locA, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
      }
    }
    else if (procId == source)
    {
      locA = &gblA[target][0];
    }
  }

  /* Broadcast local array 'gblx' to all processes */
    MPI_Bcast(gblx, nelem_locx, MPI_INT, root, MPI_COMM_WORLD);

  /* Carry out local matrix multiplication, Ax=B, on each process */
    for (index = 0; index < nelem_gblx; index++)
    {
      locB[procId] += locA[index] * gblx[index];
    }

  /* Specify number of elements to be included in the summation on each process */
    ncount = 1;

  /* Sum elements of 'locB' on all processes and store results on root process */
    for (index = 0; index < nelem_locB; index++)
    {
      MPI_Reduce(&locB[index], &gblB[index], ncount, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
    }

  /* Print elements of 'gblB' on root process */
    if (procId == root)
    {
      printf("The elements of 'gblB' on the root process is: [ ");
      for (index = 0; index < nelem_gblB; index++)
      {
        printf("%d ", gblB[index]);
```

```
      }
      printf("]\n\n");
    }

  MPI_Finalize();

  return 0;
}
```

When we recompile and run this modified code on four processes, we get the following correct output:

```
$ mpicc -g -o broadcast broadcast.c

$ mpirun -np 4 broadcast
```

The elements of the global array *gblA* are:

```
 1    2    3    4    5    6
 4    5    6    7    8    9
 7    8    9   10   11   12
10   11   12   13   14   15
```

The elements of the global vector *gblx* are: [1 2 3 4 5 6].

The elements of *gblB* on the root process is: [91 154 217 280].

Exercise 2:

When we compile and run this code on 3 processes, we obtain the following output:

```
$ mpicc -g -lm -o reduce reduce.c

$ mpirun -np 3 reduce

 0   1   2   3   4   5
 1   2   3   4   5   6
 2   3   4   5   6   7
 3   4   5   6   7   8
```

The elements of *gblx* on the root process are: 1 4 9 16 25 36

The transpose of the global array *B=Ax* on the root process is: [350 0 0 0]

The result for the global vector *gblB* certainly cannot be correct since all of the elements in the last three rows of *gblA* and all of the elements in *gblx* are nonzero and positive. Therefore, we must have at least one coding error in this program.

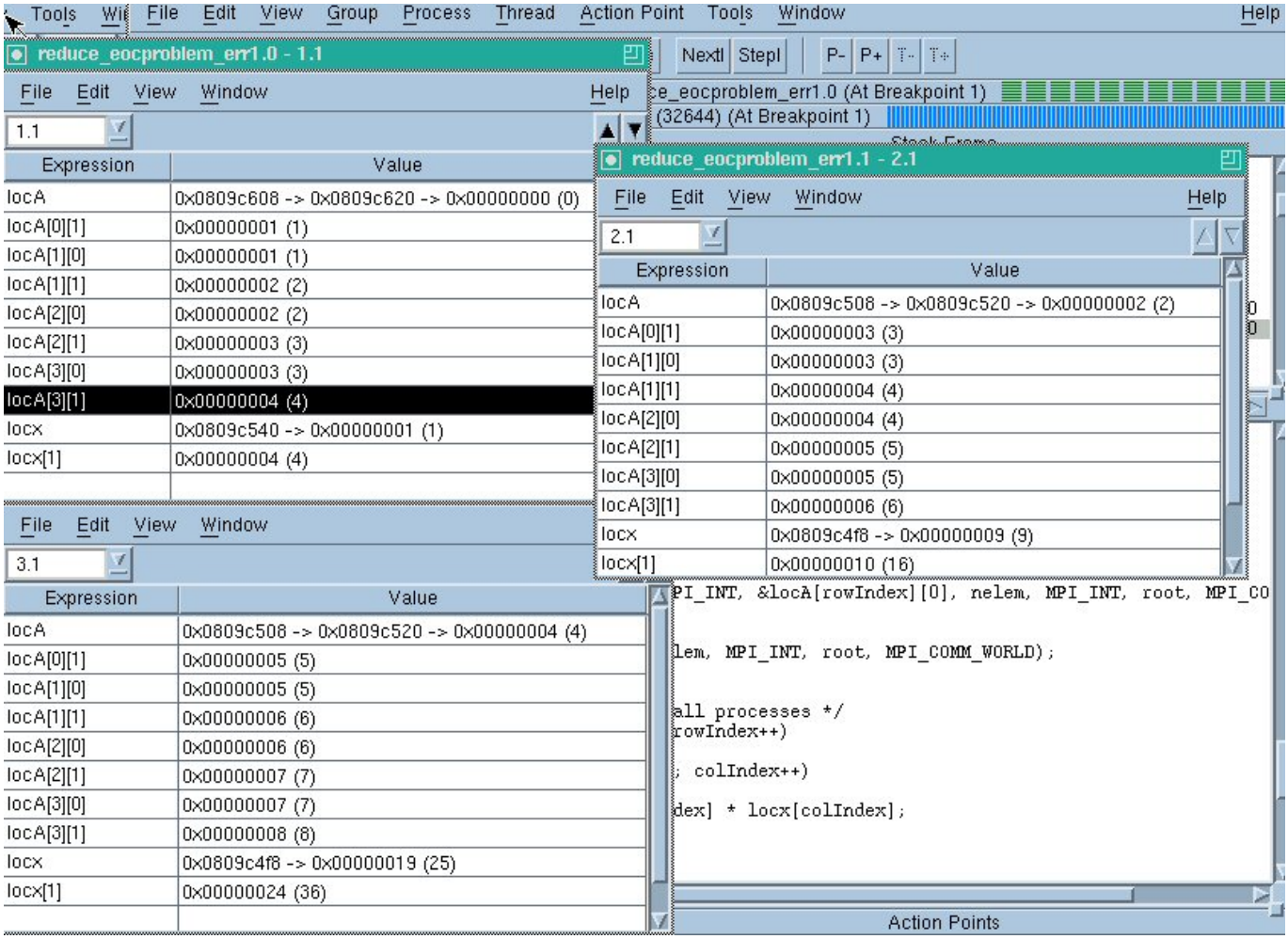
The solution below illustrates the results obtained when reduce.c is debugged in TotalView®.

```
$ mpirun -dbg=totalview -np 3 reduce
```

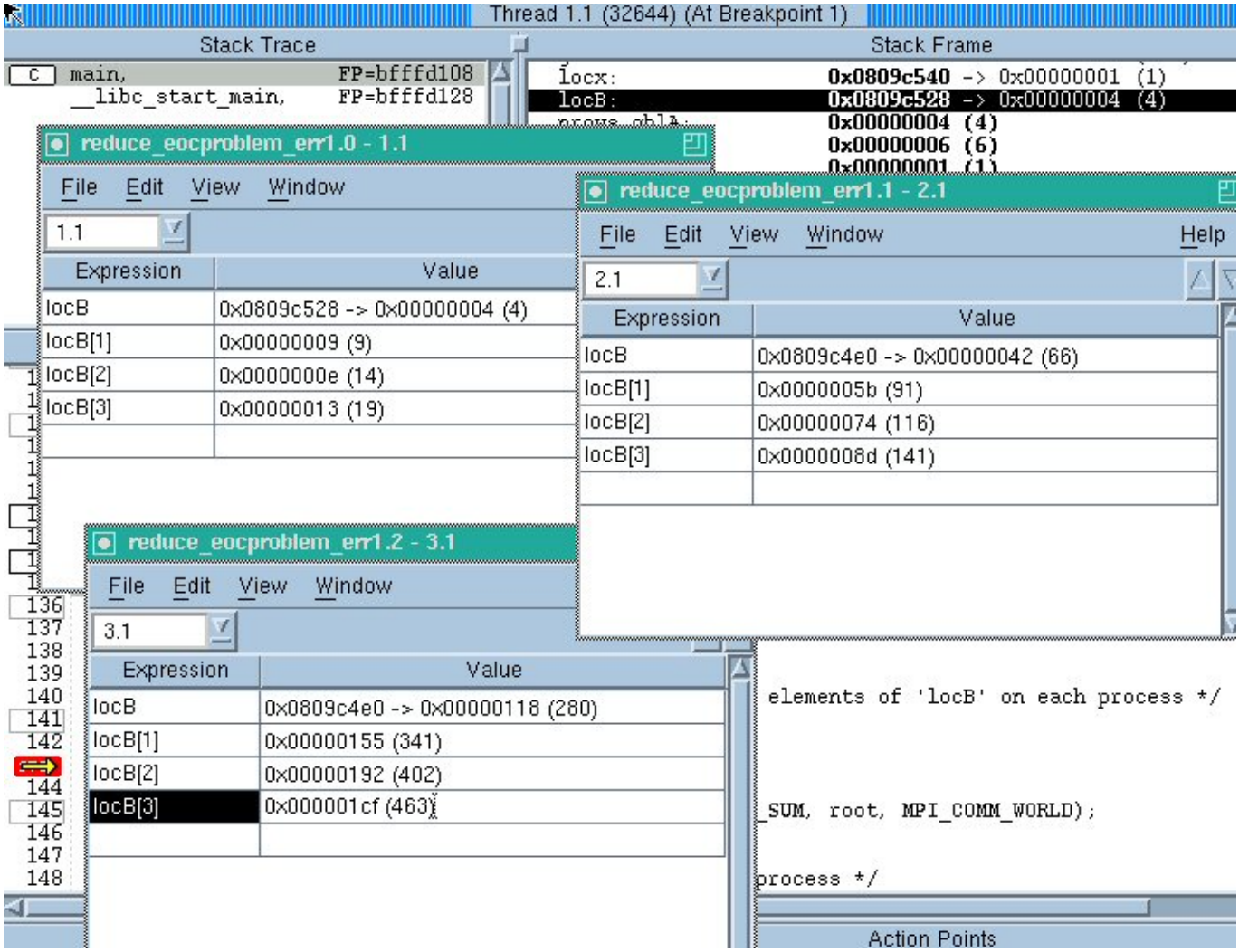
The fact that the element values of *gblA* and *gblx* are correct and that the first three elements in *gblB* are zero suggests that an error(s) could be in any of three separate sections of our code:

- the distribution of the elements of *gblA* and/or *gblx* across processes,
- the local computation of *B=Ax* on each of the four processes, or
- the collective communication call to MPI_REDUCE()

To investigate the first possibility, we set an initial breakpoint on line 132, start our run in TotalView®, and examine the element values of *locA* and *locx* on each of our three processes. As illustrated in the following figure, when we examine these data values we find that the element values for these arrays are correct, which allows us to eliminate the first possibility as the basis of our error.



To investigate the second possibility, we iterate through the two for loops on lines 132 and 134 until we complete the computation of all *nrows_locA* elements of the vector *locB* and then examine the values for each of these elements on each of our three processes. As illustrated in the figure below, every element in *locB* is greater than zero on each process, which also allows us to eliminate the second possibility as the basis of our error.



This indicates that our coding error is in our call to MPI_REDUCE(),

```
MPI_Reduce(locB, gblB, nterms, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
```

Note that we are using the correct predefined reduction operation MPI_SUM. We also see that the sum of the elements of *locB[0]* on our three processes equals the value of the first element in *gblB*, 350, on the root process,

```
gblB[0]  =  locB[0]proc0 + locB[0]proc1 + locB[0]proc2  =  4 + 66 + 280  =  350
```

This suggests that the requested reduction operation is being carried out correctly on the first element in *locB* on each process but not on any of the other three elements of *locB*. That is, the last three elements in each local vector *locB* are being excluded from the reduction operation.

We can immediately figure out what out coding error is if we think about how MPI_REDUCE() carries out a reduction operation. MPI_REDUCE() selects a single element of data on each process, specified by the first argument passed to the function, combines these values using the operation specified by the fifth argument passed to the function, and then stored the result as a single data value on the rank root process.

Note that the first argument in our call to MPI_REDUCE() is *locB*, which is simply the address of the first element in the local vector,

locB = &locB[0]

Consequently, it is only the first element in *locB* that is being included in the reduction operation, which is exactly the result we obtained in the output from our run. In order to include the other three elements of *locB* in the reduction operation, we must pass the address of each of these elements to MPI_REDUCE() in three separate calls to this function.

We can correct our coding error by making a total of four individual calls to MPI_REDUCE(), one call for each value of *locB[j]*, where *j = 0, 1, . . . , nelem_locB* with *nelem_locB = 3*. Therefore, we simply replace our single call to MPI_REDUCE( ) with the following code,

```
for (index = 0; index < nelem_locB; index++)
{
  MPI_Reduce(&locB[index], gblB, nterms, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
}
```

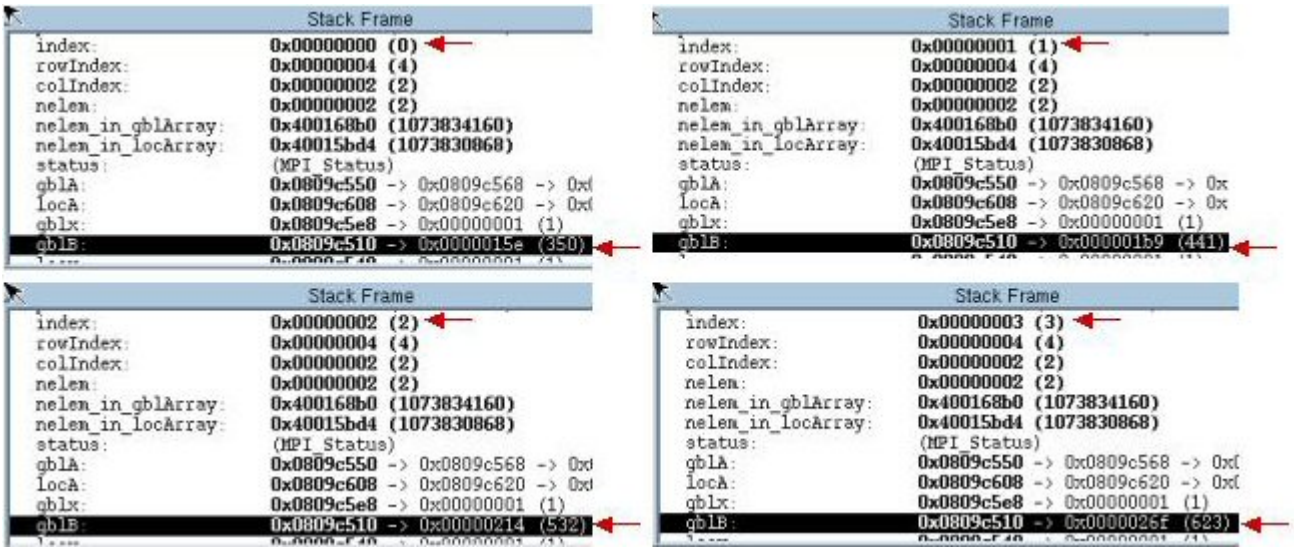
After making this correction and then recompiling and running our corrected code, we obtain the following output.

```
$ mpicc □g □lm □o reduce reduce.c

$ mpirun -np 3 reduce

 0   1   2   3   4   5
 1   2   3   4   5   6
 2   3   4   5   6   7
 3   4   5   6   7   8
```

The elements of *gblx* on the root process are: 1 4 9 16 25 36

The transpose of the global array $B = Ax$ on the root process is: [623 0 0 0]

This result for *gblB* cannot possibly be correct either. Apparently, we failed to correct all of the coding errors in our program. If we set a new breakpoint at line 143 and rerun our code in TotalView®, we can look at the individual element values of *gblB* after each of the four separate calls to MPI_REDUCE(). When we do this, we find the values shown in the following figure.



After looking at how the element values of *gblB* change after each call to MPI_REDUCE(), it is quite obvious what is happening here. During each successive call to MPI_REDUCE(), the values in *locB* on each process are being correctly summed, but the result is consistently being stored in *gblB[0]*. This is certainly not what we want to happen. Instead, we want to sum of the *locB[j]'s* from each process to be stored in the element *gblB[j]* on the root process.

If we again look at the way we have coded our call to MPI_REDUCE(), we see that our receive buffer, the second argument being passed to our function, is *gblB*, which is simply the first element in the global vector,

gblB = &gblB[0]

Consequently, the result of every one of the four reduction operations will be stored in the first element of *gblB*, overwriting any data value previously stored there. This is exactly what we see in the output from our run.

To correct this error, we must specify the second argument in each of our four calls to MPI_REDUCE() as *&gblB[index]* instead of *&gblB[0] = gblB*. This corrected code is shown below.

reduce.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
#include "mpi.h"

int main( int argc, char *argv[])
{
  int nproc, procId, root;
  int procIndex, index, rowIndex, colIndex;
  int nelem, nelem_in_gblArray, nelem_in_locArray;
  MPI_Status status;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&procId);

  int **gblA, **locA;
  int *gblx, *gblB, *locx, *locB;

  int nrows_gblA = 4;
  int ncols_gblA = 6;
  int nterms;

  int nrows_locA = nrows_gblA;
  int ncols_locA = ncols_gblA/nproc;
  int nelem_gblx = ncols_gblA;
  int nelem_locx = ncols_locA;
  int nelem_gblB = nrows_gblA;
  int nelem_locB = nelem_gblB;

  gblB = malloc(nelem_gblB*sizeof(int));
  locB = malloc(nelem_locB*sizeof(int));
  locx = malloc(nelem_locx*sizeof(int));

/* Define proc 0 as root process */
  root = 0;

  if (procId == root)
  {
  /* Construct a 2D array on the root process */
    gblA = (int**)malloc(nrows_gblA * sizeof(int));

    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      gblA[rowIndex] = (int*)malloc(ncols_gblA * sizeof(int));
    }

  /* Initialize the 2D array */
    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      for (colIndex = 0; colIndex < ncols_gblA; colIndex++)
      {
        gblA[rowIndex][colIndex] = rowIndex + colIndex;
      }
    }
  }

/* Print elements of the global array 'gblA' on the root process */
  if (procId == root)
  {
    for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
    {
      for (colIndex = 0; colIndex < ncols_gblA; colIndex++)
      {
        printf(" %d  ", gblA[rowIndex][colIndex]);
      }
      printf(" \n");
    }
    printf(" \n\n");
  }

  if (procId == root)
```

```
  {
    gblx = malloc(nelem_gblx*sizeof(int));
    for (index = 0; index < nelem_gblx; index++)
    {
      gblx[index] = pow((index + 1), 2.0);
    }
  }

/* Print elements of global array 'gblx' on root process */
  if (procId == root)
  {
    printf("The elements of 'gblx' on the root process are: ");
    for (index = 0; index < nelem_gblx; index++)
    {
      printf("%d ", gblx[index]);
    }
    printf("\n\n");
  }

/* Construct a local 2D array to receive scattered data */
  locA = (int**)malloc(nrows_locA * sizeof(int));

  for (rowIndex = 0; rowIndex < nrows_locA; rowIndex++)
  {
    locA[rowIndex] = (int*)malloc(ncols_locA * sizeof(int));
  }

/* Initialize the local arrays 'locA', 'locx', and 'locB'  */
  for (rowIndex = 0; rowIndex < nrows_locA; rowIndex++)
  {
    for (colIndex = 0; colIndex < ncols_locA; colIndex++)
    {
      locA[rowIndex][colIndex] = 0;
    }
  }

  for (index = 0; index < nelem_locx; index++)
  {
    locx[index] = 0;
  }

  for (index = 0; index < nelem_locB; index++)
  {
    locB[index] = 0;
  }

/* Specify number of elements scattered to each separate process */
  nelem = ncols_gblA/nproc;

/* Scatter elements of 'gblA' and 'gblx' across all 'nproc' processes */
  for (rowIndex = 0; rowIndex < nrows_gblA; rowIndex++)
  {
    MPI_Scatter(&gblA[rowIndex][0], nelem, MPI_INT, &locA[rowIndex][0], nelem, MPI_INT, root,
MPI_COMM_WORLD);
  }

  MPI_Scatter(gblx, nelem, MPI_INT, locx, nelem, MPI_INT, root, MPI_COMM_WORLD);


/* Carry out local matrix multiplication on all processes */
  for (rowIndex = 0; rowIndex < nrows_locA; rowIndex++)
  {
    for (colIndex = 0; colIndex < ncols_locA; colIndex++)
    {
      locB[rowIndex] += locA[rowIndex][colIndex] * locx[colIndex];
    }
  }

/* Specify number of terms to be included in the sum of elements of 'locB' on each process */
```

```
  nterms = 1;

  for (index = 0; index < nelem_locB; index++)
  {
    MPI_Reduce(&locB[index], &gblB[index], nterms, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
  }

/* Print elements of the global array B=Ax on the root process */
  if (procId == root)
  {
    printf("The transpose of the global array B=Ax on the root process is: [ ");
    for (index = 0; index < nelem_gblB; index++)
    {
      printf("%d ", gblB[index]);
    }
    printf("]\n\n");
  }

  MPI_Finalize();

  return 0;
}
```

When we recompile and run this code on three processes we get the following correct output.

```
$ mpicc □g □lm □o reduce reduce.c

$ mpirun -np 3 reduce

 0   1   2   3   4   5
 1   2   3   4   5   6
 2   3   4   5   6   7
 3   4   5   6   7   8
```

The elements of *gblx* on the root process are: 1 4 9 16 25 36

The transpose of the global array *B=Ax* on the root process is: [350 441 532 623]

3.12 Debugging OpenMP Scope Errors

3.12.1 Debugging OpenMP Scope Errors

Introduction

OpenMP can be used to parallelize code through the use of multiple threads on shared-memory systems. Shared memory means that a single address space is used for all threads, so threads can potentially write to memory locations simultaneously or in a non-deterministic order. In order to eliminate these occurrences, variables in parallel regions must be declared either *private* or *shared*. When the parallel region is entered, a separate copy of each private variable is created for each thread. The value of the variable can be changed independently on each thread, since each thread only operates on its own copy. The scope of the private copies of these variables is restricted to the parallel region unless directives are included to modify their dispensation upon entering (firstprivate) or leaving (lastprivate) the parallel region. A common OpenMP error is an incorrect declaration, resulting in the variable having the incorrect scope.

Objectives

In this lesson you will learn about scope errors in multi-threaded applications using OpenMP. An example will be given of a commonly-encountered scope error. Some heuristics will be presented to help deduce the nature of the error, and then the SGI CVD debugger will be used to pinpoint it.

3.12.2 Sample Code with an OpenMP Scope Error

The following Fortran code, simjet.f90, calculates the velocity profile of a two-dimensional planar jet of air, given two parameters defining the initial jet conditions. (Details of the physics are not of consequence here.) The output file, simjet.d, contains the parameters defining the jet and the velocity profile. At the end of the file is an edge momentum factor.

```
!-----------------------------------------------------------------
!  Similarity solution for planar, 2-D jet.
!     u....... Axial velocity, m/s
```

```
!     v....... Transverse velocity, m/s
!     x....... Axial coordinate, m  (distance from virtual origin)
!     xi...... Transverse similarity variable
!     xk...... Kinematic axial momentum, m**3/s**2
!     xnu..... Kinematic viscosity, m**2/s
!     y....... Transverse coordinate, m
!-----------------------------------------------------------------

program simjet

  implicit none
  integer, parameter :: nyd = 70
  integer :: j
  real :: xnu = 1.566e-05, x = 1.0, xk = 1.0e-9, &
       q3, cxi, cu, cv, xmom, xi, txi
  real, dimension(nyd) :: u, v, &
       y = (/ &
       0.00000e+00, 5.00169e-02, 1.00060e-01, &
       1.50149e-01, 2.00319e-01, 2.50624e-01, 3.01143e-01, &
       3.51981e-01, 4.03273e-01, 4.55181e-01, 5.07886e-01, &
       5.61579e-01, 6.16479e-01, 6.72834e-01, 7.30891e-01, &
       7.90884e-01, 8.53025e-01, 9.17497e-01, 9.84458e-01, &
       1.05404e+00, 1.12637e+00, 1.20157e+00, 1.27974e+00, &
       1.36101e+00, 1.44551e+00, 1.53335e+00, 1.62468e+00, &
       1.71963e+00, 1.81835e+00, 1.92098e+00, 2.02768e+00, &
       2.13862e+00, 2.25397e+00, 2.37389e+00, 2.49857e+00, &
       2.62820e+00, 2.76298e+00, 2.90311e+00, 3.04881e+00, &
       3.20030e+00, 3.35782e+00, 3.52159e+00, 3.69187e+00, &
       3.86893e+00, 4.05303e+00, 4.24445e+00, 4.44350e+00, &
       4.65047e+00, 4.86567e+00, 5.08945e+00, 5.32215e+00, &
       5.56412e+00, 5.81574e+00, 6.07740e+00, 6.34949e+00, &
       6.63244e+00, 6.92669e+00, 7.23269e+00, 7.55092e+00, &
       7.88187e+00, 8.22605e+00, 8.58400e+00, 8.95628e+00, &
       9.34348e+00, 9.74618e+00, 1.01650e+01, 1.06007e+01, &
       1.10538e+01, 1.15251e+01, 1.20154e+01 /)

  open (21, file = 'simjet.d', status='unknown')

  q3  = 1./3.
  cxi = 0.2752* (xk/(xnu*xnu))**q3 / x**(2./3.)
  cu  = 0.4543* (xk*xk/(xnu*x))**q3
  cv  = 0.5503* (xk*xnu/(x*x))**q3

  do j = 1, nyd
     xi = cxi*y(j)
     txi = tanh(xi)
     xmom = 1.0 - txi**2
     u(j) = cu*xmom
     v(j) = cv*(2.0*xi*xmom - txi)
  enddo

  write (21,20) x, xk, xnu
20 format(/1x,'plane 2-d jet similarity solution' &
         //1x,'distance from virtual origin  ',1pe10.3,' m' &
          /1x,'kinematic axial momentum      ',1pe10.3,' m**3/s**2' &
          /1x,'kinematic viscosity           ',1pe10.3,' m**2/s' &
        /// &
           7x,'y',9x,'xi',8x,'u',11x,'v',10x,'u/um',8x,'v/um'/)

  do j=1,nyd
     write (21,40) y(j),cxi*y(j),u(j),v(j),u(j)/u(1),v(j)/u(1)
40   format(2x,1pe10.3,0pf9.4,4(1pe12.3))

  enddo

  write (21,57) xmom
57 format(/2x,'edge momentum factor = ', 1pe10.3)

  close(21)
```

```
end program simjet
```

After compiling the code and running it serially, the end of the resulting output file was as follows:

```
   1.105E+01   4.8600   4.364E-09  -1.373E-05   2.403E-04  -7.563E-01
   1.153E+01   5.0672   2.883E-09  -1.374E-05   1.588E-04  -7.569E-01
   1.202E+01   5.2827   1.872E-09  -1.375E-05   1.031E-04  -7.573E-01
  edge momentum factor =  1.031E-04
```

An OpenMP directive was then added to parallelize the main loop as follows:

```
  !$omp parallel do private(xi,txi,xmom)

  do j = 1, nyd
     xi = cxi*y(j)
     txi = tanh(xi)
     xmom = 1.0 - txi**2
     u(j) = cu*xmom
     v(j) = cv*(2.0*xi*xmom - txi)
  enddo
```

This code was compiled and run with two threads resulting in the following end of the output file:

```
   1.105E+01   4.8600   4.364E-09  -1.373E-05   2.403E-04  -7.563E-01
   1.153E+01   5.0672   2.883E-09  -1.374E-05   1.588E-04  -7.569E-01
   1.202E+01   5.2827   1.872E-09  -1.375E-05   1.031E-04  -7.573E-01
  edge momentum factor =   0.000E+00
```

The velocity profile (table of numbers) is identical between the serial and parallel cases, but the edge momentum factor differs. Upon examining the entire output files, you would see that the whole file is identical between the two cases with the exception of the edge momentum factor.

3.12.3 Debugging the Sample Code

When serial and parallel results differ in an OpenMP code, it is often useful to apply the following simple heuristic test:

1. Run a serial version of the code (with no OpenMP directives or compiled so it does *not* interpret OpenMP directives)
2. Run the OpenMP version on a single thread.
3. Compare the serial and OpenMP results.
4. If the results are the same, suspect a data dependency.
5. If the results differ, suspect a scope error.

Running the OpenMP version of the code on one processor yields:

```
 1.105E+01   4.8600   4.364E-09  -1.373E-05   2.403E-04  -7.563E-01

 1.153E+01   5.0672   2.883E-09  -1.374E-05   1.588E-04  -7.569E-01

 1.202E+01   5.2827   1.872E-09  -1.375E-05   1.031E-04  -7.573E-01


edge momentum factor =   0.000E+00
```

These results differ from the serial (correct) result. Using our simple heuristic test, we suspect a scope error since the serial run and the single-processor run of the OpenMP version give different results. Now we move on to debugging the problem using CVD.

Debugging the Scope Error using CVD

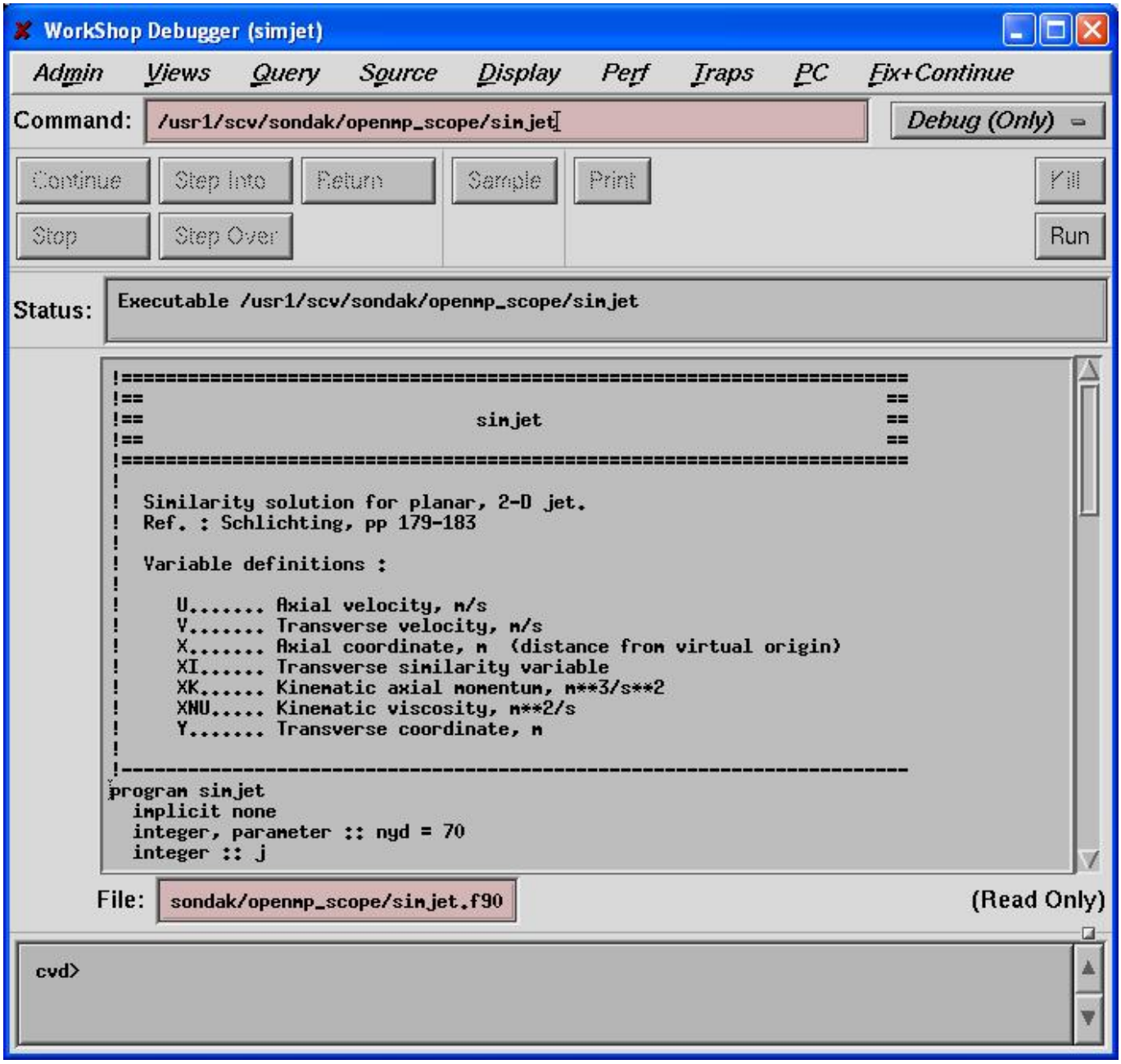
The first step is to re-compile the code with the -g flag, which creates a symbol table. Under IRIX, the compile command is:

```
% f90 -o simjet -mp -g simjet.f90
```

For debugging, we choose to use two threads and start CVD using the commands:

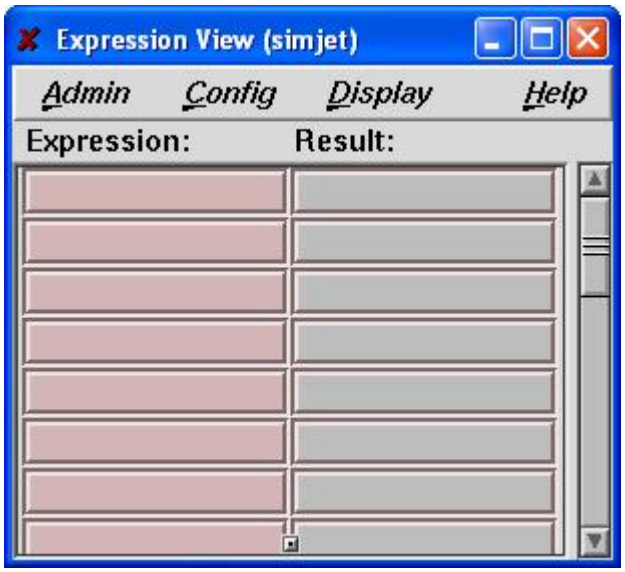
```
% setenv OMP_NUM_THREADS 2

% cvd simjet
```

This spawns the main CVD window:

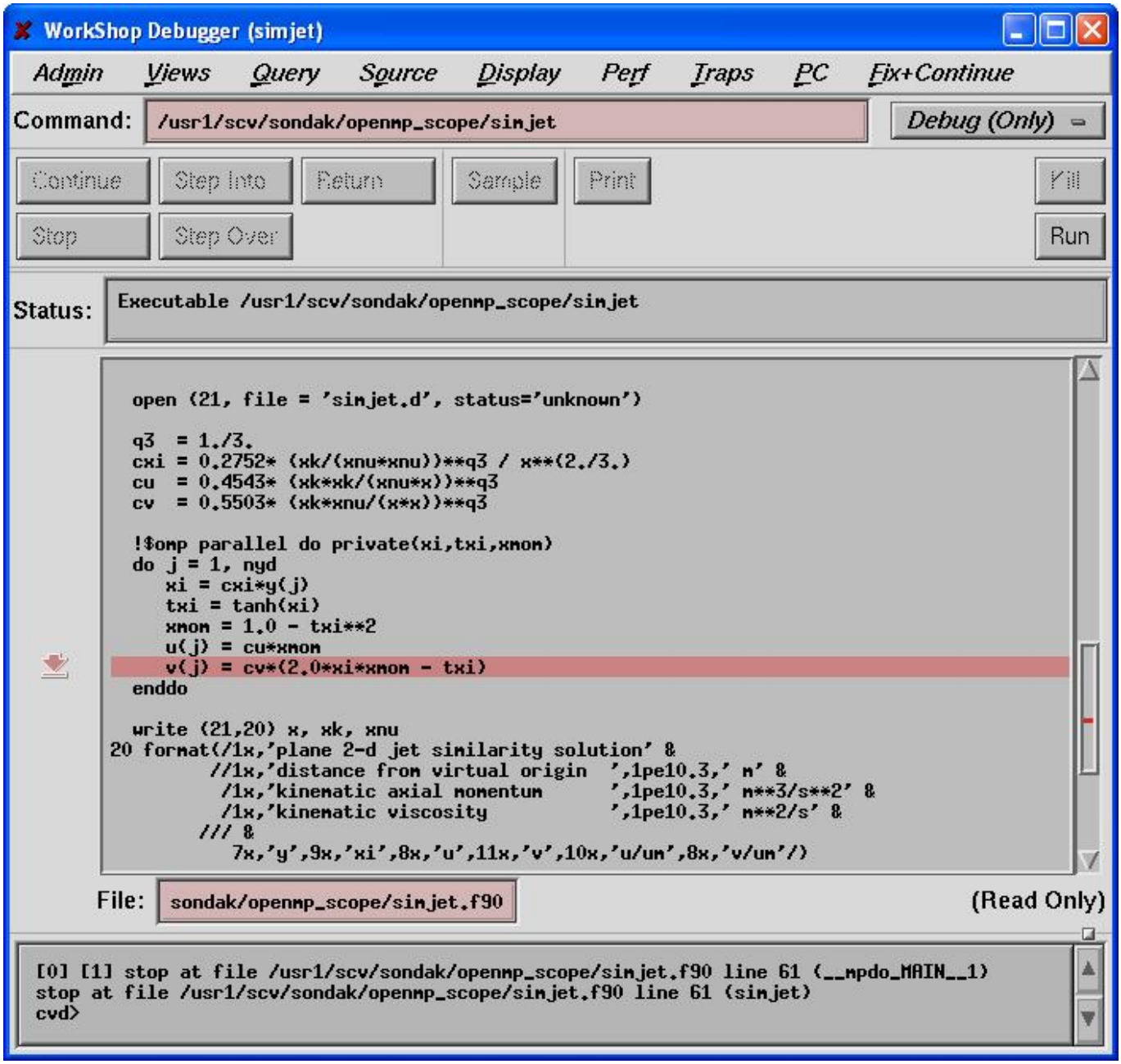


Since we are debugging a multi-threaded code, it is useful to spawn CVD's 'Multiprocess View' window (shown below) by selecting the 'Admin' menu option and then 'Multiprocess View' from the options listed.

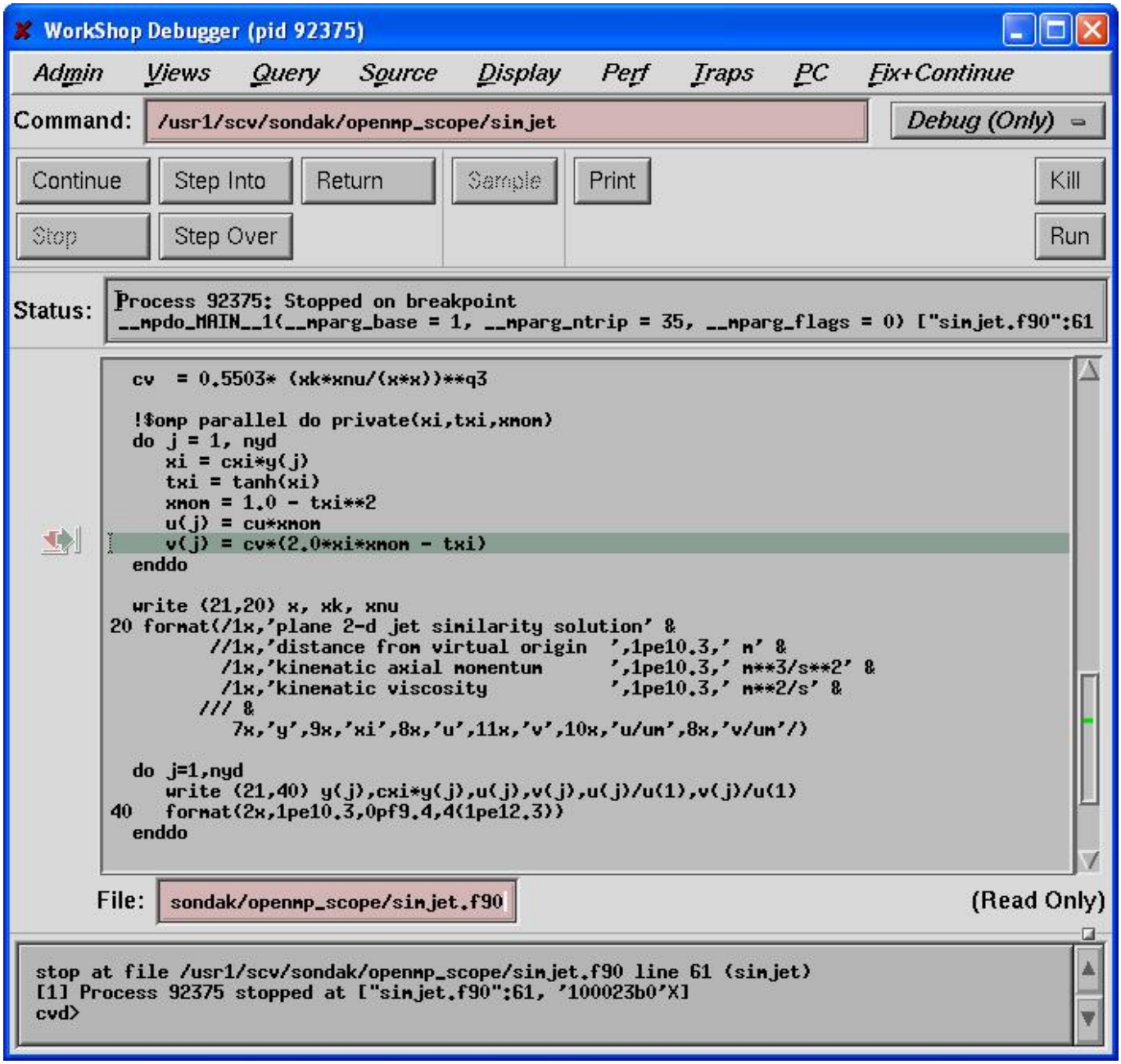
along with the “Expression View” window (shown below) by selecting the 'Views' menu option and then 'Expression View' from the options listed.



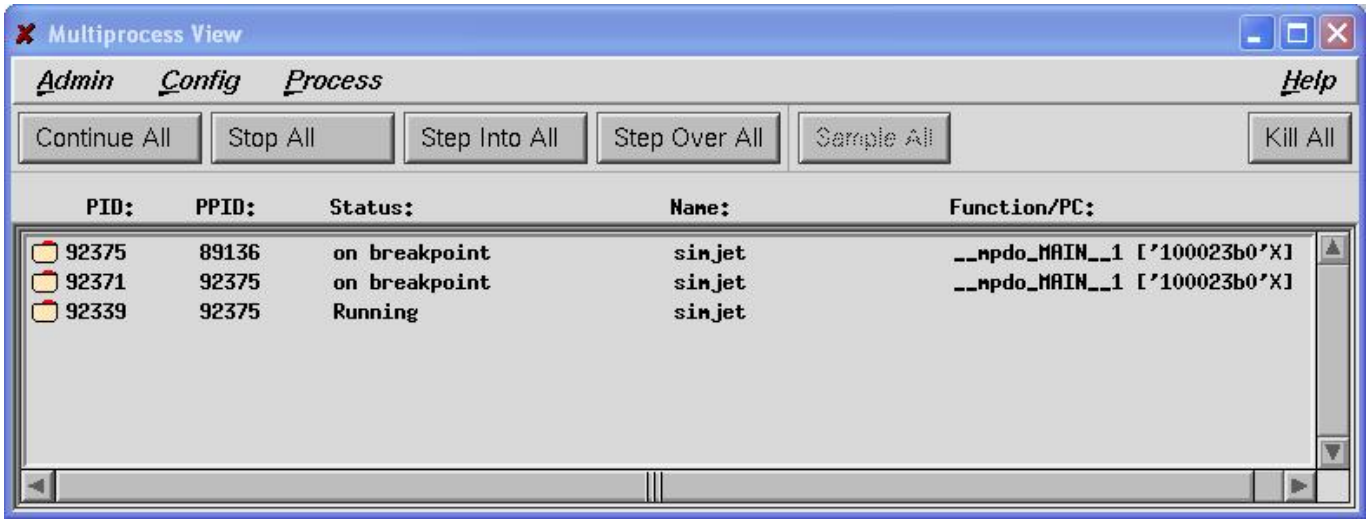
We know that *xmom* is the variable of interest, so we set a breakpoint at the end of the parallel loop after *xmom* has been computed. In CVD, you can do this by clicking the left mouse button in the area to the left of the line of interest.



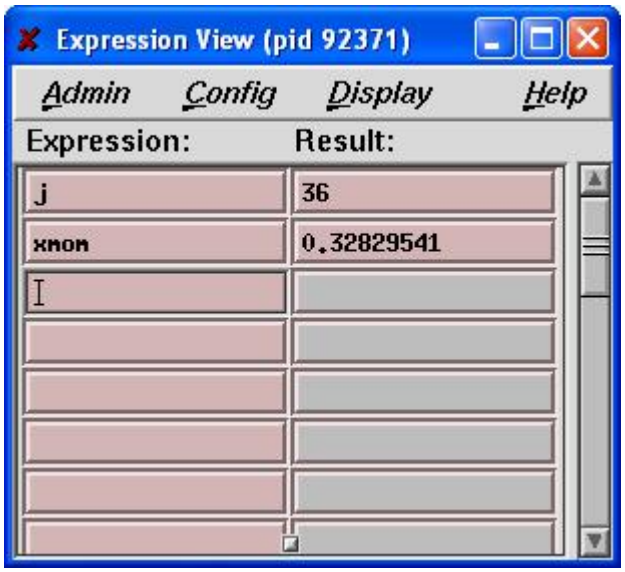
Since we want to monitor *xmom,* we place it in the expression view along with the loop index *j* and advance to the first breakpoint. The following three windows are displayed:


WorkShop Debugger (pid 92375)
Admin Views Query Source Display Perf Traps PC Fix+Continue
Command: /usr1/scv/sondak/openmp_scope/sinjet
Debug (Only)
Continue Step Into Return Sample Print Kill
Stop Step Over Run
Status: Process 92375: Stopped on breakpoint
__mpdo_MAIN__1(__mparg_base = 1, __mparg_ntrip = 35, __mparg_flags = 0) ["sinjet.f90":61
cv = 0.5503* (xk*xnu/(x*x))**q3
!$omp parallel do private(xi,txi,xmom)
do j = 1, nyd
xi = cxi*y(j)
txi = tanh(xi)
xmom = 1.0 - txi**2
u(j) = cu*xmom
v(j) = cv*(2.0*xi*xmom - txi)
enddo
write (21,20) x, xk, xnu
20 format(/1x,'plane 2-d jet similarity solution' &
//1x,'distance from virtual origin ',1pe10.3,' m' &
/1x,'kinematic axial momentum ',1pe10.3,' m**3/s**2' &
/1x,'kinematic viscosity ',1pe10.3,' m**2/s' &
/// &
7x,'y',9x,'xi',8x,'u',11x,'v',10x,'u/um',8x,'v/um'/)
do j=1,nyd
write (21,40) y(j),cxi*y(j),u(j),v(j),u(j)/u(1),v(j)/u(1)
40 format(2x,1pe10.3,0pf9.4,4(1pe12.3))
enddo
File: sondak/openmp_scope/sinjet.f90
(Read Only)
stop at file /usr1/scv/sondak/openmp_scope/sinjet.f90 line 61 (sinjet)
[1] Process 92375 stopped at ["sinjet.f90":61, '100023b0'X]
cvd>


Expression View (pid 92375)
Admin Config Display Help
Expression: Result:
j 1
xmom 1.0



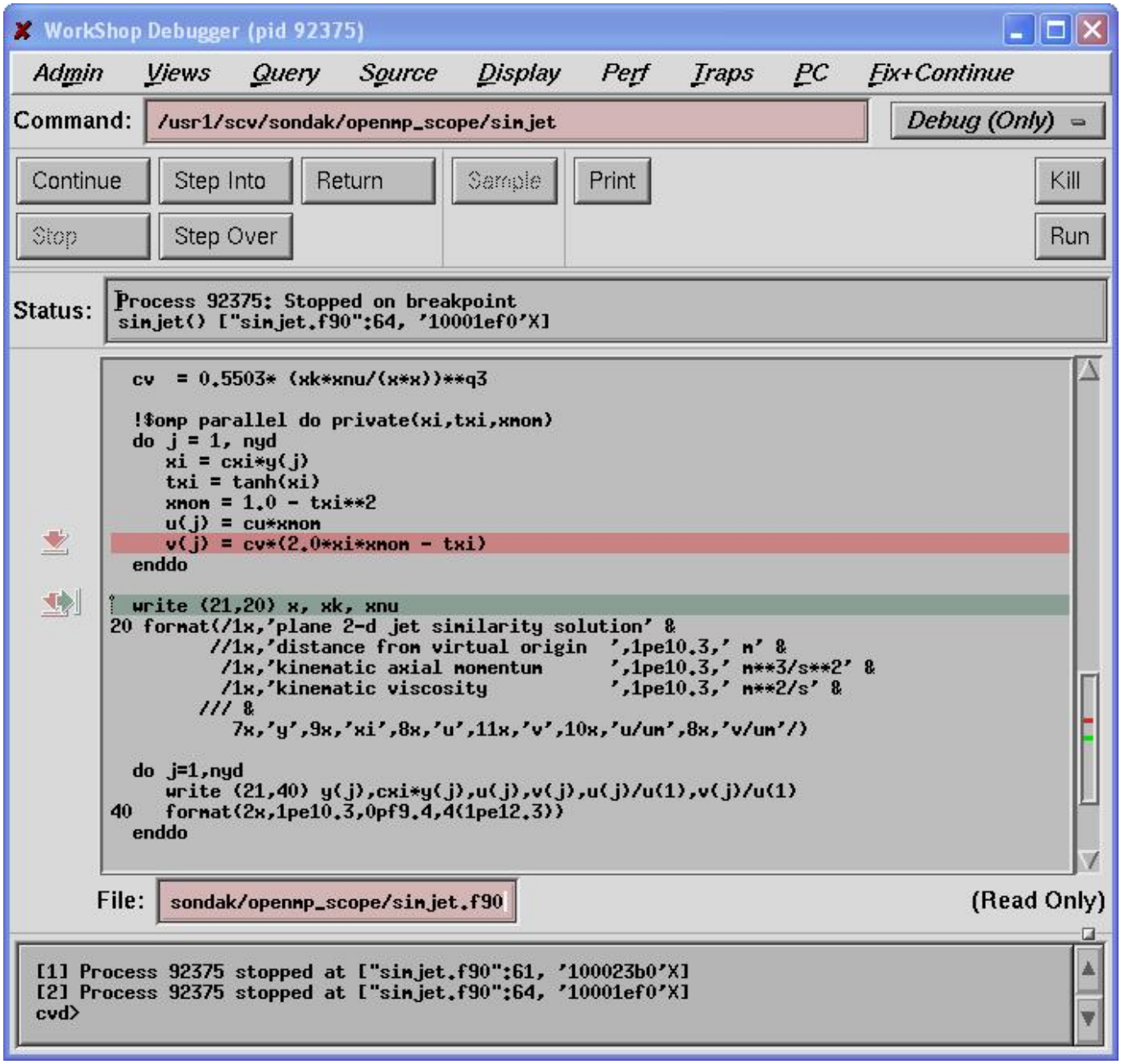
The Multiprocess View now shows three processes: a master process, which is not of concern here, and the two threads. Note that the Main window and the Expression View window both show the process id 92375 at the top, and the Expression View shows *j = 1*. By clicking on the other thread's process id (92371) in the Multiprocess View window and choosing it by selecting the Process menu item and then 'Change Focus to This Entry' from the options listed the process number at the top of both the Main and Expression View windows changes to 92371 and the Expression View values change to those for the other thread:



We are interested in the last value of *xmom* corresponding to *j = nyd = 70*, so we advance the code to that point:

Everything looks fine so far, with the value of *xmom* being equal to that shown earlier in the serial code output. Now we move outside the parallel loop. Since the additional thread disappears when we leave the parallel region, we first switch back to the original thread. The expression view now looks like:

Now we set the breakpoint and step out of the loop:



Note that the value of *xmom* has changed upon leaving the loop. This is because *xmom* was declared private, since each

thread requires its own copy of *xmom*, and it explains the bug. Upon examination of the loop, it is clear that *xmom* actually should have been declared *lastprivate*, so the last value in the serial version of the loop would have been retained after leaving the loop.

3.12.4 Scope Error Debugging Exercise

Below is a short code that is supposed to fill an array with integer values of a counter. It is designed to run with two threads. The array value for index 0 on thread 0 is supposed to equal 2, etc. If you try compiling and running this code you will find values very different than what you expect. The bug will manifest itself even on a single thread. (Recall what was mentioned in the lesson about this situation.) Use a multi-thread-capable debugger to find the problem.

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[]){

#define idim 10
      int i, icount = 1, k[idim];

#pragma omp parallel for private(icount)
      for(i=0; i<idim; i++){
            icount = icount + 1;
            k[i] = icount;
         }

      for(i=0; i<idim; i++){
            printf("k[%1i] = %1i\n",i,k[i]);
         }

   }
```

3.12.5 Scope Error Debugging Solution

In the lesson we mentioned that errors manifested on only a single thread are often scope errors. In this case the variable *icount* has been declared private since each thread must have its own value. However, the value is initialized to 1 in the serial part of the code. The *firstprivate* rather then the *private* clause is therefore required. Here is the correct code:

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[]){

#define idim 10
      int i, icount = 1, k[idim];

#pragma omp parallel for firstprivate(icount)
      for(i=0; i<idim; i++){
            icount = icount + 1;
            k[i] = icount;
         }

      for(i=0; i<idim; i++){
            printf("k[%1i] = %1i\n",i,k[i]);
         }

   }
```

3.13 Zero-based Indexing

3.13.1 Zero-based Indexing

Introduction

A common mistake that FORTRAN programmers make when writing MPI code is to forget that MPI starts counting ranks at 0 like in C, not at 1 as they are used to in FORTRAN. It is possible to get away with this mistake in simple MPI programs that hide the ranks in communicators and do not care that the master is rank 1 rather than rank 0. In more complicated programs, especially those which must deal with edge effects, starting at 1 instead of 0 can have major consequences.

Objectives

In this lesson, you will learn about zero-based indexing errors in Fortran and how to identify them using the Totalview® debugger. A sample FORTRAN program with a zero-based indexing error is provided that you may debug on your own while following along with the procedure described here.

3.13.2 Sample Program with Zero-based Indexing Error

The sample program shown below implements a parallel search of an integer array. The program should find all occurrences of a certain integer that is called the target. When a processor of a certain rank finds a target location, it should then calculate the average of

- The target value
- An element from the processor with rank one higher (the "right" processor). The right processor should send the first element from its local array.
- An element from the processor with rank one less (the "left" processor). The left processor should send the first element from its local array.

For example, if processor 1 finds the target at index 33 in its local array, it should get from processors 0 (left) and 2 (right) the first element of their local arrays. These three numbers should then be averaged.

Both the target location and the average are written to an output file. The program reads both the target value and all the array elements from an input file.

The input data file, b.data, is used for this example.

```
PROGRAM parallel_search

  include 'mpif.h'

  parameter (N=300)
  integer i, target
  integer b(N),a(N/4)
  integer rank,err
  integer status(MPI_STATUS_SIZE)
  integer end_cnt
  integer left,right
  integer lx,rx
  integer gi
  real ave

  common /pair/gi,ave  ! Put in common block to insure consecutive memory locations

  integer blocklengths(2)

  data blocklengths/1,1/                        ! Initialize blocklengths array

  integer types(2), MPI_Pair

  data types/MPI_INTEGER,MPI_REAL/      ! Initialize types array

  integer displacements(2)

  CALL MPI_INIT(err)

  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, err)

  CALL MPI_ADDRESS(gi,displacements(1),err)  ! Initialize displacements array with

  CALL MPI_ADDRESS(ave,displacements(2),err) ! memory addresses

! This routine creates the new data type MPI_Pair

  CALL MPI_TYPE_STRUCT(2,blocklengths,displacements,types, MPI_Pair,err)

 ! This routine allows it to be used in communication

  CALL MPI_TYPE_COMMIT(MPI_Pair,err)
```

```
  if (rank == 1) then
    open(unit=10,file="b.data")
    read(10,*) target
  end if

  CALL MPI_BCAST(target,1,MPI_INTEGER,0,MPI_COMM_WORLD,err)

  if (rank == 1) then
    do i=1,300
     read(10,*) b(i)
    end do
  end if

  CALL MPI_SCATTER(b,75,MPI_INTEGER,a,75,MPI_INTEGER,0,MPI_COMM_WORLD,err)

  if (rank == 1) then           ! Handle the special case of Processor 1
     left=4
     right=rank+1
  else if (rank == 4) then      ! Handle the special case of Processor 4
     left=rank-1
     right=1
  else
     left=rank-1                ! The "normal" calculation of the neighbor processors
     right=rank+1
  end if

  if (rank == 1) then
    open(unit=11,file="sfound.data")

! P0 sends the first element of its subarray a to its neighbors

    CALL MPI_SEND(a(1),1,MPI_INTEGER,left,33,MPI_COMM_WORLD, err)
    CALL MPI_SEND(a(1),1,MPI_INTEGER,right,33,MPI_COMM_WORLD, err)

! P0 gets the first elements of its left and right processor's arrays

    CALL MPI_RECV(lx,1,MPI_INTEGER,left,33,MPI_COMM_WORLD,status,err)
    CALL MPI_RECV(rx,1,MPI_INTEGER,right,33,MPI_COMM_WORLD,status,err)

    do i=1,75
      if (a(i) == target) then
        gi=(rank)*75+i
        ave=(target+lx+rx)/3.0
        write(11,*) "P",rank,gi,ave
      end if
    end do

    end_cnt=0

    do while (end_cnt .ne. 3)
      CALL MPI_RECV(MPI_BOTTOM,1,MPI_Pair,MPI_ANY_SOURCE,MPI_ANY_TAG, &
                    MPI_COMM_WORLD,status,err)
      if (status(MPI_TAG) == 52 ) then
        end_cnt=end_cnt+1
      else
       write(11,*) "P",status(MPI_SOURCE),gi,ave
      end if
    end do

  else

! Each slave sends the first element of its subarray a to its neighbors

    CALL MPI_SEND(a(1),1,MPI_INTEGER,left,33,MPI_COMM_WORLD, err)
    CALL MPI_SEND(a(1),1,MPI_INTEGER,right,33,MPI_COMM_WORLD, err)

! Each slave gets the first elements of its left and right processor's arrays

    CALL MPI_RECV(lx,1,MPI_INTEGER,left,33,MPI_COMM_WORLD,status,err)
```

```
    CALL MPI_RECV(rx,1,MPI_INTEGER,right,33,MPI_COMM_WORLD,status,err)

    do i=1,75
      if (a(i) == target) then
        gi=(rank)*75+i
        ave=(target+lx+rx)/3.0
        CALL MPI_SEND(MPI_BOTTOM,1,MPI_Pair,0,19,MPI_COMM_WORLD,err)
      end if
    end do

    gi=target ! Both are fake values
    ave=3.45  ! The point of this send is the "end" tag

    CALL MPI_SEND(MPI_BOTTOM,1,MPI_Pair,0,52,MPI_COMM_WORLD,err)

  end if

  CALL MPI_FINALIZE(error)

END PROGRAM parallel_search
```

3.13.3 Debugging the Sample Program

Now, we compile our sample code using the -g flag to enable source level debugging. When run on an IBM p690 this program generates errors:

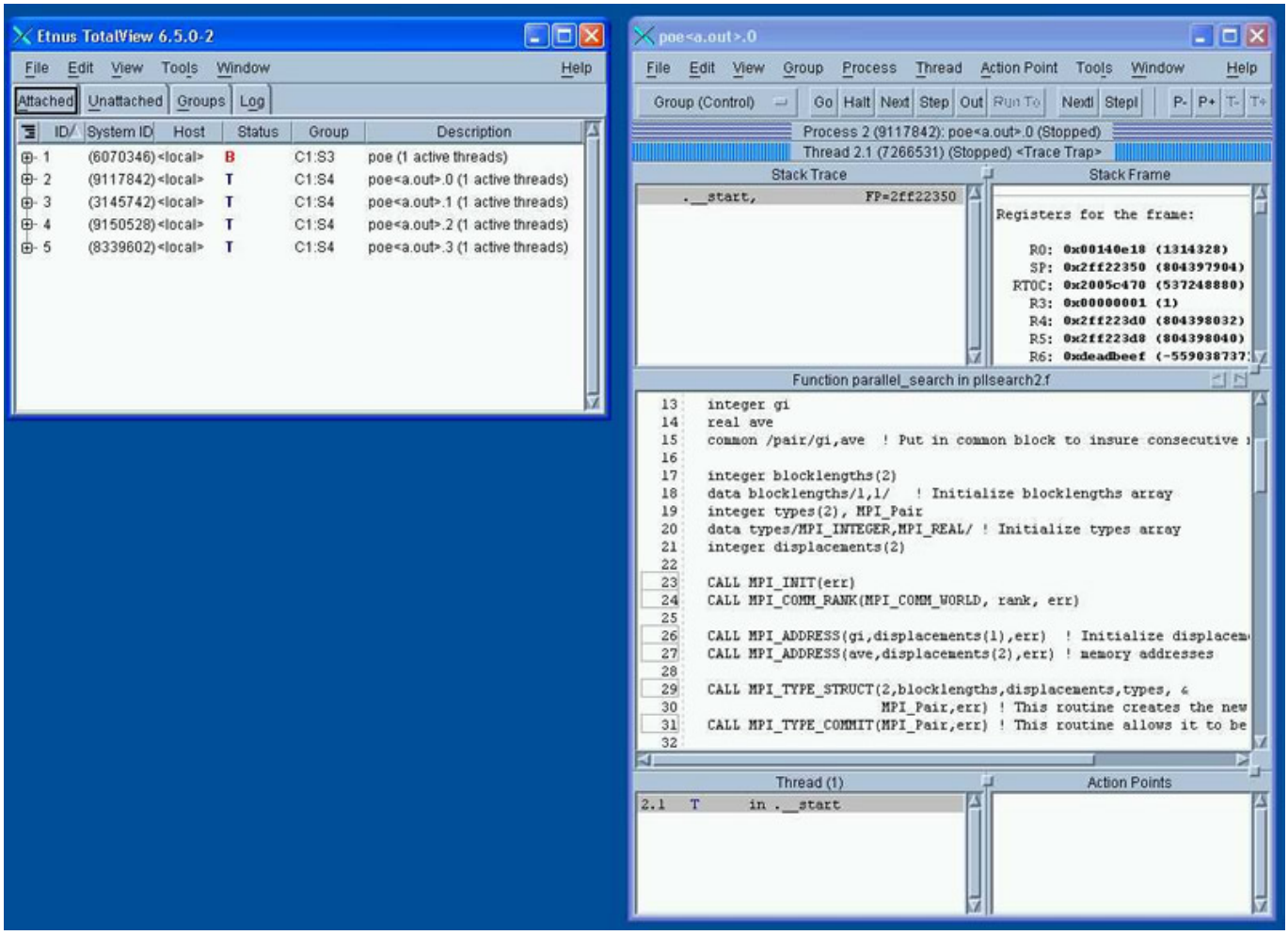
```
Cu12:~/debugging110% poe a.out -procs 4
ERROR: 0032-102 Invalid destination rank  (-1) in MPI_Send, task 0
ERROR: 0032-102 Invalid destination rank  (4) in  MPI_Send, task 3
```

It is clear from these messages that there is a problem with ranks. The invalid rank of -1 is a very strong hint as to the location of the problem. In a relatively simple code like this it should not be difficult to find the bug by just looking at the error message and source, but for the sake of illustration we will use Totalview® to show how to find this bug. You can then use a similar approach for more complex code.

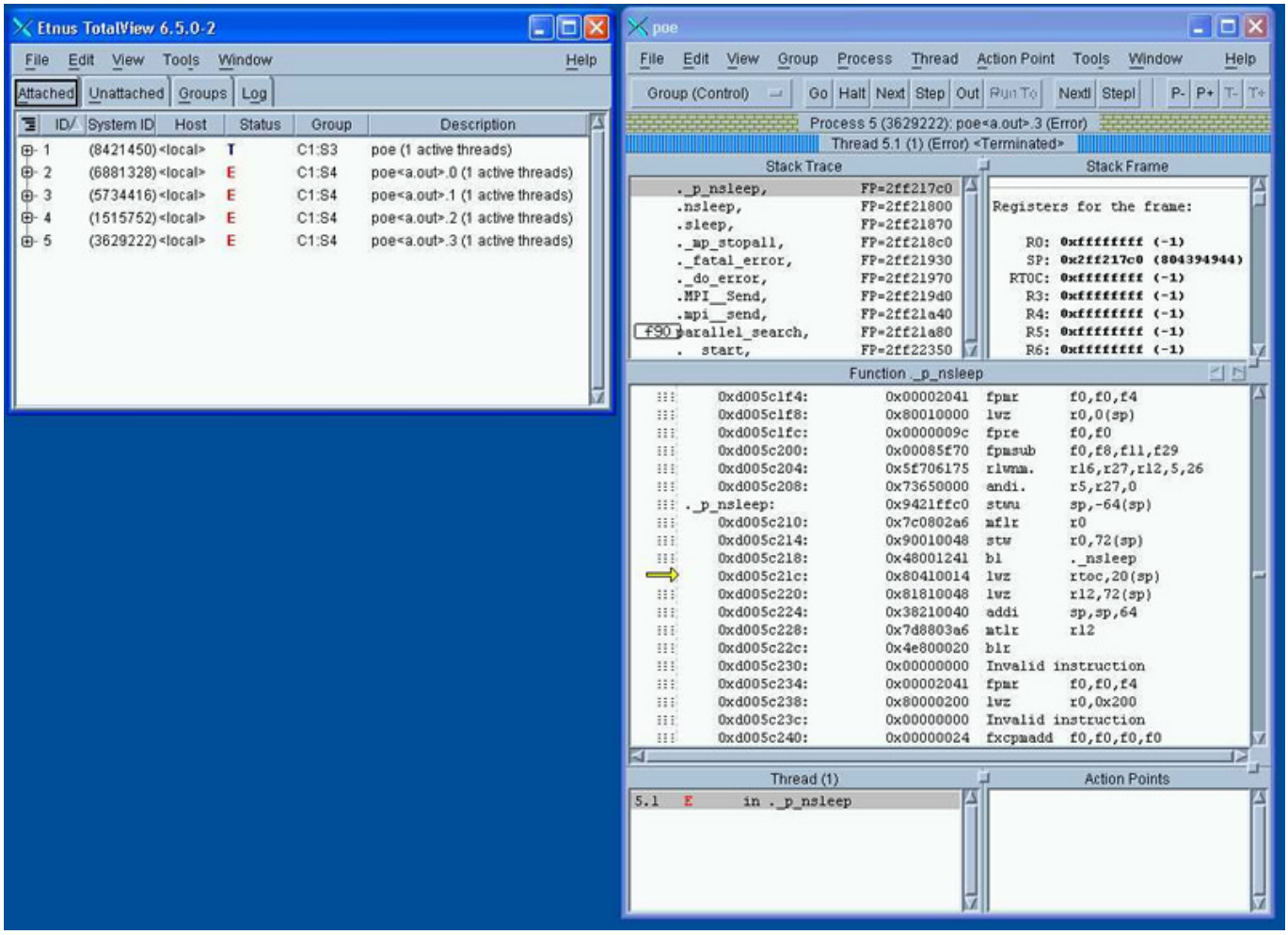
We run our sample MPI program under AIX and Totalview® by entering the following command:

```
totalview poe a.out procs 4   (or for a system using mpirun, totalview mpirun a np 4 a.out)
```

When Totalview® first starts it shows two windows like the ones shown below but without any source code. This is because of the two tier environment. Totalview® is running poe which is in turn running "a.out". To start debugging, click the 'Go' button. A dialog box will be displayed saying poe is a parallel job and asking Do you want to stop the job now? Click 'yes' and the source of parallel search should appear in the ProcessWindow on the right. The window on the left shows that Totalview® is now following 5 threads, the parent poe thread and the 4 a.out threads.



To start narrowing down the location of the bug, select 'Go' from the toolbar to run "a.out" to determine where it hangs. The result should look like the windows shown below.

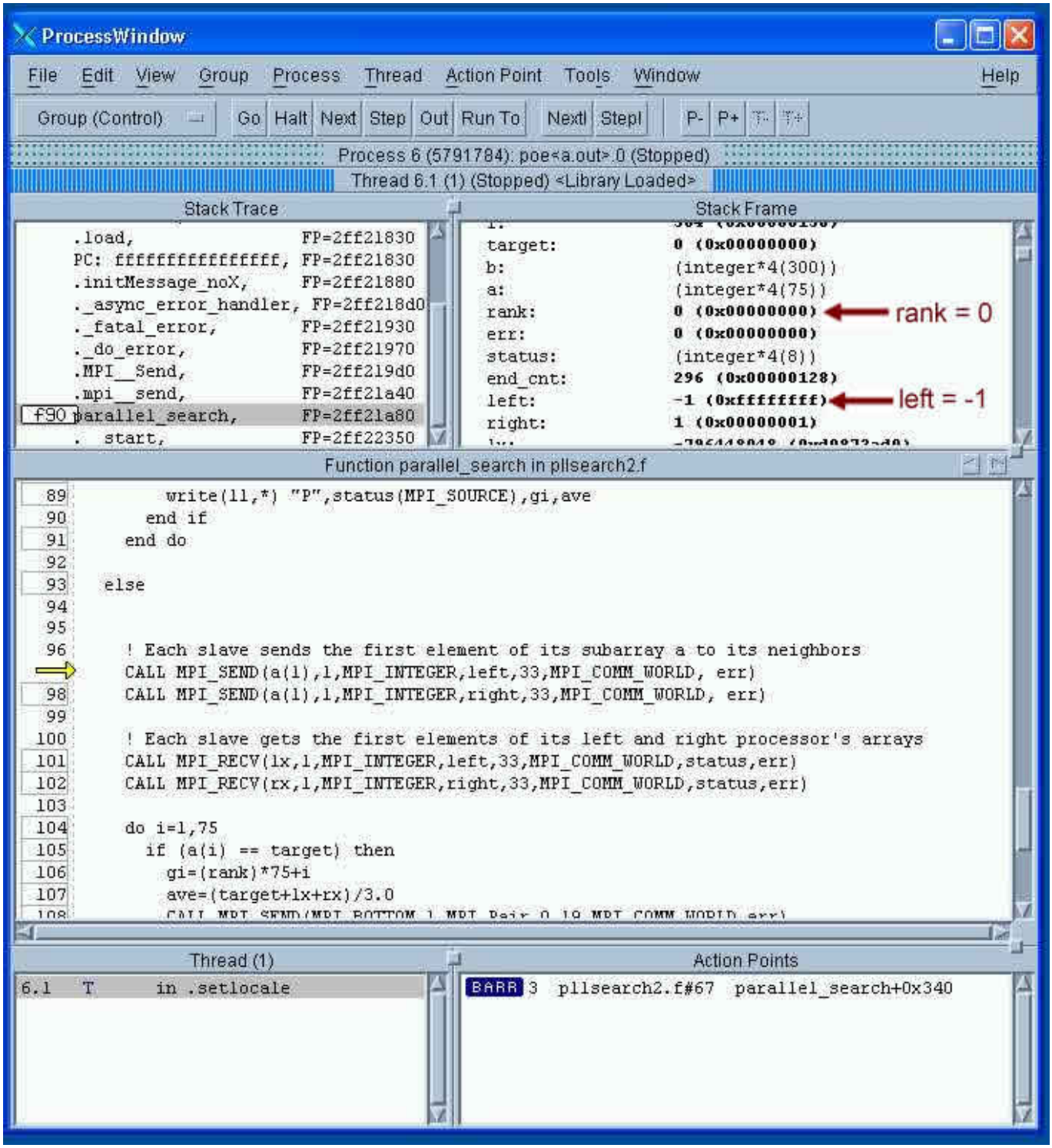


There are a couple of points to note in these windows that in a real debugging situation would probably provide enough information to help identify the bug. The first is that the source listing has switched to process 5, labeled "poe.3", indicating that there is no MPI process 4. The second is that when you double click on "f90" in the stack trace it takes you to line 96 in the source listing:

```
CALL MPI_SEND(a(1),1,MPI_INTEGER,left,33,MPI_COMM_WORLD, err)
```

Combining this information with the initial error message, "Invalid destination rank (-1) in MPI_Send, task 0," strongly suggests that *left* has a problem passing information to a neighbor.

Now that line 96 has been identified as a problem, choose restart from the 'Group' menu and insert a barrier at line 96. Once all of the processes have reached line 96 the ProcessWindow looks like the figure below.



Looking at the local variables displayed in the Stack Frame pane you can see that the *rank* is 0 and *left* is defined as -1, which should remind you that MPI ranks start from 0 and not 1.

3.13.4 Exercise

In the lesson we identified the bug as a zero indexing problem and localized it to line 96, the attempted passing of data to a process of rank -1. Your task in this exercise is to use this knowledge to fix this bug and the other one that is still lurking in the code.

3.13.5 Exercise Solution

Fixing the code turns out to be as simple as identifying the bug in the first place. The only changes that actually need to be made are in the section dealing with the definition of neighbors.

```
if (rank ==  0) then              ! Handle the special case of Processor 0
   left=3
```

```
   right=rank+1
else if (rank == 3) then        ! Handle the special case of Processor 3
   left=rank-1
   right=0
else
   left=rank-1                  ! The "normal" calculation of the neighbor processors
   right=rank+1
end if
```

Note that it is not necessary to change all instances of the type:

```
if (rank == 1) then
```

that deal with the master process, since MPI is quite happy to use any process as the master. However it IS good practice to use rank 0 as the master, especially if you are prone to forgetting the difference between MPI and Fortran as it will help you eliminate these mistakes.

CI-Tutor content for personal use only. All rights reserved. ©2015 Board of Trustees of the University of Illinois.