Debugging with TotalView
Exercises
March 22, 2010

{anmey|bierbaum|iwainsky|kapinos|reichstein|schmidl|wienke}@rz.rwth-aachen.de

1. Preparations

1. Introduction:

We prepared a small exercise in either C, C++ or Fortran for this TotalView lab. You are welcome to ask for help or further explanations.

2. Preparation:

Unpack the prepared program skeletons into your home directory (or into another suitable location) and change into the totalview lab directory.

cd $HOME
tar -xzf /home/hpclab/PPCES2010/ProgEnv/totalview_exercises.tgz
cd totalview_exercises

The current Intel compiler will be used on the Cluster Linux machines and Sun Studio compilers on the Virtual Machine (VM). We offer customized environment variables on Linux. These are $FC (Fortran), $CC (C), and $CXX (C++) for the compiler drivers.
Note: We recommend that you use these variables to build your programs.

Select a language and set the environment variable PROG_LANG accordingly:
export PROG_LANG= {c|cxx|fortran}

In the Virtual Machine (VM) the TotalView debugger is directly available without the need to load additional modules. In order to use the TotalView debugger on the cluster you have to load the module: **module load totalview**

To start the exercises in the chosen programming language, type **gmake totalview** in the totalview lab directory. The exercises will be started one after the other in the sequence as they are described. After closing a session of TotalView the next test program will be build and a new session of TotalView will be opened.

Alternatively you may choose any of the provided examples, change to that directory, and run **gmake totalview** in it.

2. Meet TotalView (Debugging basics)

The serial version of the Pi computer program will be built and the TotalView debugger will be started.

1. Inspect the options window. Press OK.
2. Inspect the main window. Would you see the source code if the program hadn't been compiled with debug information?
3. *Click View → Source As → Both* to enable assembler view alongside the source view.
4. Press the *GO* button to run the Pi program once. Note the output on the console.
5. Now run the program stepwise using the *STEP* and *NEXT* buttons. Follow the yellow arrow

- the program counter. Note the difference: *NEXT* executes the whole source code line; *STEP* steps into a subroutine if the subroutine is called in the line. Pick a line after the yellow arrow by clicking on it (the line will be highlighted in grey) and press the *RUN TO* button.

6. Now set a breakpoint by clicking on a line number. Note that you may not set breakpoints in all lines but only on lines containing statements.
7. Restart the program by clicking the *RESTART* button. Did the program reach the breakpoint?
8. Now examine the values of variables in as much ways as you can find. Note at least the dive option from the context menu (right-click) and the content of *Stack Frame* pane. Also try to navigate through the source code by using the dive option. Close the TotalView debugger by *File → Exit.*
   (End of the exercise. The next exercise will be started immediately after closing the TotalView debugger.)

3. Language-dependent features

1. Arrays in C/C++: Typecasting

(This exercise will only be performed if you choose the C or C++ programming language.)
In C/C++, an array can be represented by a pointer pointing to a memory piece. Debuggers cannot interpret such constellations the right way directly; but with a bit of user assistance the right interpretation of pointer and pointer target can be achieved.

1. After the TotalView debugger has started, inspect the options window and press OK.
2. Set a breakpoint on the line 10 (printf) and press the *GO* button to run the program.
3. Now examine the values of the ***a*** and ***b*** arrays (double-click on it to open a new variable window). Both windows should show the same content, but because ***b*** is a *pointer* to ***a***, the debugger cannot interpret it the right way.
4. (*Typecasting*) Now give some assistance to the debugger: click on the *Type* item in the ***b*** variable window and edit the assumed type of the variable ***b***. Choose the number *N* of the array elements in a meaningful manner because the debugger trusts you! Wrong numbers of *N* can lead to program abort.
   Old: int *
   New: int[*N*]*
5. Double-click on the *Value* item inside of the ***b*** variable window to obtain the overview of the array in the same way as for ***a***.
6. Run the program by clicking on *GO* button several times. Note how the values of both representations of the same data (***b*** is just a pointer to ***a***!) change.
7. Close the TotalView debugger by *File → Exit.*

(End of the exercise. The next exercise will be started immediately after closing the TotalView debugger.)

2. Fortran modules

(This exercise will only be performed if you choose Fortran programming language.)
In Fortran90/95 there are modules which can contain some variables. The non-private variables of any modules used by a USE statement e.g. in a subroutine are visible in this scope, leading to problems with identifying the current values of variables and their scope.

TotalView can show the Fortran modules in a convenient way.

1. After the TotalView debugger starts, inspect the options window and press OK.
2. Open the Fortran Modules window by *Tools → Fortran Modules*. Note that the window is empty because the program has not been started yet.
3. Set a breakpoint on the line 9 (WRITE) and press *GO* button to run the program.
4. After the program has stopped at the breakpoint, the Fortran Modules window is updated and now contains a module. Double-click on it to open a new window. This window shows the variables which are defined in the module.
5. After you have examined the module, close the TotalView debugger by *File → Exit* (End of the exercise. The next exercise will be started immediately after closing the TotalView debugger.)

4. Memory debugging

Typical memory errors are *double-free* errors (attempts to free an already freed memory piece) and *memory leaks* (some pieces of memory which are not freed at the right opportunity and become inaccessible). TotalView can help you to find such errors. In the following the search for *memory leaks* is described; many other views are available.

1. After the TotalView debugger has started, inspect the options window. Set the checkbox "*Enable memory debugging*" and press OK.
2. Let the program run. Note: if memory debugging is enabled, the runtime and memory footprint grow considerably.
3. A new window appears. Click on "*View memory data*" (glasses icon).
4. Another window, the ***MemoryScape*** window, appears.
5. Click on "*Leak Detection Reports*" and then on "*Source report*" to see the *Leak Detection Source Report*. You will be able to click through the application in the *Processes* pane. Counts the number of leaks and leaked bytes are available. The source code belonging to a particular memory leak will be shown in the right bottom *Source* pane. MemoryScape also shows a line number on which it believes to have detected a memory leak. However, note that the line number is not always accurate; moreover this line numbers must be understood as a "here or somewhere before" hint. Also, not all leaks can be detected; false positives can occur, too.
6. Close the TotalView debugger by *File → Exit.*
   (End of the exercise. The next exercise will be started immediately after closing the TotalView debugger.)

5. Debugging of OpenMP Programs

After you examined the options window and pressed the *OK* button the main window pops-up.

1. Look through the source code and set a breakpoint before the OpenMP parallel region. Let the program run into the breakpoint by pressing the *GO* button.
2. Examine the root window. How many threads are started at this time?
3. Remove this breakpoint and set another one at the OpenMP **PARALLEL** directive and press *GO*. Then press the *STEP* button and consider the root window again. How many threads do you see now? (Note: the step into the parallel region is known to be sometimes quite slow, please be patient!)
4. Now set a breakpoint on a statement *inside of* the **PARALLEL** region, remove other breakpoints, and restart the program. Press *GO* several times. Do all the threads have the

same state? What could be the cause of different thread states?

5. Right-click on the breakpoint and select *Properties*, change *When Hit, Stop* to *Thread*, press *OK*. Press *GO* at least once, note the status of the threads - now they all (except managing threads) have reached the breakpoint. By changing the *When Hit, Stop* property, the breakpoint works like a barrier.
6. Right-click on the loop variable ***i*** and select *Across Threads*. Press *OK* another couple of times and take a look on the values in the variable window. How are the values distributed? What is your assumption about the distribution of the work across the threads?
7. Now set a new breakpoint somewhere after the parallelised loop and remove the breakpoint within. The next click on GO lets the program run out of the parallel region. What happens with the variable ***i*** after the parallel region has ended? Note: the loop variable ***i*** is a private variable.

(End of the exercise. MPI debugging is integrated into the MPI lab.)