Debugging with TotalView Exercises

12.03.2013
http://www.rz.rwth-aachen.de/ppces
Tim Cramer, Paul Kapinos
cramer|kapinos@rz.rwth-aachen.de

0. Preparations

0.1. Introduction

We prepared a small exercise in either C, C++ or Fortran for this TotalView lab.
Please do not hesitate to consult us in case of question or any problems!

0.2. Preparation

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

```
cd $HOME
tar -xzf totalview_exercises.2013.tgz
cd totalview_exercises
```

The current Intel compiler will be used on the Cluster Linux. We offer customized environment variables on Linux.
These are `$FC` (Fortran), `$CC` (C), and `$CXX` (C++) for the compiler drivers.
Note: We recommend to use these variables to build your programs.
Select a language and set the environment variable `$PROG_LANG` accordingly:

```
export PROG_LANG= {c|cxx|fortran}
```

In order to use the TotalView debugger on the cluster you have to load the module:

```
module load totalview
```

To start the exercise `N` in the chosen programming language, type `gmake ueN` in the `totalview_exercises` lab directory. After closing a session of TotalView you have the opportunity to start the next.

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

1. Meet TotalView (Debugging Basics)

In this exercise the very basic handling of the TotalView debugger will be explained. To start, type `gmake ue1` in the `totalview_exercises` lab directory. 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 which may be a long-running subroutine call; `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.
9. Close the TotalView debugger by `File → Exit`.

(End of the exercise)

2. Language-dependent Features

In this exercise some language-specific features of the TotalView debugger will be explained. To start, type `gmake ue2` in the `totalview_exercises` lab directory. Depended on the `PROG_LANG` environment variable either C or Fortran exercise will be started.

2.1. 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 in 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 in 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)

2.2. Fortran Modules

(This exercise will only be performed if you choose Fortran programming language.)
In Fortran90/95 there is a feature available called `Fortran Modules`. A module can contain some variables and also subroutines. The non-private variables of any modules activated 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` in the `Process` window. 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)

3. 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.
To start, type `gmake ue3` in the `totalview_exercises` lab directory.

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. Counter for the number of leaks and the amount of 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. Is there a real memory leak in the given example? Try to solve the problem and do a leak detection again.
7. Close the TotalView debugger by `File → Exit`.

(End of the exercise)

4. Using of Core Files

With TotalView it is possible to do post-mortem analyse of a program, i.e. to analyse a core dump produced after a crash. You can open a core file either by just opening TotalView and choosing the same-named option in the `Options` window and browsing the paths to the executable file and core file, or directly from the command line: `totalview a.out corefile`. To start, go to the `[C|C++|F]-postMortem` dir (e.g. `cd F-postMortem`) and type `gmake die`. The program will be build, start and then crashes during the runtime, producing a core file, check it by `ls -la core*`

Now you can start TotalView:

```
module load totalview
totalview jacobi.exe <core file>
```

TotalView will start and give you an opportunity to see the call stack of the program immediately before it dies. Try to find out, why the program crashes. (To do it lazy way just type `gmake ue4` in the `totalview_exercises` lab directory - this will execute the all above steps for you).
Close the TotalView debugger by `File → Exit`.

(End of the exercise)

5. (optional) Debugging basics of OpenMP Programs

OpenMP programs are multithreaded, that is, there is only one process available and all communication is done over shared memory. The threads can all read and write the shared memory; each of them also can have some private data. Many typical errors of multithreaded programs cannot be discovered efficiently by using debugger; nevertheless debugging may be very helpful.
To start, type `gmake openmp` in the `totalview_exercises` lab directory.
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? You can rotate between the threads using `T- / T+` keys.
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 `GO` 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.)
8. Close the TotalView debugger by `File → Exit`.

(End of the exercise)

6. (optional) Debugging of MPI Programs

A Message Passing Interface (MPI) program is a program which run with more than one process. The processes have separate memory and communicate by explicit messages only. Debugging an MPI program with a debugger is a hard job, because many typical errors cannot be discovered easily with a debugger. Nevertheless, a debugger is much more better than no debugger.

6.1. Start Debugging of MPI Program and Debugging Basics

Usually a MPI program cannot be started directly. To mock-up the environment and to start the multiple processes the proper way a special program is used, usually called `mpiexec` or `mpirun`. TotalView can hook up into a MPI program in two ways: by adding `-tv` flag into the command line (`Classic Launch`) or by starting the debugger as for serial debugging and setiing up the parallel environment in the Parallel pane (`New Launch`).
To start, type `gmake mpi` in the `totalview_exercises` lab directory. (If the compilation fails, the environment may be broken; try `module reload`)

1. In the `Startup Parameters` window click on `Parallel` tab. Choose `Open MPI` as `Parallel System` for the defaultly loaded Open MPI and set the number of processes (`Tasks`, at least 2, try out more). Click on `OK`.
2. Set a breakpoint on the `MPI_Init` call, press `Go`. Be patient.
3. You can look the value of the variables over the all processes by right-click on the name and choosing Across Processes. What values has the `myrank` variable?
4. Set a breakpoint on the `MPI_Comm_size` call, press `Go`. Be patient.
   How did the values of `myrank` variable changed?
5. You can rotate between the processes `P-` / `P+` keys.
6. Click on `GO`. The program will hang in `Running` state. Click on `Tools` → `Message Queue Graph` to visualize the messages. Use this information in the next exercise!

6.2. The Debugging Challenge: Find the Errors

To challange you, we have prepared the MPI program used in the last exercise with several errors.

Try to find the five errors in the example.
(Hint: These errors are not actually programming errors, but violate the MPI-standard. This causes the parallel program to possibly exhibit unexpected, undesired and faulty behavior)