

Parallel Debugging with the Etnus TotalView

Princeton University

March 5, 2004

Dr. Jeff Durachta

Overview

- Introductory material
 - Course viewpoint
 - What is a “debugger”
 - Manifestations of programming error
 - Sources of programming error
 - “Beyond the debugger”
- Use of TotalView
 - Compiling
 - Launching jobs w/ MPI
 - “Diving”
 - Program motion
 - Tools
 - Parallel specific aspects

The Basics

- Course viewpoint
 - Course is about debugging techniques and the use of TotalView to enable those techniques.
 - Primary emphasis is on the use of techniques to solve problems, not a comprehensive review of TotalView capabilities.
 - Course is taught from the viewpoint (and limitations) of a knowledgeable, experienced user.
 - not a TotalView capabilities sales pitch.
 - This should be viewed as a forum for discussion.
 - Questions and comments about alternate methods are welcome.
 - Many questions will likely require research and responses will be posted to the group at a later date.
 - Online docs: etnus.com -> Support -> Documentation

What is a "debugger"?

- A tool for observing the result of high level source instructions as those instructions are executed on the target machine.
- In this context, one can:
 - View source at the high level language and assembler level.
 - Step (execute) program source "line by line".
 - See and manipulate the value of program variables at run time.

How do program errors manifest themselves?

- "Wrong" results
- "Segmentation Violations", "Bus Errors" and other memory errors leading to a core dump
- Other "bizarre behavior"
 - "Scaling" problems
 - Runs fine on 20PEs, but at 200PEs...
 - "Reproducibility" problems
 - Program "hangs"
 - Other runtime errors

Sources of errors

- Errors in logic flow
 - Basic "serial" logic flow
 - Parallel update errors
 - "Race" conditions
- Incorrect variables or arithmetic
- Language use errors
 - Pointer
 - Uninitialized variables
- Initialization errors
- Array bounds errors
- Other memory errors
 - Stack oversubscription (automatic arrays)
- When all else fails, blame the compiler
 - Optimization errors
 - Runtime environment and/or library errors

Parallel “Error Psychologies”

- The “process 0” mindset.
- The “single program, multiple data” (SPMD) mindset.
- The 10 KLoC / day “Code warrior”

Beyond Debugger Use

- Memory errors (mishandled pointers, stack overrun, etc) can be some of the most difficult problems to track down.
 - Optimization changes behavior
 - PE count generally changes behavior (but this can be an asset!)
- Optimization bugs are in general also some of the most difficult problems to track down.
- There is no substitute for incremental development and extensive testing.
 - Reproducibility tests across PE counts is an essential development tool.

Beyond Debugger Use

- The “Dual debugger sessions” technique is also an extremely useful (but often tedious).
 - Problems which appear for particular PE configurations
 - Changes from base level code
 - Cross platform porting
- In addition to “partial optimization”, use of global checksums and print statements to “triangulate” problem location can be useful.
 - In a large, complex application, array diving and local “debugger” checksums can be inefficient and time consuming.
 - Constant recompilation generally limits utility of this technique.
 - Don’t get locked into trying the same thing over and over on the same PE count in hopes of divine intervention.

Compiling for debugging

- Use of -g: disables any optimization
 - useful for many types of problems
 - will often miss certain types of language use errors such as uninitialized variables as well as many memory errors.
- Some compilers support “optimized” debugging
 - often loose access to certain types of variables (such as loop variables)
- “Partial optimization” is often a useful technique
 - Hierarchical “de-optimization”: track down optimizer bugs.
 - Spot “de-optimization”: long running codes.

Starting TotalView

- Non-parallel
 - `totalview <program_name> <corefile_name> -a <program_command_line>`
 - “Stripped” executables generally leave no call stack information in the core file.
- Parallel
 - Platform specific
 - Our platform:
 - “Old”: `mpirun -tv -np <npes> <prg.x>`
 - “New”: `mpirun -dbg=totalview -np <npes> <prg.x>`
- Review Menus
 - File
 - Edit
 - View
- Setting source location
- Setting console input

“Diving”

- Source
- Stack Trace (call tree)
- Stack Frame (variables)
- Action Points Frame
- Observing / setting variables
- Arrays
 - Array sections
 - Casting program variables
- Structures (C and F90)

Moving Through a Program

- Step / Next
- Setting Breakpoints
- Stepping “Out” of a routine
- Setting watchpoints / eval points
 - Careful: - expensive!
- Examining the call stack
- Function lookup
- Action Point Menu

Tools

- Array filtering
 - by comparison (e.g. > 100)
 - by IEEE value (\$nan, \$inf, \$denorm)
 - by range ($[>]lv:[<]hv$)
 - filter expressions
- Array statistics
- Array sorting
- Array visualization
- Monitoring memory use
- On-the-fly code edits

Parallel Specific Aspects

- Starting TotalView
 - Platform specific
 - Our platform:
 - “Old”: `mpirun -tv -np <npes> <prg.x>`
 - “New”: `mpirun -dbg=totalview -np <npes> <prg.x>`
- Action points
- Process windows
- Observing / setting variables
 - “Laminated” panes
 - Diving laminated panes
 - Editing laminated variables
 - Visualizing laminated variables