Advanced Parallel Debugging with TotalView

Nikolay Piskun
Software Architect, TotalView products, Perforce Software

August 2022
Agenda

- HPC Debugging and Dynamic Analysis with TotalView
- Advanced Debugging Technologies for HPC
- Combining HPC Debugging Technologies
- Q&A
HPC Debugging and Dynamic Analysis With TotalView
HPC Debugging With TotalView

- Comprehensive multi-process/thread dynamic analysis and debugging
- Debug Hybrid MPI/OpenMP applications
- Advanced C, C++ and Fortran support
- NVIDIA / CUDA GPU debugging support
- AMD / ROCm GPU Debugging
- Integrated reverse debugging
- Mixed language C/C++ and Python debugging
- Memory debugging and leak detection
- Batch/unattended debugging

Supported Technologies:

<table>
<thead>
<tr>
<th>LANGUAGES</th>
<th>OPERATING SYSTEMS</th>
<th>APPLICATIONS</th>
<th>PLATFORMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Linux</td>
<td>MPI</td>
<td>Intel</td>
</tr>
<tr>
<td>C</td>
<td>Mac OS</td>
<td>CUDA</td>
<td>NVIDIA</td>
</tr>
<tr>
<td>Ftn</td>
<td></td>
<td>OpenMP</td>
<td>AMD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ROCm</td>
</tr>
</tbody>
</table>
Debuggers – More Than Just a Tool to Find Bugs

• Understand complex code
• Improve developer efficiency
• Collaborate with team members
• Improve code quality
• Shorten development time
UI Navigation and Process Control
TotalView’s Default Views

1. Processes & Threads
   - Control View
     • Lookup File or Function
     • Documents

2. Source View

3. Call Stack View

4. Local Variables View

5. Data View, Command Line, Input/Output

6. Action Points, Replay Bookmarks

7. Array Tool
### Process and Threads View

#### Processes & Threads

<table>
<thead>
<tr>
<th>Description</th>
<th># P</th>
<th># T</th>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc_fork_loop (53)</td>
<td>4</td>
<td>4</td>
<td>p1-4</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>4</td>
<td>4</td>
<td>p1-4</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>4</td>
<td>4</td>
<td>p2.1, p4.1, p3.2,...</td>
</tr>
<tr>
<td>sleep</td>
<td>4</td>
<td>4</td>
<td>p1.3, p4.1, p3.2,...</td>
</tr>
<tr>
<td>tc_fork_loop.exe#62</td>
<td>4</td>
<td>4</td>
<td>p2.1, p4.1, p3.2,...</td>
</tr>
<tr>
<td>1.3</td>
<td>1</td>
<td>1</td>
<td>p1.3</td>
</tr>
<tr>
<td>2.1</td>
<td>1</td>
<td>1</td>
<td>p2.1</td>
</tr>
<tr>
<td>3.2</td>
<td>1</td>
<td>1</td>
<td>p3.2</td>
</tr>
<tr>
<td>4.1</td>
<td>1</td>
<td>1</td>
<td>p4.1</td>
</tr>
<tr>
<td>stoped</td>
<td>4</td>
<td>8</td>
<td>p1.1, p3.1, p1.2,...</td>
</tr>
<tr>
<td>__select__thread</td>
<td>2</td>
<td>3</td>
<td>p1.2, p2.2, p3.2</td>
</tr>
<tr>
<td>&lt;unknown line&gt;</td>
<td>2</td>
<td>3</td>
<td>p1.2, p2.2, p3.2</td>
</tr>
<tr>
<td>1.2</td>
<td>1</td>
<td>1</td>
<td>p1.2</td>
</tr>
<tr>
<td>2.2</td>
<td>1</td>
<td>1</td>
<td>p2.2</td>
</tr>
<tr>
<td>2.3</td>
<td>1</td>
<td>1</td>
<td>p3.2</td>
</tr>
<tr>
<td>sleep</td>
<td>3</td>
<td>5</td>
<td>p1.1, p3.1, p4.2,...</td>
</tr>
<tr>
<td>tc_fork_loop.exe#62</td>
<td>3</td>
<td>5</td>
<td>p1.1, p3.1, p4.2,...</td>
</tr>
<tr>
<td>1.1</td>
<td>1</td>
<td>1</td>
<td>p1.1</td>
</tr>
<tr>
<td>3.1</td>
<td>1</td>
<td>1</td>
<td>p3.1</td>
</tr>
<tr>
<td>3.3</td>
<td>1</td>
<td>1</td>
<td>p3.3</td>
</tr>
</tbody>
</table>

**Select process or thread attributes to group by:**
- [ ] Control Group
- [x] Share Group
- [ ] Hostname
- [x] Process State
- [x] Thread State
- [x] Function
- [x] Source Line
- [ ] PC
- [ ] Action Point ID
- [ ] Stop Reason
- [ ] Process ID
- [x] Thread ID
- [ ] Process Held
- [ ] Thread Held
- [ ] Replay Mode
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <pthread.h>

#define MAX_THREADS 10
#define LOCAL_FORK_COUNT 10
#define NULL NONINFRINGEMENT

int main (int argc, char **argv)
{
    int fork_count = 0;
    int args_ok = 1;
    int arg_count = 1;
    char *arg;
    pthread_mutexattr_t mattr;

    signal (SIGFPE, sig_fpe_handler);
    signal (SIGHUP, (void(*)(int))sig_hup_handler);

    #IFDEF __linux
    /* The linux implementation of pthreads uses these signals, so we'd better not */
    #ENDIF

    pthread_mutexattr_init (&mattr);
    pthread_mutexattr_setpshared (&mattr, FALSE);
    pthread_mutexattr_setscope (&mattr, PTHREAD_SCOPE_SYSTEM);
    pthread_mutexattr_setprotocol (&mattr, PTHREAD_PROTOCOL_PRIO_PROTECT);
    pthread_mutex_init (&forker_mutex, &mattr);

    for (fork_count = 0; fork_count < MAX_THREADS; fork_count++)
    {
        int new_tid = fork (forker, &fork_count, &attr);
        if (new_tid == -1)
            printf ("fork failed; result=%d, errno=%d\n", new_tid, errno);
        else
            thread_ids[total_threads++] = new_tid;

        fork (forker, &fork_count, &attr);
    }

    pthread_mutex_destroy (&forker_mutex);
}
```

Call Stack View and Local Variables View

- **Call Stack View:**
  - funcB
  - funcA
  - funcB
  - funcA
  - funcB
  - funcA
  - funcB
  - funcA

- **Local Variables View:**
  - **Arguments:**
    - b: int, Value: 0x00000012 (18)
  - **Block at Line 47:**
    - c: int, Value: 0x00000014 (20)
    - i: int, Value: 0x00000000 (0)
    - v: int[20], Value: (*int[20])
      - [0]: int, Value: 0x00000000 (0)
      - [1]: int, Value: 0x00000000 (0)
      - [2]: int, Value: 0x00000000 (0)
      - [3]: int, Value: 0x00000000 (0)
      - [4]: int, Value: 0x00000000 (0)
      - [5]: int, Value: 0x00000000 (0)
      - [6]: int, Value: 0x00000000 (0)
      - [7]: int, Value: 0x00000000 (0)
      - [8]: int, Value: 0x00000000 (0)
# Action Points View

<table>
<thead>
<tr>
<th>ID</th>
<th>Type</th>
<th>Stop</th>
<th>Location</th>
<th>Line</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Break</td>
<td></td>
<td>.../ReplayEngine_demo.cxx#27</td>
<td>ReplayEngine_demo.cxx (line 27)</td>
<td>main</td>
</tr>
<tr>
<td>2</td>
<td>Watch</td>
<td></td>
<td>4 bytes @ 0x601058 (arraylength)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Patch Code With Evaluation Points

- Evaluation points allow a segment of code to be run at a line number
- Patch code on the fly
- Use special directives such as $stopthread and $stopprocess to control threads and processes
Preferences

File > Preferences Menu

Or

“Gear” Toolbar Item

Preferences

Display Settings

Customize user interface display settings.

Appearance
Choose the best interface style for your development.

User Interface Style
Choose the type of user interface.

Font Size
Choose the font size for the user interface.

Display Settings changes will take effect the next time the product is started.
Advanced Debugging Technologies for HPC
MPI/OpenMP/GPU Hybrid Debugging
A variety of parallel programming models exist to extract maximum performance out of compute resources.

Message passing models are used to maximize parallelism across compute nodes – MPI technology.

Thread models, a type of shared memory programming, is used to maximize parallelism across cores within a compute node – OpenMP technology.

A hybrid programming model combines the parallelism provided by the message passing model (MPI) with the thread model (OpenMP).

Hybrid model also applicable to a CPU-GPU (Graphics Processing Unit) programming.
Debugging Hybrid Models – MPI/OpenMP/GPU

Hybrid Debugging with TotalView

• MPI Debugging
• OpenMP Debugging
• GPU Debugging
• Hybrid debugging
  • Mixing MPI and OpenMP

See it in action: https://totalview.io/webinars/debugging-hybrid-mpi-openmp-applications-remotely
CUDA Debugging
TotalView for the NVIDIA® GPU Accelerator

- NVIDIA Tesla, Fermi, Kepler, Pascal, Volta, Turing, Ampere
- NVIDIA CUDA 9.2, 10 and 11
- With support for Unified Memory
- Debugging 64-bit CUDA programs
- Support for dynamic parallelism
- Support for MPI based clusters and multi-card configurations
- Flexible Display and Navigation on the CUDA device
  - Physical (device, SM, Warp, Lane)
  - Logical (Grid, Block) tuples
- GPU Status view shows how code runs on GPUs
- Support for types and separate memory address spaces
- Leverages CUDA memcheck
Advanced GPU Debugging With the GPU Status View

• Easily understand how your code is running across one or more GPUs.
• Use a simple attribute aggregation interface and filters to define an informative GPU Status display.
• Built to support one or more GPUs within a node and across a cluster.
AMD / ROCm GPU Debugging
TotalView ROCm GPU Support

- Process launch, attach, detach, etc.
- GPU ELF code-object load events
- Both deferred and non-deferred loading
- Registers (scalar, vector, general, special)
- Instruction disassembly
- Breakpoint create/delete, events
- Single-stepping and fast smart-stepping
- Stack unwinding (including inlined functions)
- GPU navigation controls
- Variable display (with AFAR compilers only)

Compile as follows

- ROCm 4.5/5.x: 
  -O0 -ggdb
- afar001-264: 
  -O0 -mlirm -amdgpu-spill-cfi-saved-regps -gheterogeneous-dwarf
- afar001-273: 
  -O0 -g
ROCm GPU Agents Are Represented As TotalView Threads

- TotalView uses a “one TotalView thread per GPU agent (device)” model (like CUDA)
- All waves on an agent within a process are grouped within a single “super thread”
- Each super thread has a GPU focus thread (a lane, within a wave, on the agent) controlled by the user

Super thread 1.-1 for Agent 3 (active)

Super thread 1.-2 for Agent 4 (idle)
ROCm GPU Focus Control

**New UI**

- **Logical focus by work-group / work-item**

- **Physical focus by for Agent 3 (active)**

**Classic UI**

- Logical focus by work-group / work-item
- Physical focus by for Agent 3 (active)
Logical and Physical Focus, and Grid Dimensions

Logical focus displayed in thread status

Physical focus
Agent:Queue:Dispatch:Workgroup:Lane

Grid dimensions
Variable Display With the AFAR Compilers

- AFAR compilers can generate DWARF for variables
- There are limitations (ask AMD)
- Support planned for ROCm 5.1 (AFAIK)
- Built-in variables (block/thread idx/dim) can be displayed and used in expressions
Debugging AMD GPUs with TotalView for AMD GPUs

• TotalView does not “officially” support AMD GPUs yet, but…

• “Unofficial” support is included in production versions of TotalView

• Official AMD GPU support coming later this year

• Enabled it using the “-rocm” flag, for example:
  • totalview -rocm a.out

• Latest TotalView 2022.2 version supports ROCm 5.1
Remote Debugging
TotalView Remote UI

- Combine the convenience of establishing a remote connection to a cluster and the ability to run the TotalView GUI locally.
- Front-end GUI architecture does not need to match back-end target architecture (macOS front-end -> Linux back-end)
- Secure communications
- Convenient saved sessions
- Once connected, debug as normal with access to all TotalView features

See it in action: https://totalview.io/video-tutorials/how-use-remote-user-interface-debugging
Reverse Debugging Connections
Disconnect Backend Job Launch with Reverse Connect

- Start a debugging session using TotalView Reverse Connect.
- Reverse Connect enables the debugger to be submitted to a cluster and connected to the GUI once run.
- Enables running TotalView UI on the front-end node and remotely debug jobs executing on the compute nodes.
- Very easy to utilize, simply prefix job launch or application start with “tvconnect” command.

```
#!/bin/bash
#SBATCH -J hybrid_fib
... #SBATCH -n 2
#SBATCH -c 4
#SBATCH --mem-per-cpu=4000
export OMP_NUM_THREADS=4
tvconnect srun -n 2 --cpus-per-task=4 --mpi=pmix ./hybrid_fib
```
ANL Connect Demo

- TotalView Reverse Connect Demo
- TotalView Remote Debugging Demo
Reverse Debugging
Reverse Debugging With TotalView

- Reverse debugging provides the ability for developers to go back in execution history.
- Activated either before program starts running or at some point after execution begins.
- Capturing and deterministically replay execution.
- Enables stepping backwards and forward by function, line, or instruction.
- Run backwards to breakpoints.
- Run backwards and stop when a variable changes value.
- Saving recording files for later analysis or collaboration.

See it in action: https://totalview.io/video-tutorials/reverse-debugging
Reverse Debugging Controls

Run forward
- Next forward over functions
- Step forward into functions
- Advance forward out of function call
- Advance forward to selected line
- Advance to “live” session
- Create a bookmark at this point in recorded history
- Save the recorded session

Run backwards
- Next backwards over functions
- Step backwards into functions
- Advance backwards to calling function
- Advance backward to selected line
Memory Debugging
TotalView HPC Memory Debugging

- Easily find memory leaks and other memory errors
- Understand heap usage
- Detect malloc/free new/delete API misuse
- Detect buffer overruns
- Understand where memory is being used
- Remote and MPI debugging
Python Debugging
Mixed Language Python Debugging

• Debugging one language is difficult enough.
• Understanding the flow of execution across language barriers is hard.
• Examining and comparing data in both languages is challenging.

• What TotalView provides:
  • Easy python debugging session setup.
  • Fully integrated Python and C/C++ call stack.
  • ”Glue” layers between the languages removed.
  • Easily examine and compare variables in Python and C++.
  • Modest system requirements.
  • Utilize reverse debugging and memory debugging.

See it in action: https://totalview.io/video-tutorials/debugging-python-and-c-mixed-language-applications
Find Tough Bugs by Combing Debugging Technologies
Combine Multiple Debugging Technologies

- Find where a mutex lock was acquired
  - Combine reverse debugging and watchpoints
  - Run backwards until pthread_mutex_t __owner changes
- Mix source code debugging, reverse debugging and memory debugging
  - Find memory allocations and leaks during your debugging session
- Use TotalView’s Remote UI for efficient debugging using all TotalView’s features from your laptop
Attach and Detach from a Parallel Job

- Peek at the state of your parallel job
- Use TotalView’s attach and detach capabilities to examine the job and then let it continue to run
- Attaching to starter process enables TotalView to discover and attach to all (or a subset) of the ranks
Process/Thread Aggregation

- Aggregate process and thread state to quickly understand the state of the job
- Find outliers quickly
- Views allow different configuration to be easily switched
Using TotalView for Parallel Debugging on ANL
TotalView remote debugging on Linux and Mac OS

- Download and install TotalView on your linux or mac. (ignore license)
  - `/grand/ATRESC2022/EXAMPLES/track-6-tools/TotalView/`
  - `www.totalview.io/downloads`

- Copy `/grand/ATRESC2022/EXAMPLES/track-6-tools/TotalView/2022.labs.tar.gz` to your area and untar it

- Run make to build examples.

- Connect to remote front node from the `terminal`

- Run labs remotely
TotalView is available on Theta, ThetaGPU and Cooley

- Installed at: /soft/debuggers/totalview-2022-08-04/toolworks/totalview.2022.2.13/bin/totalview

- Connect to Cooley (use soft add +totalview to setup Reverse Connect)
  - Get allocation first
    - qsub -A ATPESC2022 -n 1 -q training -l
    - soft add +totalview
  - totalview -args aprun -np <N> ./demoMpi_v2 (*)
  - tvconnect aprun -np <N> ./demoMpi_v2 (*)
  - (*) Supposed to work 😊

- Connect to ThetaGPU (use module load totalview to setup Reverse Connect)
  - Get allocation first
    - qsub -A ATPESC2022 -n 1 -q single-gpu -l
    - module load totalview

- Connect to Theta (use module load totalview to setup Reverse Connect)
  - module swap PrgEnv-intel PrgEnv-cray ; module swap PrgEnv-cray PrgEnv-intel
  - setenv CRAYPE_LINK_TYPE dynamic

- Get allocation first
  - qsub -A ATPESC2022 -n 4 -q debug-flat-quad -l
  - module load totalview totalview-support
  - totalview -args aprun -np <N> ./demoMpi_v2 (*)
  - tvconnect aprun -np <N> ./demoMpi_v2 (*)
Hands-on labs

- Remotely connect to machine and enable Reverse Connection
- Copy /grand/ATRESC2022/EXAMPLES/track-6-tools/TotalView/ATRESC2022-TV-labs.tar.gz
- Programs are in labs/programs/

Labs:
- Lab 1 Debugger Basic
- Lab 2 Viewing, Examining, Watching and Editing Data
- Optional Lab 3 Examining and Controlling a Parallel Application (on Cooley)
  - Using remote connect (tvconnect)
  - qsub –q training tvconnect.job
  - Modify and submit tvconnect.job on your machine

Bonus lab: on thetaGPU: (ssh –y thetagpusn1)
- qsub -l -n 1 -t 30 -q single-gpu -A ATPESC2022
- /usr/local/cuda/bin/nvcc –g –G tx_cuda_matmul.cu –o tx_cuda_matmul
- /grand/ATRESC2022/EXAMPLES/track-6-tools/TotalView/toolworks/totalview.2022.2.13/bin/tvconnect tx_cuda_matmul
Remote submission of batch job.

- Submit job from TotalView (qsub 2022/labs/programs/tvconnect-thetaGPU.job)

- tvconnect-thetaGPU.job:

  ```bash
  #!/bin/bash
  #COBALT -t 30
  #COBALT -n 4
  #COBALT -q single-gpu
  #COBALT -A ATPESC2022
  module load totalview
  tvconnect tx_cuda_matmul
  ```
TotalView Resources and Documentation

• TotalView website:
  • https://totalview.io

• TotalView documentation:
  • https://help.totalview.io

• TotalView Video Tutorials:
  • https://totalview.io/support/video-tutorials

• Other Resources:
  • Blog: https://totalview.io/blog
Summary

• Use of modern debugger saves you time.

• TotalView can help you because:
  • It’s cross-platform (the only debugger you ever need)
  • Allow you to debug accelerators (GPU) and CPU in one session
  • Allow you to debug multiple languages (C++/Python/Fortran)
TotalView
Resources and Documentation
TotalView Resources and Documentation

- TotalView website:  
  https://totalview.io

- TotalView documentation:
  
  - https://help.totalview.io
  
  - User Guides: Debugging, Memory Debugging and Reverse Debugging
  
  - Reference Guides: Using the CLI, Transformations, Running TotalView

- Blog:  
  https://totalview.io/blog

- Video Tutorials:  
  https://totalview.io/support/video-tutorials
Q&A
Questions

• Any questions or comments?
  • Don’t hesitate to reach out to me directly with any questions or comments!
  • Email: npiskun@perforce.com

• Thank you for your time today!