LBL/NERSC Agenda – September 2022

• Introduction
• Latest TotalView Features
• TotalView Roadmap
• Remote Debugging Techniques
• Review of General Debugging Features
• TotalView Debugging at NERSC Best Practices
• GPU Debugging with TotalView on Perlmutter (10:00am)

• MPI and OpenMP debugging
• Reverse Debugging
• Memory Debugging
• Common TotalView Usage Questions
• Q&A
Solving the Hardest Challenges in DevOps
Perforce Products

Agile Management
- Helix ALM
- Hansoft
- Gliffy

Code Management & Collaboration
- Helix Core
- Methodics
- Helix4Git
- JRebel
- TotalView

Application Mgmt. & Components
- Puppet
- Akana
- OpenLogic
- Zend
- Visualization
- SourcePro
- IMSL

Automated Testing
- Helix QAC
- Klocwork
- Perfecto
- BlazeMeter
Overview of TotalView Labs
Overview of TotalView Labs

**Nine different labs and accompanying example programs**

- Lab 1 - Debugger Basics: Startup, Basic Process Control, and Navigation
- Lab 2 - Viewing, Examining, Watching, and Editing Data
- Lab 3 - Examining and Controlling a Parallel Application
- Lab 4 - Exploring Heap Memory in an MPI Application
- Lab 5 - Debugging Memory Comparisons and Heap Baseline *
- Lab 6 - Memory Corruption discovery using Red Zones *
- Lab 7 - Batch Mode Debugging with TVScript
- Lab 8 - Reverse Debugging with ReplayEngine
- Lab 9 - Asynchronous Control Lab

**Notes**

- * Labs 5 and 6 require use of TotalView’s Classic UI
- Sample program breakpoint files were created with GNU compilers. If a different compiler is used, they may not load and will need to be recreated.
- Several example programs use OpenMPI so you will need to configure your environment beforehand.
- We do not have a lab specific to Python Debugging yet. There are good examples and instructions in the TotalView totalview.<version>/linux-x86-64/examples/PythonExamples directory.
TotalView Features
What is TotalView used for?

- Provides interactive Dynamic Analysis capabilities to help:
  - Understand complex code
  - Improve code quality
  - Collaborate with team members to resolve issues faster
  - Shorten development time

- Finds problems and bugs in applications including:
  - Program crash or incorrect behavior
  - Data issues
  - Application memory leaks and errors
  - Communication problems between processes and threads
  - CUDA application analysis and debugging

- Contains batch and Continuous Integration capabilities to:
  - Debug applications in an automated run/test environment
TotalView Features

- Multi-process/thread dynamic analysis and debugging
- Comprehensive C, C++ and Fortran Support
- Thread specific breakpoints with individual thread control
  - View thread specific stack and data
- View complex data types easily
- MPI, OpenMP, Hybrid support
- NVIDIA (CUDA) and AMD (HIP) GPU support
- Convenient remote debugging
- Integrated Reverse debugging
- Mixed Language - Python C/C++ debugging
- Memory debugging
- Script debugging
- Linux, macOS and UNIX
- More than just a tool to find bugs
  - Understand complex code
  - Improve developer efficiency
  - Collaborate with team members
  - Improve code quality
  - Shorten development time
Recent TotalView Features
TotalView Major Additions and Updates (September 2022)

**Major Feature Updates**
- Startup performance improvements
- Remote UI connections
- TotalView Reverse Connect
- Display thread names
- Filter `dlopen` events to improve startup performance
- Debug through start shell scripts
- Remote Display Client update
- Undo LiveRecorder support
- Security updates

**Platform Updates**
- Cray EX (Shasta)
- macOS Monterey and Big Sur
- CUDA 11.0 – 11.7, A100 Ampere and MIG support
- AMD and NVIDIA GPU support
- OpenMPI 4 and 5, OpenMP 5.x
- GCC 9, 10 and 11, Intel Parallel Studio XE 2020, Intel oneAPI 2022,
- RHEL/CentOS 8, Fedora 32 – 34, Ubuntu 20.04, 64-bit Solaris
- Python 3.5 – 3.9 support
TotalView Major Additions and Updates (September 2022)

- Data Debugging Dive Stacks
- Memory Debugging Block Notify
- Memory debugging leak detection, heap status, and memory events
- NVIDIA GPU Status View, CUDA Memcheck
- Process Hold
- Dive-in-All
- Data View workflow improvements
- Performance improvements
- Array statistics

- Documents view
- Local Variable view
- Input/Output view
- Font size preference
- Detaching from processes
- Dark theme
Remote Debugging Techniques
Remote Debugging Technologies

- Debugging on a remote HPC cluster can be a challenge
  - Setting up the secure remote connection
  - Launching/connecting to the target application
  - Interactive debugging UI
- TotalView Remote Debugging Options
  - **NERSC - NoMachine**
    - Follow: https://docs.nersc.gov/connect/nx/
  - **TotalView Remote UI**
    - Run the TotalView UI on a remote client and connect to the remote TotalView debugger
  - **TotalView Remote Display Client (RDC)**
    - Conveniently setup a remote VNC connection
  - **VNC Server**
    - Set up a remote VNC desktop for efficient development and debugging
  - **TotalView Reverse Connect**
    - Connect back to the TotalView UI from remote node
Remote Debugging - TotalView Remote UI
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
- Front-end GUI currently supports macOS and Linux x86/x86-64. Windows client is coming.
Remote UI Architecture

Remote System

TotalView UI

SSH

Front-end Node

TotalView UI Debugger “Client”

Batch Node

tvdsvr

srun

Compute Nodes

Rank 0
Setting up the Remote UI for NERSC

- Use NERSC sshproxy.sh to set up SSH Keys
- Example: cori.nersc.gov
- Configure Remote UI Session
  - Connection Name: cori.nersc.gov
  - Remote Host: wburns@cori.nersc.gov
  - Private Key File: /Users/bburns/.ssh/nersc
  - Remote Command(s):
    module load totalview
  - No need to specify TotalView Remote Installation Directory
- On Start Page, select “cori.nersc.gov” for Launch Remote Debugger
Remote Debugging – TotalView Remote Display Client (RDC)
TotalView Remote Display Client (RDC)

• Offers users the ability to easily set up and operate a TotalView debug session that is running on another system

• Consists of three components
  • Client – runs on local machine
  • Server – runs on any system supported by TotalView and “invisibly” manages the secure connection between host and client
  • Viewer – window that appears on the client system

• Remote Display Client is available for:
  • Linux x86, x86-64
  • Windows
  • macOS
TotalView Remote Display Client

- Free to install on as many clients as needed
- No license required to run the client
- Presents a local window that displays TotalView or MemoryScape running on the remote machine
- Requires SSH and X Windows on Server
TotalView Remote Display Client

- User must provide information necessary to connect to remote host
- Passwords are NOT stored
- Information required includes:
  - Username, public key file, other ssh information
  - Directory where TotalView/MemoryScape is located
  - Path and name of executable to be debugged
  - If using indirect connection with host jump, each host
    - Host name
    - Access type (Username, public key, other ssh information)
    - Access value
- Client also allows for batch submission via PBS Pro or LoadLeveler
TotalView Remote Display Client
Session Profile Management

- Connection information can be saved as a profile, including all host jumping information
- Multiple profiles can be generated
- Profiles can be exported and shared
- Generated profiles can be imported for use by other users
Setting up the Remote Display Client for NERSC

- Use NERSC sshproxy.sh to set up SSH Keys
- Example: Cori
- Configure RDC Session
  - Remote Host: `cori.nersc.gov`
  - Other SSH Options: `-l wburns -i /Users/bburns/.ssh/nersc`
  - Path to TotalView on Remote Host: `/global/common/cori_cle7up03/software/toolworks/totalview.2022.2.13/bin`
- Click “Launch Debug Session”
- Starts Classic TotalView UI
- Exit the Classic UI and run “totalview -newui” to run the new UI
- Can forward X11 display from Compute Node to VNC Server session by setting DISPLAY environment variable
RDC Demo

- Remote Display Client Demo

TotalView Power Tip

- On Windows installations, select TotalView RDC from the TotalView Remote Display Client folder. Do not select “Start”.
Remote Debugging – VNC Server
Debugging in a VNC Server Session

- Use a VNC Server to construct a remote desktop
- Provides efficient graphical display to a local VNC Viewer running on your workstation or laptop
Setting up VNC Server for Cori at NERSC

• Use NERSC sshproxy.sh to set up SSH Keys

• Example: Cori

• Create SSH tunnel for VNC session
  • Create a "tunnel" for remote display information from port 5999 on the end system to port 5999 on your local system
  • `ssh -l wburns -i /Users/bburns/.ssh/nersc -L 5999:localhost:5999 cori.nersc.gov`

• Start VNC Server on login node
  • On the resulting Cori login node, e.g. cori08
  • Start a VNC Server for display 99, which will be opened through port 5999 on the system and tunnelled
    • `vncserver -geometry 1800x970 :99`

• Start VNC Viewer on your local system and connect to the tunneled local port
  • `localhost:5999`
SSH Tunnel with Putty on Windows

1. Start the PuTTY application on your desktop. In the Session windows, enter the hostname or IP address and port number of the destination SSH server. Make sure the connection type is set to SSH.

2. Add hostname of the SSH server you want to access remotely.

1. In the left sidebar under the Category options, navigate to the Connection >> SSH >> Tunnels.

2. Select Local to define the type of SSH port forward.

3. In the Source port field, enter the port number to use on your local system. (For example Source port: 5050)

4. Next, in the Destination field, enter the destination address followed by the port number. (For example Destination: 127.0.0.1:5432).

5. Verify the details you added, and press Add button. You can add multiple entries here.

6. All done. Connect the SSH session to make the tunnel. The tunnel will work until the SSH session is inactive.
VNC Server Demo

- VNC Demo

TotalView Power Tip

- Use the VNC Server “-geometry” argument to define your VNC desktop size, e.g. vncserver -geometry 1920x1048 :99
- Modify your ~/.vnc/xsession file to control which X11 window manager is used and any startup applications.

```bash
#!/bin/sh
xsetroot -solid grey
xterm -geometry 80x24+10+10 -ls -title "$VNCDESKTOP Desktop" &
icewm &
```
TotalView Reverse Connections
TotalView Reverse Connections

The Problem:

- Establishing an interactive debugging session in a cluster environment can be difficult
  - Timing issues when submitting through a job manager and when the job runs
  - The organization of modern HPC systems often makes it difficult to deploy tools such as TotalView
  - The compute nodes in a cluster may not have access to any X libraries or X forwarding
  - Launching a GUI on a compute node may not be possible

The Solution:

- Disconnect starting debugger UI from the backend job launch and debug session acquisition
- TotalView Reverse Connect workflow enables developers to start the TotalView UI on a front-end node and, when a job is run in the cluster, a remote TotalView reverse connect agent connects it back to the waiting UI
Reverse Connection Flow

1. tvconnect writes request
   4. tvconnect reads response

2. TotalView UI reads request
3. TotalView returns response

$HOME/.totalview/connect

6. socket connection opened

BATCH NODE

tvconnect

5. exec

tdsvr

srunit

COMPUTE NODES

Rank 0

FRONT-END NODE

TotalView UI

Rank 0
Reverse Connection Flow

1. tvconnect writes request
2. TotalView UI reads request
3. TotalView returns response
4. tvconnect reads response
5. exec
6. socket connection opened

$HOME/.totalview/connect

BATCH NODE

FRONT-END NODE

TotalView UI

COMPUTE NODES

Rank 0
Reverse Connection Flow

**FRONT-END NODE**

1. TotalView UI reads request
2. TotalView UI reads request
3. TotalView returns response

**BATCH NODE**

1. tvconnect writes request
2. tvconnect reads response
3. tvconnect reads response
4. tvconnect reads response
5. execute

**$HOME/.totalview/connect**

**COMPUTE NODES**

Rank 0
Reverse Connection Flow

1. tvconnect writes request
2. TotalView UI reads request
3. TotalView returns response
4. tvconnect reads response
5. exec
6. socket connection opened

$HOME/.totalview/connect

TotalView UI

Batch Node

Compute Nodes

Rank 0

TotalView by Perforce © Perforce Software, Inc.
Reverse Connection Flow

1. tvconnect writes request
2. TotalView UI reads request
3. TotalView returns response
4. tvconnect reads response
5. exec
6. socket connection opened

FRONT-END NODE

BATCH NODE

$HOME/.totalview/connect

COMPUTE NODES

Rank 0
Reverse Connection Flow

1. tvconnect writes request
2. TotalView UI reads request
3. TotalView returns response
4. tvconnect reads response
5. exec
6. socket connection opened
Batch Script Submission 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
```
Reverse Connect Demo

- TotalView Reverse Connect Demo
Remote Debugging at NERSC
TotalView Debugging at NERSC

• Interactive Debugging on Allocated Nodes
  • Allocate one or more nodes
  • Use any of the remote debugging techniques

• Batch
  • Utilize Reverse Connect (tvconnect) to connect back to a “waiting” TotalView
  • TotalView can be run within NoMachine, VNC, or on laptop with Remote UI
Debugging on Cori

- **Things To Know**
  - Environment variable “TVD_DISABLE_CRAY” must **not** be set on Cori, but it must be set on Perlmutter
    - “module load totalview” does the right thing on each system
      - **Cori:** Does not set TVD_DISABLE_CRAY; TotalView uses MRNet/CTI (Cray Tools Interface)
      - **Perlmutter:** Does set TVD_DISABLE_CRAY=1; TotalView uses MRNet/SSH (as a temporary workaround for a CTI/SLURM/GPU problem)

- **Starting TotalView on Cori**
  - `module load totalview`
  - `cd <your programs directory>`
  - `salloc -N 1 -t 60 -q interactive -C haswell`
  - `srun -n 8 ./<your application>`
  - `totalview -args srun -n 8 <your application>`
Startup
The Start Page is the place to start new debugging sessions, restore recent sessions and learn about the latest new TotalView features.
Debugging a New Program

<table>
<thead>
<tr>
<th>Program Session</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Session Details</strong></td>
<td><strong>Program Details</strong></td>
</tr>
<tr>
<td><strong>Session Name</strong></td>
<td><strong>File Name</strong></td>
</tr>
<tr>
<td>Enter or select a session name, e.g. myprogram with ReplayEngine...</td>
<td>Enter program path and name, e.g. /home/smith/myprogram...</td>
</tr>
<tr>
<td><strong>Debug Options</strong></td>
<td><strong>Arguments</strong></td>
</tr>
<tr>
<td><strong>Reverse Debugging</strong></td>
<td>Enter any program arguments. Ex. -option foo</td>
</tr>
<tr>
<td>Enable reverse debugging with ReplayEngine</td>
<td></td>
</tr>
<tr>
<td><strong>Python Debugging</strong></td>
<td></td>
</tr>
<tr>
<td>Enable call stack filtering for Python</td>
<td></td>
</tr>
<tr>
<td><strong>Memory Debugging</strong></td>
<td></td>
</tr>
<tr>
<td>Enable memory debugging</td>
<td></td>
</tr>
<tr>
<td><strong>Standard Input Redirection</strong></td>
<td><strong>Program Environment</strong></td>
</tr>
<tr>
<td><strong>Redirect standard input from file</strong></td>
<td>Environment variables for the program</td>
</tr>
<tr>
<td>Enter input file path and name</td>
<td>Enter line-separated NAME=VALUE pairs</td>
</tr>
<tr>
<td><strong>Standard Output/Error Redirection</strong></td>
<td></td>
</tr>
</tbody>
</table>
Debugging a Parallel Program

Session Editor

Parallel Session

Session Details

Session Name
[Enter or select a session name, e.g. myprogram with ReplayEngine...]

Parallel Details

Parallel System
[Select your parallel system]

Tasks:
[Enter the number of tasks]

Nodes:
[Enter the number of nodes]

Additional Starter Arguments
[Enter starter arguments as needed]

Standard Input Redirection

Program Details

File Name
[Enter program path and name, e.g. /home/smith/myprog]

Arguments
[Enter any program arguments, Ex. -option foo]

Debug Options

Reverse Debugging
Enable reverse debugging with ReplayEngine

Python Debugging
Enable call stack filtering for Python

Memory Debugging
Enable memory debugging

Program Environment

RESET LOAD SESSION CANCEL
Attach to a Running Program

**TotalView Power Tip**

- Launch a remote TotalView debug server to attach to programs on a remote system:
  
  `totalview executable -r hostname[:port]`

- This will be available in the UI in a coming release.
Open a Core File or Replay Recording Session

Session Editor

Core or Replay Recording Session

Session Details

Session Name

Core or Replay Recording File

Core or Replay Recording File Name

Program Details

File Name

BROWSE...

BROWSE...

RESET  LOAD SESSION  CANCEL
Starting a Previous Debugging Session

What do you want to do today?

- Debug a Program
- Debug a Parallel Program
- Attach To Process
- Load Core or Replay Recording File
- Listen For Reverse Connections
- Launch Remote Debugger

Recent Sessions

- python
  Last run on Jan 06, 2021
- ReplayEngine_demo
  Last run on Nov 19, 2020
- ReplayEngine_demo
  Last run on Apr 07, 2020
- ThreadWorkers
  Last run on Feb 04, 2020

Help/Support

Support
Find how to contact us at our Support Center.
https://totalview.io/support

What's New

New in TotalView 2020.3  November 2020
UI Navigation and Process Control
TotalView’s Default Views

Processes & Threads View
Lookup File or Function
Documents
Source View
Call Stack View
Local Variables View
Data View, Command Line, Input/Output
Action Points, Replay Bookmarks
### Process and Threads View

<table>
<thead>
<tr>
<th>Description</th>
<th>#P</th>
<th>#T</th>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>tx_fork_loop(S3)</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>snore</td>
<td>4</td>
<td>4</td>
<td>p2.1, p4.1, p3.2,...</td>
</tr>
<tr>
<td>tx_fork_loop.cxx</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>Stopped</td>
<td>4</td>
<td>8</td>
<td>p1.1, p3.1, p1.2,...</td>
</tr>
<tr>
<td>__select_acancel</td>
<td>2</td>
<td>3</td>
<td>p1.2, p2.3</td>
</tr>
<tr>
<td>&lt;unknown line&gt;</td>
<td>2</td>
<td>3</td>
<td>p1.2, p2.3</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>p2.3</td>
</tr>
<tr>
<td>snore</td>
<td>3</td>
<td>5</td>
<td>p1.1, p3.1, p4.2,...</td>
</tr>
<tr>
<td>tx_fork_loop.cxx</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
- State Group
- Hostname
- Process State
- Thread State
- Function
- Source Line
- PC
- Action Point ID
- Stop Reason
- Process ID
- Thread ID
- Process Held
- Thread Held
- Replay Mode
#ifdef

if (whoops)
{
    printf("pthread_create failed; result=%d, errno=%d\n", whoops, errno);
    exit(1);
}

thread_ptids[total_threads++] = new_tid;
printf("%d: Spun off %ld.\n", (int)getpid(), (long)new_tid);

} /* fork_count */ /* never returns */

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

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

#ifndef __linux
    /* the Linux implementation of pthreads uses these signals, so we'd better not */
#endif

    /* Source View */
Call Stack View and Local Variables View

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

Local Variables View:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>int</td>
<td>0x00000012 (18)</td>
</tr>
<tr>
<td>c</td>
<td>int</td>
<td>0x00000014 (20)</td>
</tr>
<tr>
<td>i</td>
<td>int</td>
<td>0x00000000 (0)</td>
</tr>
<tr>
<td>v</td>
<td>int[20]</td>
<td>(int[20])</td>
</tr>
<tr>
<td>[0]</td>
<td>int</td>
<td>0x00000000 (0)</td>
</tr>
<tr>
<td>[1]</td>
<td>int</td>
<td>0x00000000 (0)</td>
</tr>
<tr>
<td>[2]</td>
<td>int</td>
<td>0x00000000 (0)</td>
</tr>
<tr>
<td>[3]</td>
<td>int</td>
<td>0x00000000 (0)</td>
</tr>
<tr>
<td>[4]</td>
<td>int</td>
<td>0x00000000 (0)</td>
</tr>
<tr>
<td>[5]</td>
<td>int</td>
<td>0x00000000 (0)</td>
</tr>
<tr>
<td>[6]</td>
<td>int</td>
<td>0x00000000 (0)</td>
</tr>
<tr>
<td>[7]</td>
<td>int</td>
<td>0x00000000 (0)</td>
</tr>
<tr>
<td>[8]</td>
<td>int</td>
<td>0x00000000 (0)</td>
</tr>
</tbody>
</table>
# 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>✓</td>
<td>1</td>
<td></td>
<td>.../ReplayEngine-demo.cxx#27</td>
<td>ReplayEngine_dmc.cxx (line 27)</td>
<td>main</td>
</tr>
<tr>
<td>✓</td>
<td>2</td>
<td></td>
<td>4 bytes @ 0x601058 (arraylength)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data View, Command Line View and Input/Output View

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Thread ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>int</td>
<td>11</td>
<td>0x00000014 (20)</td>
</tr>
<tr>
<td>p</td>
<td>int*</td>
<td>11</td>
<td>0x7ffe86f00 -&gt; 0x00000014 (20)</td>
</tr>
</tbody>
</table>

**Command Line View**

```
Linux x86_04 Totalview 2020.3.11
Created process 1 (3892), named "ReplayEngine_demo"
Thread 1.1 has appeared
Thread 1.1 hit breakpoint 1 at line 27 in "main"
Thread 1.1 relieved
Thread 1.1 hit breakpoint 1 at line 33 in "main"
Thread 1.1 relieved
Thread 1.1 hit breakpoint 1 at line 31 in "main"
```

**Input/Output View**

```
Flip this text
\* Flip this text
Flip some more text
\* Flip some more text
```

**Standard Input:**

<table>
<thead>
<tr>
<th>SEND</th>
</tr>
</thead>
</table>
Lookup File or Function
Preferences

File > Preferences Menu

Or

"Gear" Toolbar Item

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.

- New Interface
  - Modern, dockable style user interface with improved low to medium scale multi-process and multi-thread dynamic analysis and debugging.

- Classic Interface
  - Traditional, dedicated window for very high-scale multi-process dynamic analysis and debugging.

Font Size

Choose the font size for the user interface.

Display Settings changes will take effect the next time the product is started.
### TotalView Toolbar

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go</td>
<td>Sets the thread to running until it reaches a stopping point. Often this will be a breakpoint that you have set, but the thread could stop for other reasons.</td>
</tr>
<tr>
<td>Halt</td>
<td>Stops the thread at its current execution point.</td>
</tr>
<tr>
<td>Kill</td>
<td>Stops program execution. Existing breakpoints and other settings remain in effect.</td>
</tr>
<tr>
<td>Restart</td>
<td>Stops program execution and restarts the program from the beginning. Existing breakpoints and other settings remain in effect. This is the same as clicking Kill followed by Go.</td>
</tr>
<tr>
<td>Next</td>
<td>Moves the thread to the next line of execution. If the line the thread was on includes one or more function calls, TotalView does not step into these functions but just executes them and returns.</td>
</tr>
<tr>
<td>Step</td>
<td>Like Next, except that TotalView does step into any function calls, so the thread stops at the first line of execution of the first function call.</td>
</tr>
<tr>
<td>Out</td>
<td>If the thread is in a block of execution, runs the thread to the first line of execution beyond that block.</td>
</tr>
<tr>
<td>Run To</td>
<td>If there is a code line selected in one of the Source views, the thread will stop at this line, assuming of course that it ever makes it there. This operates like a one-time, temporary breakpoint.</td>
</tr>
</tbody>
</table>
Stepping Commands

Select Step ▼ in the toolbar. TotalView stops the program just before the first executable statement, the call to setjmp (context);
Stepping Commands

Select Next to advance to the while loop on line 31, and then select Step to step into the readexpr() function. (Next would step over, or execute it).
Using Set PC

- Resumes execution from an arbitrary point
- Select the line
- Thread->Set PC
Demo

- TotalView threads demo (txdining)
Action Points
Action Points

- **Breakpoint**
- **Evaluation Point (Evalpoint)**
- **Watchpoint**
- **Barrierpoint**
Setting Breakpoints

- Setting action points
  - Single-click line number
  - Right clicking on the line number and using the context menu
  - Clicking a line in the source view then selecting the Action Points -> Set breakpoint menu option
Setting Breakpoints

- Breakpoint->At Location...
  - Specify function name or line number
  - If function name, TotalView sets a breakpoint at first executable line in the function
Pending Breakpoints

- Useful when setting a breakpoint on a library that has not yet been loaded into memory
Modifying Breakpoints

- **Enable / Disable / Delete a breakpoint**
- **Adjust the breakpoints width**

- **Group**: Stops all running threads in all processes in the group.
- **Process**: Stops all the running threads in the process containing the thread that hit the breakpoint.
- **Thread**: Stops only the thread that first executes to this breakpoint.
Evalpoints

Create Evaluation Point

Evaluate this expression at location:  `tx_basic_data.cxx#48`

```cpp
if (i == 3) {
    $stop$
}
```

Enter an expression, for example:  `if (i == 20) $stop`

**Language:** C++
Evalpoints

- Use Eval points to:
  - Include instructions that stop a process and its relatives
  - Test potential fixes or patches for your program
  - Include a goto for C or Fortran that transfers control to a line number in your program
  - Execute a TotalView function
  - Set the values of your program’s variables
Evalpoints Examples

• Print the value of a variable to the command line
  
  ```c
  printf("The value of result is %d\n", result);
  ```

• Skip some code
  
  ```c
  goto 63;
  ```

• Stop a loop after a certain number of iterations
  
  ```c
  if ( (i % 100) == 0) {
    printf("The value of i is %d\n", i);
    $stop;
  }
  ```

Watchpoints

- Watchpoints are set on a specific memory location
- Execution is stopped when the value stored in that memory location changes
- A breakpoint stops **before** an instruction executes. A watchpoint stops **after** an instruction executes
Using Watchpoint Expressions

• TotalView has two variables that are used exclusively with watchpoint expressions:
  • $oldval: The value of the memory locations before a change is made.
  • $newval: The value of the memory locations after a change is made.

• Example 1
  
  ```c
  if (iValue != 42 && iValue != 44) {
    iNewValue = $newval; iOldValue = $oldval; $stop;
  }
  ```

• Example 2
  
  ```c
  if ($oldval >= 0 && $newval < 0) $stop
  ```
Barrier Breakpoints

- Used to synchronize a group of threads or processes defined in the action point
- Threads or processes are held at barrier point until all threads or processes in the group arrive
- When all threads or processes arrive the barrier is satisfied and the threads or processes are released
Saving Breakpoints

From the Action Points menu select Save or Save As to save breakpoints
Turn on option to save action points on exit
Demo

- TotalView evaluation point demo (Combined)
Examining and Editing Data
Call Stack and Local Variables

Call Stack View
- Lists the set of call frames as the program calls from one function or method to another
- Filter button used to turn on or off filtering of frames.

Local Variables View
- Displays local variables relative to the current thread of interest and the selected stack frame
- Organized by arguments and blocks
- To edit values, add variable to the Data View
The Data View Panel

- Data View allows deeper exploration of data structures
- Edit data values
- Cast to new data types
- Add data to the Data View using the context menu or by dragging and dropping
The Data View Panel – Expanding Arrays and Structures

Select the right arrow to display the substructures in a complex variable

Any nested structures are displayed in the data view
The Data View Panel – Diving on Data

Dive on a single element to view individual data in the Data View

![Data View Panel Example]

- **myStruct2**: struct simple_struct
- **foo**: (struct simple_struct)
- **a**: char, value 'a' (0x61, or 97)
- **dynints**: int*, value 0x01fb010 -> 0x00000000 (0)
- **myStruct2.foo**: int[5], value (int[5])
The Data View – Dive in All

- Dive in All
  - Use Dive in All to easily see each member of a data structure from an array of structures
The Data View – Dive in New Data Window

- Dive in New Data Window
  - Use Dive in New Data Window add data structures to new Data Views for focused data debugging
The Data View Panel – Entering Expressions

Enter a new expression in the Data View panel to view that data

Type the expression in the [Add New expression] field

A new expression is added

Increment a variable
The Data View Panel – Dereferencing a Pointer

Dereferencing a pointer

When you dive on a variable, it is not dereferenced automatically

Double click in the Name column to make it editable and dereference the pointer

The Data View displays the variables value
The Data View Panel - Casting

Casting to another type

Cast a variable into an array by adding the array specifier

TotalView displays the array
Viewing Data in Fortran

The qualified subroutine name appears in the Call Stack view.

The qualified variable names appear in the Local Variable panel.
Fortran Common Blocks

For each common block defined in the scope of a subroutine or function, TotalView creates an entry in that function’s common block list.

The names of common block members have function scope, not global scope.

If you select the function in the Call Stack view, the common blocks and their variables appear in the Local Variables panel.
Fortran User-Defined Types

TotalView displays user-defined types in the Local Variables panel, which you can then add to the Data View for more detail.
Advanced C++ and Data Debugging
C++ Container Transformations

• TotalView transforms many of the C++ and STL containers including:
  • array, forward_list, tuple, map, set, vector and others.
Advanced C++ Support

- TotalView supports debugging the latest C++11/14/17 features including:
  - lambdas, transformations for smart pointers, auto types, R-Value references, range-based loops, strongly-typed enums, initializer lists, user defined literals

```cpp
#include <functional>
#include <vector>
#include <iostream>

double eval(std::function<double(double)> f, double x = 2.0){
    return f(x);
}

int main()
    // // one line lambdas
auto lambda1 = []{int a, float b) { return a < b; }
// two line lambda
auto lambda2 = []{int a, float & b)
    if (a < b)
        return 1;
    if (b>a)
        return -1;
    return 0;
};
bool b = lambda1(3, 3.14);
int l = lambda2(3, 3.14);
for (int i=0; i<10; i++)
    b = lambda1(i, 3.14+i);

std::function<double(double)> f0 = []{double x{ 
    return 1.0;
};
auto f1 = []{double x{
    return x;};
#define type(f)
        std::function<double(double)> f0, f1 []{double x{ 

```
Array Statistics

- Easily display a set of statistics for the filtered portion of your array

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>400</td>
</tr>
<tr>
<td>Zero Count</td>
<td>0</td>
</tr>
<tr>
<td>Sum</td>
<td>597909.794</td>
</tr>
<tr>
<td>Minimum</td>
<td>6.28304</td>
</tr>
<tr>
<td>Maximum</td>
<td>7273.40418</td>
</tr>
<tr>
<td>Median</td>
<td>915.75308</td>
</tr>
<tr>
<td>Mean</td>
<td>1494.774485</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>1566.2306267959</td>
</tr>
<tr>
<td>First Quartile</td>
<td>283.91487</td>
</tr>
<tr>
<td>Third Quartile</td>
<td>2259.14557</td>
</tr>
<tr>
<td>Lower Adjacent Value</td>
<td>6.28304</td>
</tr>
<tr>
<td>Upper Adjacent Value</td>
<td>5195.2867</td>
</tr>
<tr>
<td>Nan Count</td>
<td>0</td>
</tr>
<tr>
<td>Infinity Count</td>
<td>0</td>
</tr>
<tr>
<td>Denormalized Count</td>
<td>0</td>
</tr>
<tr>
<td>Checksum</td>
<td>45976</td>
</tr>
</tbody>
</table>
Demo

- TotalView STL types (Combined)
- TotalView dive in all demo (Combined)
Q&A
TotalView Reverse Debugging
Reverse debugging

• How do you isolate an intermittent failure?
  – Without TotalView
    • Set a breakpoint in code
    • Realize you ran past the problem
    • Re-load
    • Set breakpoint earlier
    • Hope it fails
    • Keep repeating
  – With TotalView
    • Set a breakpoint
    • Start recording
    • See failure
    • Run backwards/forwards in context of failing execution
  – Reverse Debugging
    • Re-creates the context when going backwards
    • Focus down to a specific problem area easily
    • Saves days in recreating a failure
Recording and Playback

- When ReplayEngine is saving state information, it is in **Record Mode**

- The saved state information is the program’s execution history

- You can save the execution history at any time and reload the recording when debugging the executable in a subsequent session

- Using a ReplayEngine command, either from the Toolbar or the CLI, shifts ReplayEngine into **ReplayMode**

- Debugging commands that do not work in ReplayMode include:
  - Changing a variable’s value
  - Functions that alter memory
  - Running threads asynchronously
Reverse Debugging Controls

- Run forward
- Run backwards
- Next forward over functions
- Next backwards over functions
- Step forward into functions
- Step backwards into functions
- Advance forward out of function call
- Advance backwards to calling function
- Advance forward to selected line
- Advance backward to selected line
- Advance to “live” session
- Create a bookmark at this point in recorded history
- Save the recorded session
Saving and Loading Execution History

- TotalView can save the current ReplayEngine execution history to file at any time

- The saved recording can be loaded into TotalView using any of the following:
  - At startup, using the same syntax as when opening a core file:
    totalview -newUI executable recording-file
  - On the Start Page view by selecting Load Core File or Replay Recording File
Replay Bookmarks

- Replay bookmarks mark a point in the execution of a program, allowing you to quickly jump back to that point in time.

Creating a Replay Bookmark

Activating a Replay Bookmark
Setting Preferences for ReplayEngine

- You can set the following preferences for ReplayEngine
  - the maximum amount of memory to allocate to ReplayEngine
  - The preferred behaviour when the memory limit is reached

- Setting the maximum amount of memory. The default value ‘0’ specifies to limit the maximum size by available memory only.
  
  \[
  \text{dset TV::replay\_history\_size value}
  \]
  
  \text{e.g. dset TV::replay\_history\_size 1024M}

- Setting the preferred behaviour. By default, the oldest history is discarded so that recording can continue
  
  \[
  \text{dset TV::replay\_history\_mode 1 (Discard oldest history and continue recording)}
  \]
  \[
  \text{dset TV::replay\_history\_mode 2 (Stop the process when the buffer is full)}
  \]
Demo

• TotalView ReplayEngine Demo

TotalView Power Tip

• When debugging an MPI application, set a breakpoint after MPI_Init and then turn on reverse debugging.
TotalView Memory Debugging
What is a Memory Bug?

- A Memory Bug is a mistake in the management of heap memory
  - Leaking: Failure to free memory
  - Dangling references: Failure to clear pointers
  - Failure to check for error conditions
- Memory Corruption
  - Writing to memory not allocated
  - Overrunning array bounds
TotalView Heap Interposition Agent (HIA) Technology

- Advantages of TotalView HIA Technology
  - Use it with your existing builds
    - No Source Code or Binary Instrumentation
  - Programs run nearly full speed
    - Low performance overhead
  - Low memory overhead
    - Efficient memory usage
Memory Debugging Features

TotalView Memory Debugging Features

• Leak detection
• Heap usage
• Dangling pointer detection

Coming Features

• View the heap
• Automatically detect allocation problems
• Memory Corruption Detection - Guard Blocks & Red Zones
• Memory Block Painting
• Memory Hoarding
• Memory Comparisons between processes
Memory Debugging in TotalView’s New UI

TotalView 2022.1 Features

• Leak detection
• Dangling pointer detection
• View the heap
• Automatically detect allocation problems

Coming Features

• Memory Corruption Detection - Guard Blocks & Red Zones
• Memory Block Painting
• Memory Hoarding
• Memory Comparisons between processes
Demo

- Memory Debugging Demo
Debugging Parallel Applications
Multi-Thread and Multi-Process Debugging

- TotalView provides the power to
  - Simultaneously debug many threads and processes in a single debugging session
  - Supports MPI, fork/exec, OpenMP, pthreads, std::thread, et al
  - Help locate deadlocks and race conditions
  - Understand complex applications utilizing threads

- By
  - Providing control of entire groups of processes, individual processes or even down to individual threads within a process
  - Enabling thread level breakpoints and barrier controls
  - Showing aggregated thread and process state display
Starting a Parallel Program Session from the UI

- From New Parallel Session page select:
  - MPI preference
  - Number of tasks
  - Number of nodes
  - Starter arguments
- Click Start Session to save and launch

TotalView Power Tip

- Launching a parallel job from the UI is ok for small scale or simple jobs.
- The recommended way is to launch through the command line (next slide).
Starting a Parallel Program Session from the Command Line

**General Command Line:** `totalview --args <starter> -n ## <partition> myprogram`

<table>
<thead>
<tr>
<th>MPI</th>
<th>Startup Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM PowerLE (@LLNL)</td>
<td><code>totalview --args lrun -n 16 myprog</code></td>
</tr>
<tr>
<td>Linux under SLURM</td>
<td><code>totalview --args srun -n 16 -p pdebug myprog</code></td>
</tr>
<tr>
<td>Open MPI / MPICH / Intel MPI</td>
<td><code>totalview --args mpirun -np 16 myprog</code></td>
</tr>
</tbody>
</table>

The order of arguments and executables differs between platforms.

Use of `tvconnect` can also simplify a parallel debugging session launch.
Parallel Debugging Group, Process and Thread Control

Select either
- Group
- Process
- Thread
Parallel Preferences

Attach Behavior controls if TotalView should attach to all of the processes, none or ask what to do.

After Attach Behavior controls if parallel job stops, runs or if TotalView should ask what to do.
Multi-Thread Debugging Techniques

• Multiple ID’s for threads
  • pthread library ID – Displayed by default in TotalView
  • OS Light Weight Process (LWP) ID
  • TotalView thread ID – ProcessID.ThreadID, e.g. 1.3

• Finding deadlocks due to mutex misuse
  • Utilize ReplayEngine/reverse debugging
  • Leverage watchpoints to find when mutex was acquired
  • Set the “Open process window at breakpoint” preference on the Action Points tab
  • To get LWP id, turn off TotalView user threads (-no_user_threads)
  • TotalView normally just displays the pthread ID
Multi-Thread Debugging Techniques

• Dealing with thread starvation
  • A tough problem to solve...
  • Utilize prior technique for watching when mutex’s are locked/unlocked
  • Leverage Evaluation Points and TotalView’s built-in Statements
    • $countthread expression
    • $holdthread
    • $stopthread
  • Halt the program during execution several times to see where execution is at in the Stack Trace
Multi-Process Debugging Techniques

• For high-scale debugging sessions, use command line launch of the parallel job instead of the Parallel Program Session in UI.
  • UI Parallel Program Session uses a flexible “bootstrap” parallel session mechanism for easy debug session setup but takes longer to launch.

• Enable reverse debugging on a per-process basis
  • Halt a specific process and enable reverse debugging on the fly

• Memory debugging can be enabled on one or more processes
Demo

- TotalView MPI Demo (mpi_array_broken)
Debugging NVIDIA GPUs and CUDA with TotalView
Introductions

• John DelSignore (TotalView Chief Architect)
  jdelsignore@perforce.com
• Scot Halverson (NVIDIA Solutions Architect)
  shalverson@nvidia.com
• Andrew Gontarek (NVIDIA Software Engineer – Devtools Compute Debugger)
  agontarek@nvidia.com
TotalView for the NVIDIA® GPU Accelerator

- NVIDIA Tesla, Fermi, Kepler, Pascal, Volta, Turing, Ampere
- NVIDIA Ampere cards are in testing
- NVIDIA CUDA 9.2, 10 and 11
  - With support for Unified Memory
- Debugging 64-bit CUDA programs
- Features and capabilities include
  - 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
  - CUDA device window reveals what is running where
  - Support for types and separate memory address spaces
  - Leverages CUDA memcheck
TotalView CUDA Debugging Model
GPU Memory Hierarchy

- Hierarchical memory
  - Local (thread)
    - Local
    - Register
  - Shared (block)
  - Global (GPU)
    - Global
    - Constant
    - Texture
  - System (host)
### Supported Type Storage Qualifiers

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@generic</td>
<td>An offset within generic storage</td>
</tr>
<tr>
<td>@frame</td>
<td>An offset within frame storage</td>
</tr>
<tr>
<td>@global</td>
<td>An offset within global storage</td>
</tr>
<tr>
<td>@local</td>
<td>An offset within local storage</td>
</tr>
<tr>
<td>@parameter</td>
<td>An offset within parameter storage</td>
</tr>
<tr>
<td>@iparam</td>
<td>Input parameter</td>
</tr>
<tr>
<td>@oparam</td>
<td>Output parameter</td>
</tr>
<tr>
<td>@shared</td>
<td>An offset within shared storage</td>
</tr>
<tr>
<td>@surface</td>
<td>An offset within surface storage</td>
</tr>
<tr>
<td>@tексampler</td>
<td>An offset within texture sampler storage</td>
</tr>
<tr>
<td>@texture</td>
<td>An offset within texture storage</td>
</tr>
<tr>
<td>@rtvar</td>
<td>Built-in runtime variables</td>
</tr>
<tr>
<td>@register</td>
<td>A PTX register name</td>
</tr>
<tr>
<td>@sregister</td>
<td>A PTX special register name</td>
</tr>
</tbody>
</table>
Control of Threads and Warps

- Warps advance synchronously
  - They share a PC

- Single step operation advances all GPU threads in the same warp

- Stepping over a `__syncthreads()` call will advance all relevant threads

- To advance more than one warp
  - Continue, possibly after setting a new breakpoint
  - Select a line and “Run To”

NVIDIA GPU and CUDA Parallelization

- CUDA uses the single instruction multiple thread (SIMT) model of parallelization.

- CUDA GPUs made up of many computing units called cores
  - Cores includes an arithmetic logic unit (ALU) and a floating-point unit (FPU).

- Cores collected into groups called streaming multiprocessors (SMs).

- Computing tasks are parallelized by breaking them into numerous subtasks called threads.

- Threads are organized into blocks.

- Blocks are divided into warps whose size matches the number of cores in an SM.

- Each warp gets assigned to a particular SM for execution. GPUs have one or more SMs.

- SM control unit directs each of its cores to execute the same instructions simultaneously for each thread in the assigned warp.
Compiling for CUDA debugging

When compiling an NVIDIA CUDA program for debugging, it is necessary to pass the `-g -G` options to the `nvcc` compiler driver. These options disable most compiler optimization and include symbolic debugging information in the driver executable file, making it possible to debug the application.

```bash
% /usr/local/bin/nvcc -g -G -c tx_cuda_matmul.cu -o tx_cuda_matmul.o
% /usr/local/bin/nvcc -g -G -Xlinker=-R/usr/local/cuda/lib64 \ tx_cuda_matmul.o -o tx_cuda_matmul
% ./tx_cuda_matmul
A:
[ 0][ 0] 0.000000
...output deleted for brevity...
[ 1][ 1] 131.000000
```
Compiling for a specific GPU architecture (avoids JIT’ing from PTX)

**Compiling for Ampere**
- `gencode arch=compute_80,code=sm_80`

**Compiling for Volta**
- `gencode arch=compute_70,code=sm_70`

**Compiling for Pascal**
- `gencode arch=compute_60,code=sm_60`

**Compiling for Kepler**
- `gencode arch=compute_35,code=sm_35`

**Compiling for Fermi and Tesla**
- `gencode arch=compute_20,code=sm_20`
- `gencode arch=compute_10,code=sm_10`

**Compiling for Fermi**
- `gencode arch=compute_20,code=sm_20`
A TotalView Session with CUDA

A standard TotalView installation supports debugging CUDA applications running on both the host and GPU processors.

TotalView dynamically detects a CUDA install on your system. To start the TotalView GUI or CLI, provide the name of your CUDA host executable to the totalview or totalviewcli command.

For example, to start the TotalView GUI on the sample program, use the following command:

```
% totalview tx_cuda_matmul
```

* This example is just a single node, no MPI application
Source View Opened on CUDA host code
Set Breakpoints in CUDA Kernel Code Before Launch

Set breakpoints in the CUDA or OpenMP TARGET region code before you start the process.

Hollow breakpoint indicates a breakpoint will be set when the code is loaded onto the GPU.
Stopped at a Breakpoint in CUDA Kernel Code

- Bold line numbers indicate source code lines where the compiler generated code, which are good places to set breakpoints.
CUDA thread IDs and Coordinate Spaces

- Host thread IDs have a positive thread ID (p1.1)
- CUDA thread IDs have a negative thread ID (p1.-1)
GPU Physical and Logical Focus Toolbars

**Logical** toolbar displays the Block and Thread coordinates.

**Physical** toolbar displays the Device number, Streaming Multiprocessor, Warp and Lane.

To view a CUDA host thread, select a thread with a positive thread ID in the Process and Threads view.

To view a CUDA GPU thread, select a thread with a negative thread ID, then use the GPU focus controls in the logical or physical toolbar to focus on a specific GPU thread or lane.
The identifier @local is a TotalView built-in type storage qualifier that tells the debugger the storage kind of "A" is local storage.

The debugger uses the storage qualifier to determine how to locate A in device memory.

"elements" is a pointer to a float in @generic storage.
Stepping GPU Code

- Single-step operations advance all the GPU hardware lanes in the same warp
- To advance the execution of more than one warp, you may either:
  - Set a breakpoint and continue the process, or
  - Select a line number in the source pane and select “Run To”.

Select the line to run to and then click “Run To” in the toolbar.
GPU Status View

Displays the state of all the GPUs being debugged.

Fully configurable to allow aggregating, sorting and filtering based on physical or logical attributes.
Enabling CUDA Memory Checker Feature

From the Program Session Dialog

From the Debug Menu
Debugging on Perlmutter
Debugging on Perlmutter (Things to Know)

- If you bind processes to GPUs using `srun`, the debugger cannot determine which GPUs the processes are using
  - Workaround – Do not use the “--gpu-bind” option when debugging
  - Perforce is working with HPE, NVIDIA, and SCHEDMD on a solution

- Watchpoints in GPU memory are not support on NVIDIA GPUs, but CPU watchpoints are supported

- On Perlmutter (not Cori), the environment variable “TVD_DISABLE_CRAY=1” must be set to disable using Cray CTI
  - “module load totalview” should set TVD_DISABLE_CRAY=1 on Perlmutter (but not on Cori)
  - SSH is used to instantiate the TV/MRNet tree
  - Using CTI requires a change to SLURM that allows tool/application processes to overlap GPU access
    - SLURM 22.05 or later (Perlmutter is still on 21.08)
    - MRNet/CTI version that drives SLURM 22.05 properly
  - Requires passwordless SSH between nodes
    - However, as of 9/28/22, passwordless SSH was not working on some Perlmutter nodes (use one node as a workaround)
    - NERSC is working on a fix
Using SSH to between NERSC nodes can generate a lot of terminal output

- Each SSH generates a long “NOTICE TO USERS” message

- The messages can be suppressed by adding the following lines to your “$HOST/.ssh/config” file:

  ```
  # The "LogLevel quiet" option stops the "NOTICE TO USERS" messages
  Host *
      LogLevel quiet
  ```

- The above is not necessary, but it does reduce terminal output
Debugging on Perlmutter (Supported Start-ups)

• TotalView supports **interactive** and **batch** debugging sessions

  • **Interactive** debugging sessions
    • Use `salloc` to allocate interactive nodes
    • Start TotalView on `srun` within the allocation
    • Allows restarting `srun` multiple times within the same allocation

  • **Batch** debugging sessions
    • Use `sbatch` to submit a batch job
    • Batch script uses `tvconnect srun` ... to request a “reverse connect” to TotalView
    • Start TotalView on a login node and accept the “reverse connect” request
    • To restart `srun` multiple times, invoke `tvconnect srun` in a loop in the script
Debugging on Perlmutter (Interactive Start-up)

- Load the “totalview” module
  ```bash
  module load totalview
  ```
- Allocate some nodes, for example
  ```bash
  salloc -A nvendor -C gpu -N 2 -G 8 -t 60 -q interactive_sss1
  ```
- An interactive shell (bash, csh, etc.) will start inside the allocation
- Start `totalview` on `srun`, for example
  ```bash
  totalview --args srun -n 8 -G 8 -c 32 --cpu-bind=cores ./b.out
  ```
  - Remember, “--gpu-bind” does not currently work, so do not use it while debugging
Debugging on Perlmutter (Batch Start-up)

• Example batch script using tvconnect

```bash
#!/bin/bash -x
#SBATCH -A nvendor
#SBATCH -C gpu
#SBATCH -N 2
#SBATCH -G 8
#SBATCH -t 30
#SBATCH --qos=debug

module load totalview
tvconnect srun b.out
```

• When the batch script starts, tvconnect blocks until a totalview accepts the reverse connect request

• On the login node, load the “totalview” module and start totalview

```bash
module load totalview
totalview
```
Debugging on Perlmutter (Batch Start-up)

- TotalView will “Listen For Reverse Connections” by default, but make sure the option is enabled

- When the batch script executes the `tvconnect` command, TotalView will post a dialog

- Select “Yes” to connect TotalView to the batch job
Debugging on Perlmutter (Common to Interactive/Batch)

- Once TotalView starts-up on `srun`, the following steps are common to interactive / batch debugging

  - Typically
    - Select “Go” to start `srun`
    - `srun` will launch the parallel program
    - TotalView detects the parallel program launch and attaches to the MPI processes

- When the jobs goes parallel, TotalView will post a dialog
Stop the job when it goes parallel?

- Click “Yes” to stop the parallel job, which is useful if you want to
  - Navigate to source files / functions
  - Set breakpoints
- Click “No” to allow the job to run, which is useful if you
  - Had saved breakpoints from a previous session
  - Know the program is going to crash (SEGV, etc.)
TotalView will focus on `main()` in rank 0
Navigate to a file or function you want to debug

```c
#include <stdlib.h>

//void vecAdd_wrapper(void);
void vecAdd_wrapper(int, int);

volatile int spinner = 0;
int main( int argc, char* argv[] )
{
    int rank, np = argc;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    vecAdd_wrapper(rank, np);
    if (argc > 1) {
        spinner = 0;
        if (spinner > argc) {
            printf("Sleeping %d seconds...
", spinner);
            fflush(stdout);
            while (spinner--)
                sleep(1);
        }
        MPI_Finalize();
    }
    if (spinner == argc) {
        printf("%s
", argv);
        Sleep(spinner);
    }
    MPI_Finalize();
}```
Find the CUDA kernel and select a line number to plant a breakpoint

- Line numbers indicate if there’s code at that line
  - Pale line numbers indicate no code (yet)
  - Bold line numbers indicate code
- CUDA code is dynamically loaded at runtime, so TotalView does not have any debug information until the CUDA kernel is launched
- Select a line number in the CUDA kernel that will have CUDA code loaded
  - Hollow breakpoint markers indicate no code yet
  - Solid breakpoint markers indicate code
- Source line information for a source file is unified for both GPU and CPU code
Click the “Go” button to run the application and launch the kernel.
Stopped at a breakpoint in the CUDA kernel
Source view stopped in a CUDA kernel

- Line number information for the GPU code is **unified** with the CPU code.
- The hollow breakpoint marker turns solid, indicating that there is now code at that line.
- The PC arrow and highlighted source line indicates where the warp is stopped.
GPU thread focus and navigation controls

- “GPU (Logical)” control displays and allows focusing on a specific Block and Thread

- “GPU (Physical)” control displays and allows focusing on a specific Device, SM, Warp, and Lane
CUDA stack backtrace and local variables

- Call Stack
- Open the drawer for details
- Local Variables
GPU Status view

- The “GPU Status” view displays an aggregated overview of one or more of the GPUs in the whole job, in a single process, or on a single GPU.

- The “GPU Status” view controls allow:
  - Selecting the set of properties to display
  - Aggregation by the selected properties
  - Sorting by the selected properties
  - Creating compound filters to include/exclude properties that are equal, not equal, greater, etc.

- Allows you to get a “big picture” of the state of one or more of the GPUs in your job.
Batch Debugging with TVScript
tvscript

- A straightforward language for unattended and/or batch debugging with TotalView and/or MemoryScape
- Usable whenever jobs need to be submitted or batched
- Can be used for automation
- A more powerful version of printf, no recompilation necessary between runs
- Schedule automated debug runs with `cron` jobs
- Expand its capabilities using TCL
tvscript

tvscript [options] [filename ] [ -a program_args]

**options**
TotalView and tvscript command-line options.

**filename**
The program being debugged.

- **-a program_args**
Program arguments.
tvscript

• All of the following information is provided by default for each print
  • Process id
  • Thread id
  • Rank
  • Timestamp
  • Event/Action description

• A single output file is written containing all of the information regardless of the number of processes/threads being debugged
## Supported tvscript events

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Event</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>General event</td>
<td>any_event</td>
<td>A generated event occurred.</td>
</tr>
<tr>
<td>Memory debugging event</td>
<td>addr_not_at_start</td>
<td>Program attempted to free a block using an incorrect address.</td>
</tr>
<tr>
<td></td>
<td>alloc_not_in_heap</td>
<td>The memory allocator returned a block not in the heap; the heap may be corrupt.</td>
</tr>
<tr>
<td></td>
<td>alloc_null</td>
<td>An allocation either failed or returned NULL; this usually means that the system is out of memory.</td>
</tr>
<tr>
<td></td>
<td>alloc_returned_bad_alignment</td>
<td>The memory allocator returned a misaligned block; the heap may be corrupt.</td>
</tr>
<tr>
<td></td>
<td>any_memory_event</td>
<td>A memory event occurred.</td>
</tr>
<tr>
<td></td>
<td>bad_alignment_argument</td>
<td>Program supplied an invalid alignment argument to the heap manager.</td>
</tr>
<tr>
<td></td>
<td>double_alloc</td>
<td>The memory allocator returned a block currently being used; the heap may be corrupt.</td>
</tr>
<tr>
<td></td>
<td>double_dealloc</td>
<td>Program attempted to free an already freed block.</td>
</tr>
<tr>
<td></td>
<td>free_not_allocated</td>
<td>Program attempted to free an address that is not in the heap.</td>
</tr>
<tr>
<td></td>
<td>guard_corruption</td>
<td>Program overwrote the guard areas around a block.</td>
</tr>
</tbody>
</table>
## Supported tvscript events

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Event</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code debugging event</td>
<td><code>hoard_low_memory_threshold</code></td>
<td>Hoard low memory threshold crossed.</td>
</tr>
<tr>
<td>Source code debugging event</td>
<td><code>realloc_not_allocated</code></td>
<td>Program attempted to reallocate an address that is not in the heap.</td>
</tr>
<tr>
<td>Source code debugging event</td>
<td><code>rz_overrun</code></td>
<td>Program attempted to access memory beyond the end of an allocated block.</td>
</tr>
<tr>
<td>Source code debugging event</td>
<td><code>rzUnderrun</code></td>
<td>Program attempted to access memory before the start of an allocated block.</td>
</tr>
<tr>
<td>Source code debugging event</td>
<td><code>rz_use_after_free</code></td>
<td>Program attempted to access a block of memory after it has been deallocated.</td>
</tr>
<tr>
<td>Source code debugging event</td>
<td><code>rz_use_after_free_overrun</code></td>
<td>Program attempted to access memory beyond the end of a deallocated block.</td>
</tr>
<tr>
<td>Source code debugging event</td>
<td><code>rz_use_after_free_underun</code></td>
<td>Program attempted to access memory before the start of a deallocated block.</td>
</tr>
<tr>
<td>Source code debugging event</td>
<td><code>termination_notification</code></td>
<td>The target is terminating.</td>
</tr>
<tr>
<td>Reverse debugging</td>
<td><code>actionpoint</code></td>
<td>A thread hit an action point.</td>
</tr>
<tr>
<td>Reverse debugging</td>
<td><code>error</code></td>
<td>An error occurred.</td>
</tr>
<tr>
<td>Reverse debugging</td>
<td><code>stopped_at_end</code></td>
<td>The program is stopped at the end of execution and is about to exit.</td>
</tr>
</tbody>
</table>
## Supported tvscript actions

<table>
<thead>
<tr>
<th>Action Type</th>
<th>Action</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory debugging actions</td>
<td>check_guard_blocks</td>
<td>Checks all guard blocks and write violations into the log file.</td>
</tr>
<tr>
<td></td>
<td>list_allocations</td>
<td>Writes a list of all memory allocations into the log file.</td>
</tr>
<tr>
<td></td>
<td>listLeaks</td>
<td>Writes a list of all memory leaks into the log file.</td>
</tr>
<tr>
<td></td>
<td>save_memory_debugging_file</td>
<td>Generates and saves a memory debugging file.</td>
</tr>
<tr>
<td></td>
<td>save_text_heap_status_source_view</td>
<td>Generates and saves a text version of the Heap Status Source View Report.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source code debugging actions</th>
<th>display_backtrace [level-level-num] [num_levels] [options]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-level-level-num sets the level at which information starts being logged.</td>
</tr>
<tr>
<td></td>
<td>num_levels restricts output to this number of levels in the call stack.</td>
</tr>
<tr>
<td></td>
<td>If you do not set a level, tvscript displays all levels in the call stack.</td>
</tr>
<tr>
<td></td>
<td>options is one or more of the following:</td>
</tr>
<tr>
<td></td>
<td>-no</td>
</tr>
<tr>
<td></td>
<td>-no</td>
</tr>
<tr>
<td></td>
<td>-no</td>
</tr>
<tr>
<td></td>
<td>-no</td>
</tr>
<tr>
<td></td>
<td>-no</td>
</tr>
<tr>
<td></td>
<td>-no</td>
</tr>
<tr>
<td></td>
<td>-no</td>
</tr>
</tbody>
</table>
## Supported tvscript actions

<table>
<thead>
<tr>
<th>Action Type</th>
<th>Action</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>print</strong> [-slice (slice_exp)] (variable\</td>
<td>exp)</td>
</tr>
<tr>
<td>Reverse debugging actions</td>
<td><strong>enable_reverse_debugging</strong></td>
<td>Turns on ReplayEngine reverse debugging and begins recording the execution of the program.</td>
</tr>
<tr>
<td></td>
<td><strong>save_replay_recording_file</strong></td>
<td>Saves a ReplayEngine recording file. The filename is of the form <code>&lt;ProcessName&gt;-&lt;PID&gt;-&lt;DATE&gt;.&lt;INDEX&gt;.recording.</code></td>
</tr>
</tbody>
</table>
tvscript examples

Simple example

```bash
tvscript\n-create_actionpoint "method1=>display_backtrace -show_arguments"\n-create_actionpoint "method2#37=>display_backtrace\n        -show_locals -level 1"\n-event_action "error=>display_backtrace -show_arguments\n        -show_locals"\n-display_specifiers "noshow_pid,noshow_tid"\n-maxruntime "00:00:30"\n~/work/filterapp /filterapp -a 20
```

MPI example

```bash
tvscript -mpi "Open MPI" -tasks 4 \n-create_actionpoint \n"hello.c#14=>display_backtrace"\n~/tests/MPI_hello
```
tvscript examples

Memory Debugging example

tvscript -maxruntime "00:00:30" \
-event_action "any_event=save_memory_debugging_file" \n-guard_blocks -hoard_freed_memory -detect_leaks \n~/work/filterapp -a 20

ReplayEngine example
tvscript \n-create_actionpoint "main=>enable_reverse_debugging" \n-event_action "stopped_at_end=>save_replay_recording_file" \nfilterapp
Demo

- TVScript demo (tvscript --script_file file tvscript_example.tvd ex2)
Python Debugging
Python in HPC

- Python development trends:
  - Increased usage of Python to build applications that call out to C++
  - Provides access to
    - High-performance routines
    - Leverage existing algorithms and libraries
    - Utilize advanced multi-threaded capabilities
  - Calling between languages easily enabled using technologies such as SWIG, ctypes, Pybind, Cython, CFFI, etc
  - Debugging mixed language applications is not easy
Python debugging with TotalView

• 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
  • Support for Python 2.7 and Python 3.5 and above

• What TotalView does not provide (yet):
  • Setting breakpoints and stepping within Python code

TotalView Power Tip

• Latest versions of Python 3.7 and 3.8 changed internal data structures, impacting TotalView’s ability to extract program state. An update will be available in an upcoming TotalView release.
Python without Filtering

Glue code
Python with filtering
Demo

- TotalView Python / C++ debugging demo (test_python_types.py)
Common TotalView Usage Hints
Common TotalView Usage Hints

- TotalView can’t find the program source
  - Did you compile with -g?
  - How to adjust the TotalView search paths? Preferences -> Search Path

- Python Debugging
  - Making sure proper system debug packages are installed for Python

- X11 forwarding performance
  - If users are forwarding X11 displays through ssh TotalView UI performance can be bad

- Understanding different ways to stop program execution with TotalView Action Points
  - Using a watchpoint on a local variable

- Focus
  - Diving on a variable that is no longer in scope. Check the Local Variables window for in scope variables
  - TotalView doesn’t change focus to the thread hitting a breakpoint. Set Action Point Preferences to “Automatically focus on threads/processes at breakpoint”
Common TotalView Usage Hints (cont.)

• MPI Debugging
  • Differences in launching MPI job from within the TotalView UI vs the command line.
  • TotalView runs an MPI program without stopping. Set the Parallel Preferences to “Ask What To Do” in After Attach Behavior
  • Using wrong attributes in processes and threads view

• Reverse Debugging
  • Running out of memory by not setting the maximum memory allocated to ReplayEngine
  • Defer turning on reverse debugging until later in program execution to avoid slow initialization phases
  • Adjust reverse debugging circular buffer size to reduce resources

• Memory Debugging
  • Starting with All memory debugging options enabled rather than Low
  • Not setting a size restriction for Red Zones
  • Issues with getting memory debugging turned on in an MPI job. May have to set LD_PRELOAD environment variable or worst case, prelink HIA
Common TotalView Usage Hints (cont.)

• Differences in functionality between new UI and classic UI
  • How to switch between them. Preferences -> Display or totalview –newUI and totalview -oldUI
  • Where the gaps still are in functionality

• Reverse Connect with tvconnect
  • When I use Reverse Connect I get the following obscure message: myProgram is an invalid or incompatible executable file format for the target platform
  • The message indicates an incompatible file format but most often this occurs if the program provided to tvconnect for TotalView to debug cannot be found. The easiest way to resolve problem is to provide the full path to the target application, e.g., tvconnect /home/usr/myProgram

• How do I get help?
  • How to submit a support ticket? techsupport@roguewave.com
  • Where is TV documentation (locally and on the internet). https://help.totalview.io/
  • Are there videos I can watch to learn how to use TotalView? https://totalview.io/support/video-tutorials
TotalView Resources and Documentation
TotalView Resources and Documentation

- TotalView website: [https://totalview.io](https://totalview.io)
- TotalView documentation:
  - [https://help.totalview.io](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](https://totalview.io/blog)
- Video Tutorials: [https://totalview.io/support/video-tutorials](https://totalview.io/support/video-tutorials)
Q&A
Contact us

- Bill Burns (Senior Director of Software Engineering and Product Manager)
  bburns@perforce.com
- John DelSignore (TotalView Chief Architect)
  jdelsignore@perforce.com
- Scot Halverson (NVIDIA Solutions Architect)
  shalverson@nvidia.com
- Peter Thompson (Senior Support Engineer)
  pthompson@perforce.com
- Bruce Ryan (Senior Account Executive)
  bryan@perforce.com
- Ken Hill (Senior Sales Engineer)
  khill@perforce.com