TotalView Training - NERSC

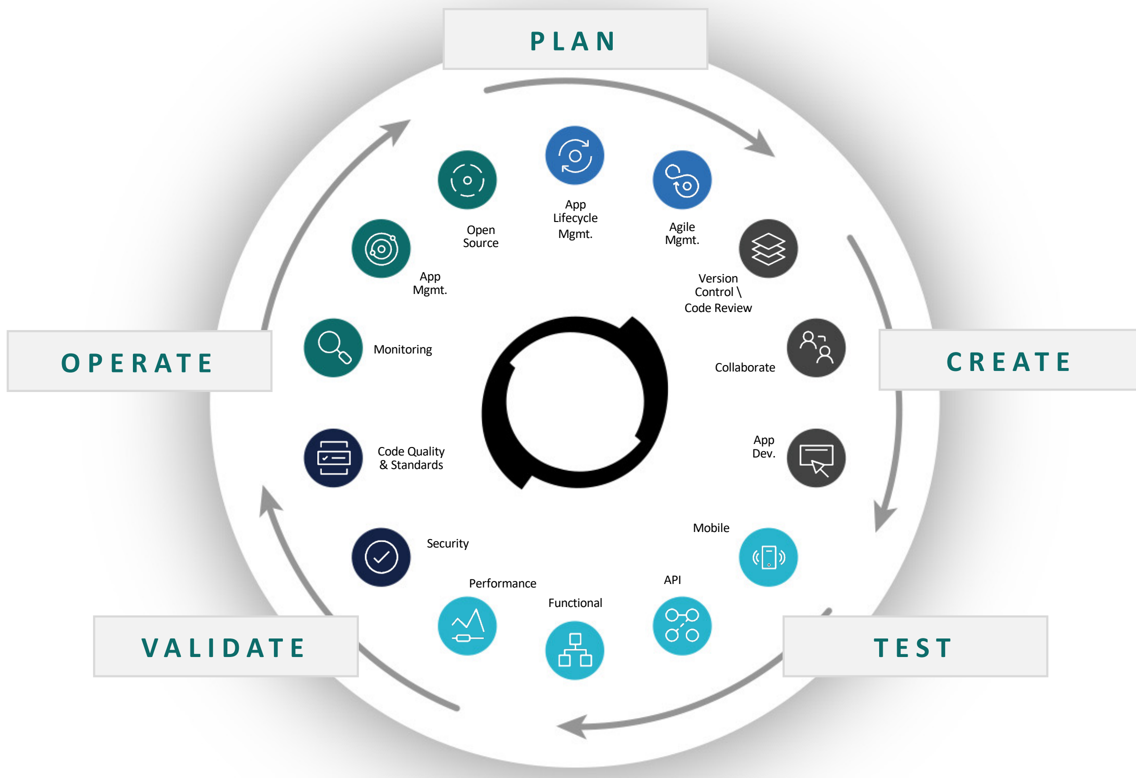
SEPTEMBER 29, 2022

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

PERFORCE

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

TotalView

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

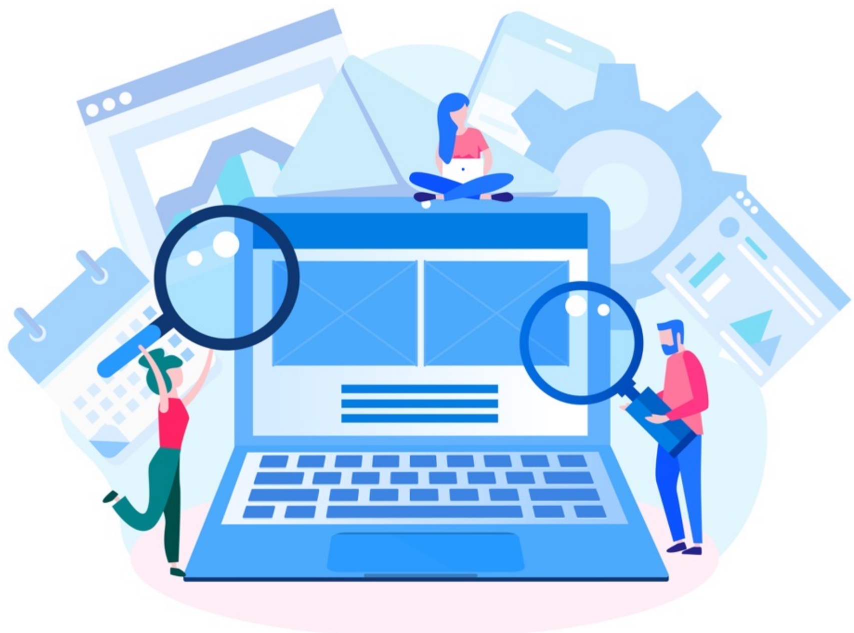
Notes

- * 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

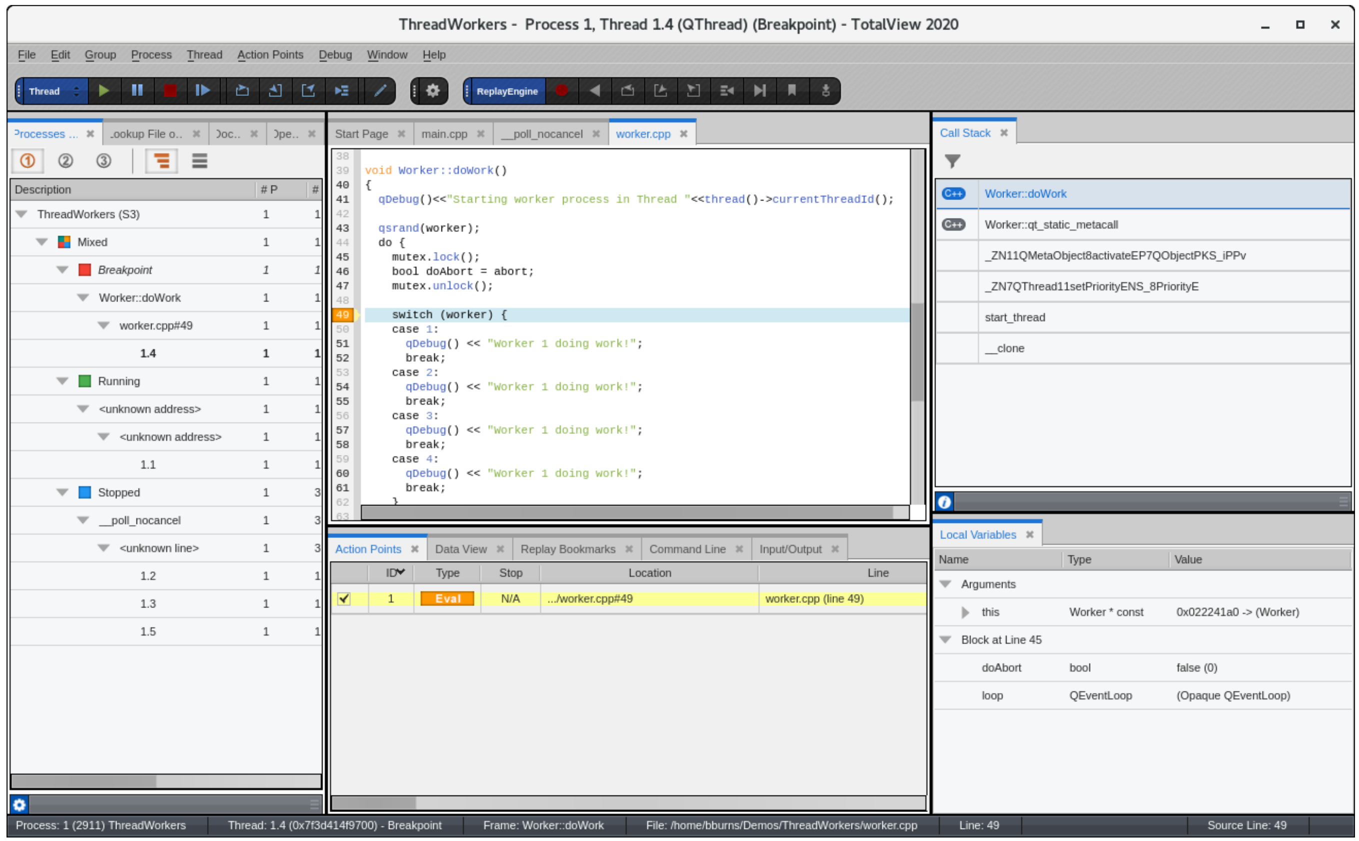
What is TotalView used for?

- 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

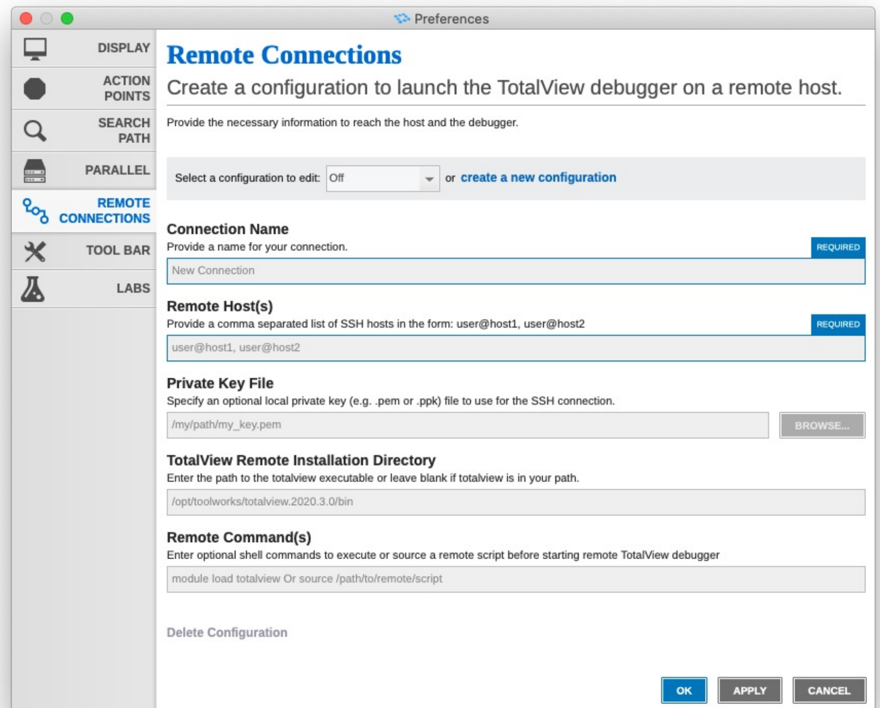
Remote Debugging Technologies

- 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

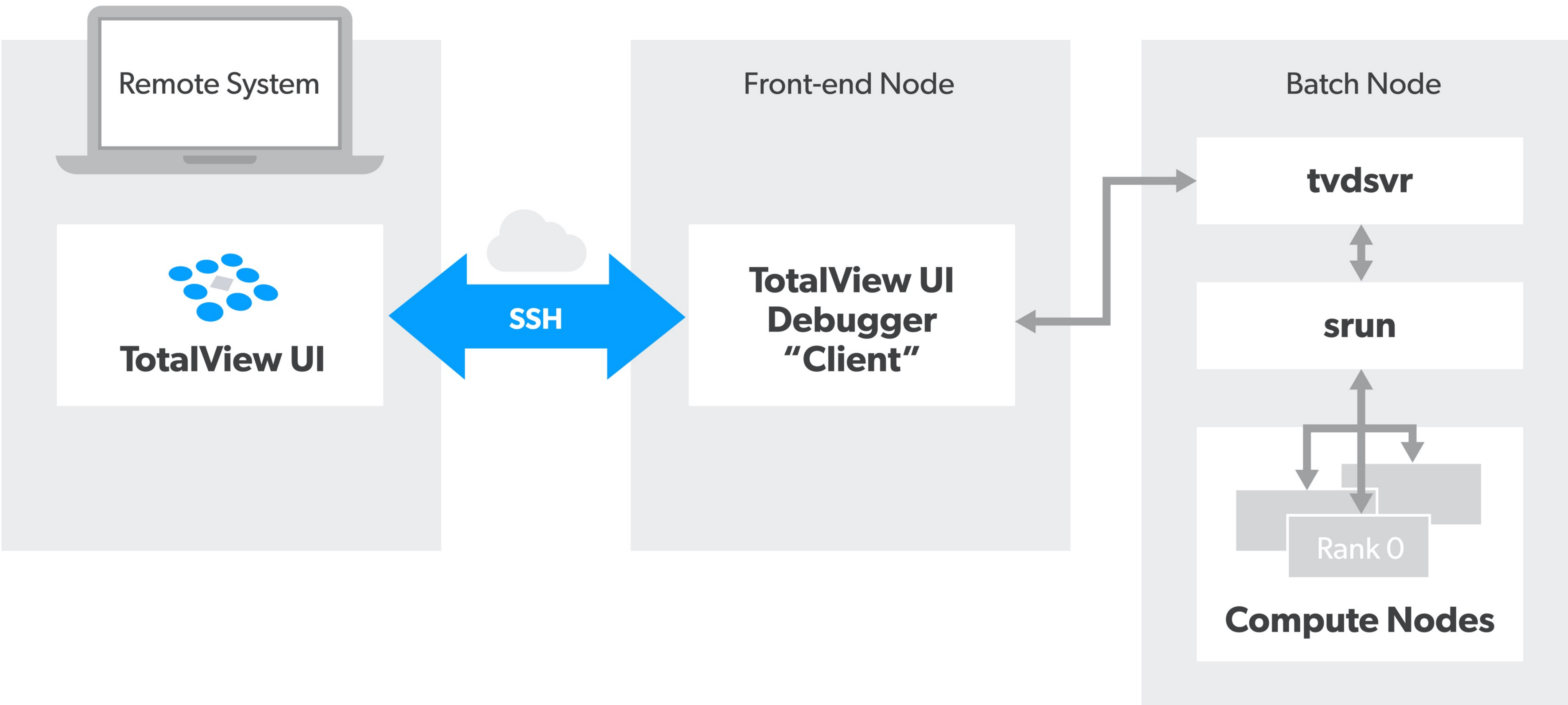
Remote Debugging - TotalView Remote UI

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.

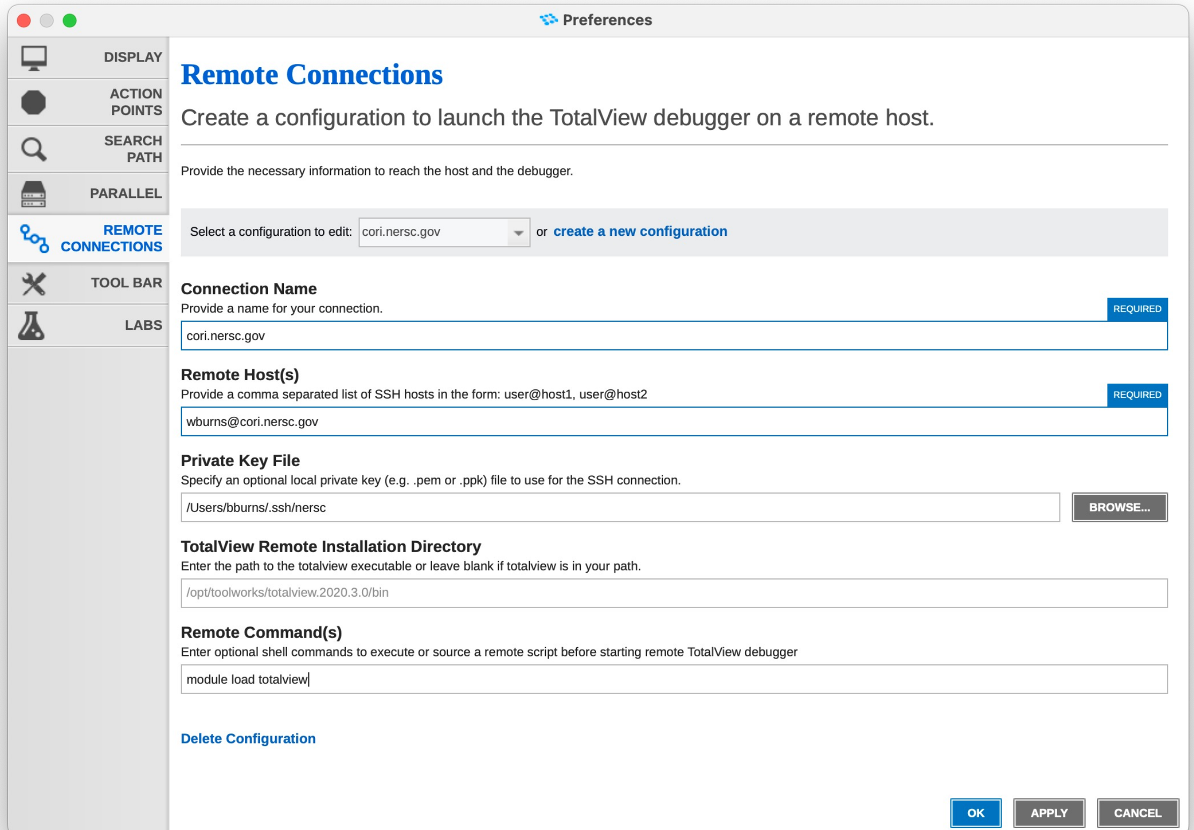


Remote UI Architecture



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

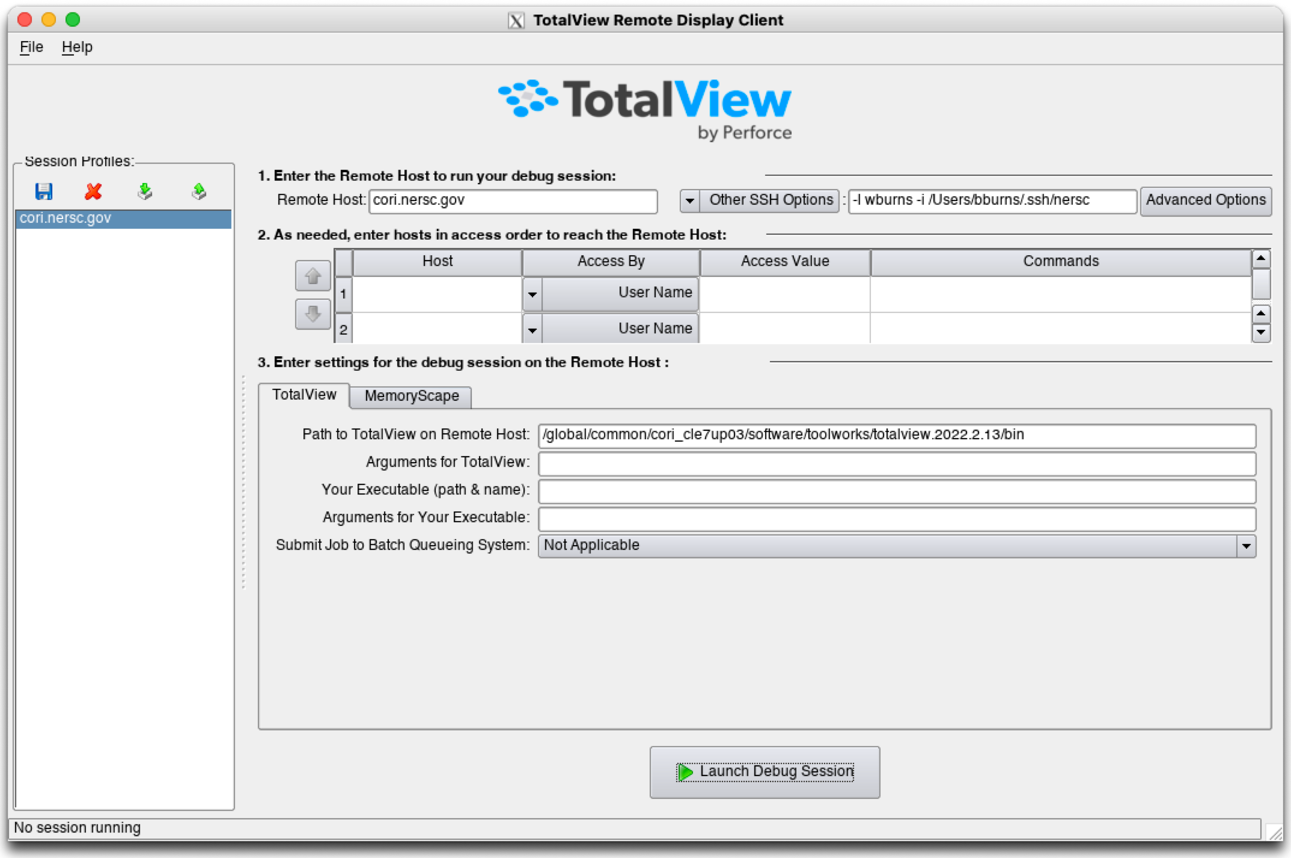
TotalView Remote Display Client

- 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

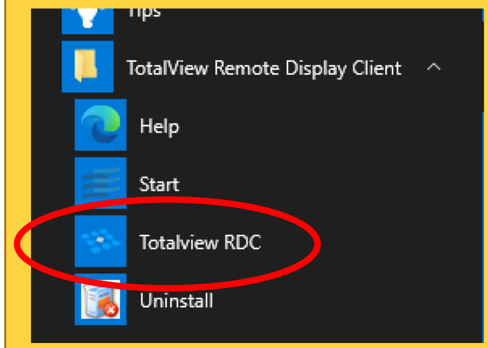
- Use NERSC sshproxy.sh to set up SSH Keys
- Example : Cori
- Configure RDC Session
  - Remote Host: **cori.nersc.gov**
  - Other SSH Options: **-l wburns -i /Users/bburns/.ssh/nersc**
  - Path to TotalView on Remote Host: **/global/common/cori_cle7up03/software/toolworks/totalview.2022.2.13/bin**
- 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

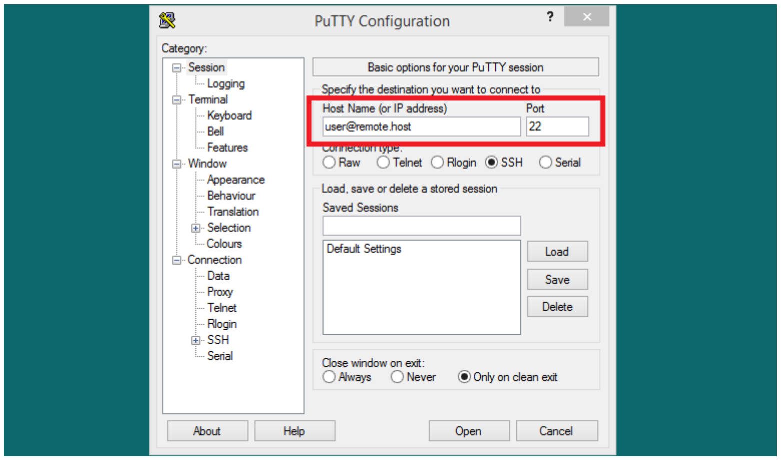
Debugging in a VNC Server Session

- 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

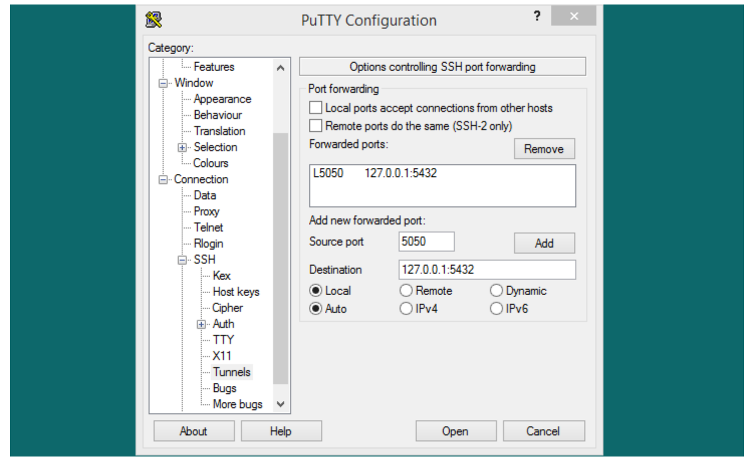
- Use NERSC sshproxy.sh to set up SSH Keys
- Example : Cori
- Create SSH tunnel for VNC session
  - Create a ”tunnel” for remote display information from port 5999 on the end system to port 5999 on your local system
  - `ssh -l wburns -i /Users/bburns/.ssh/nersc -L 5999:localhost:5999 cori.nersc.gov`
- 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.

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

TotalView Reverse Connections

TotalView Reverse Connections

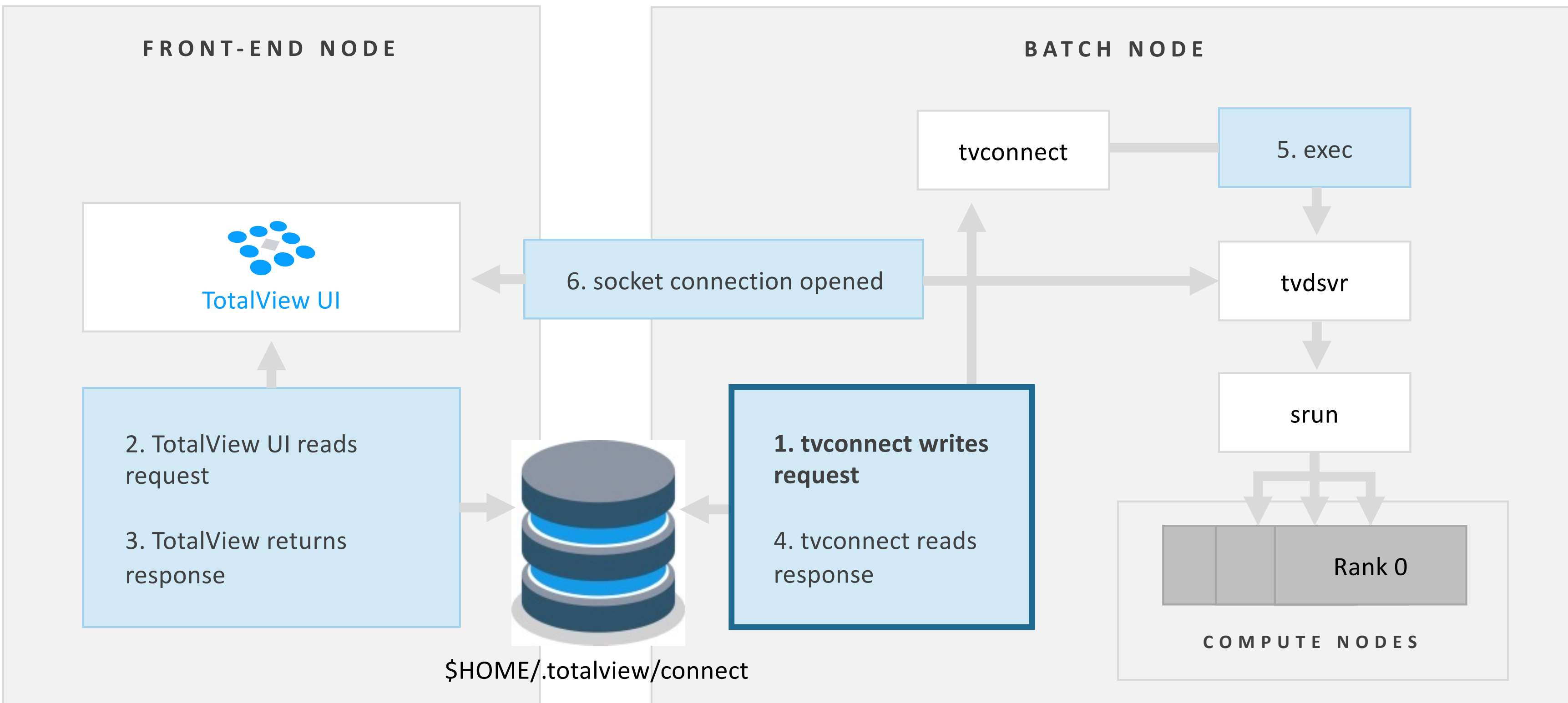
The Problem:

- 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

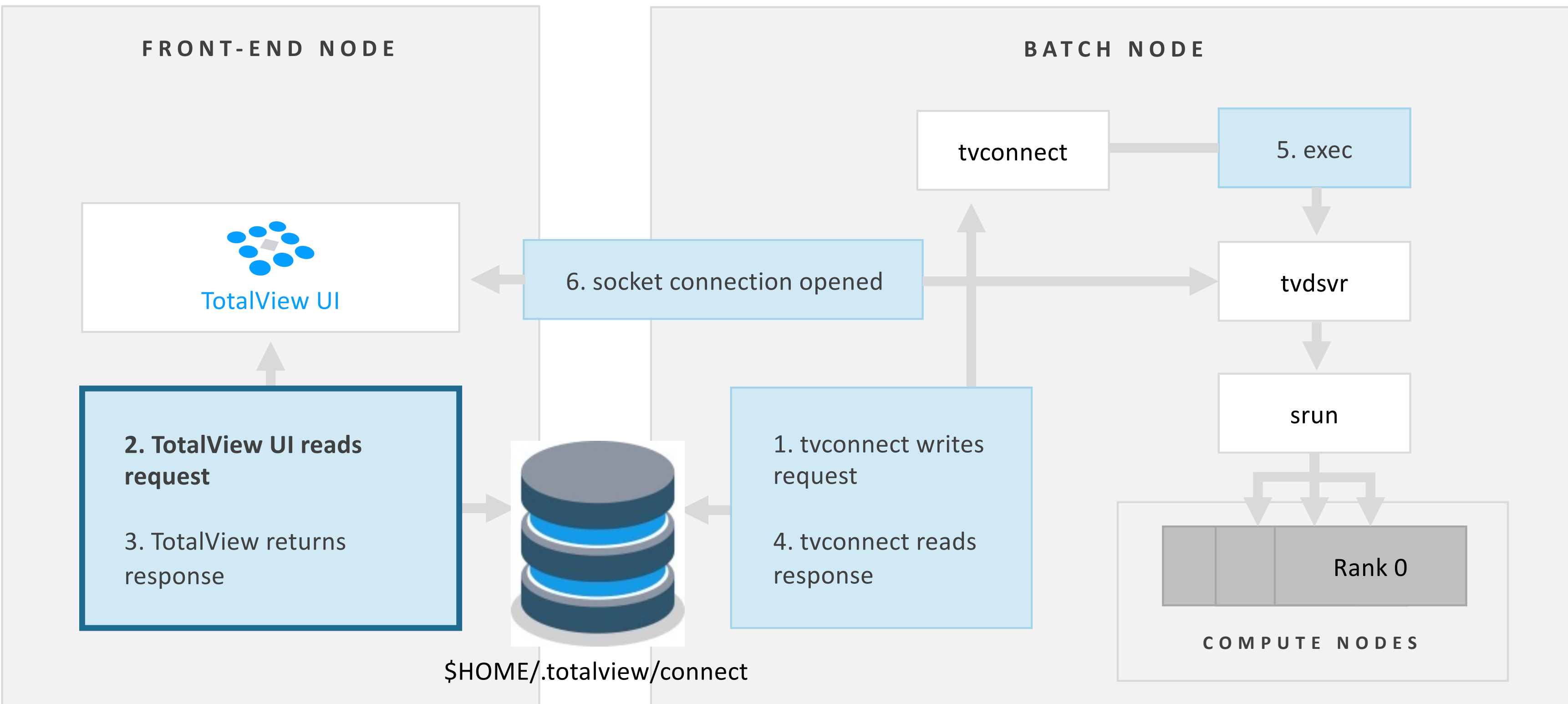
The Solution:

- 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

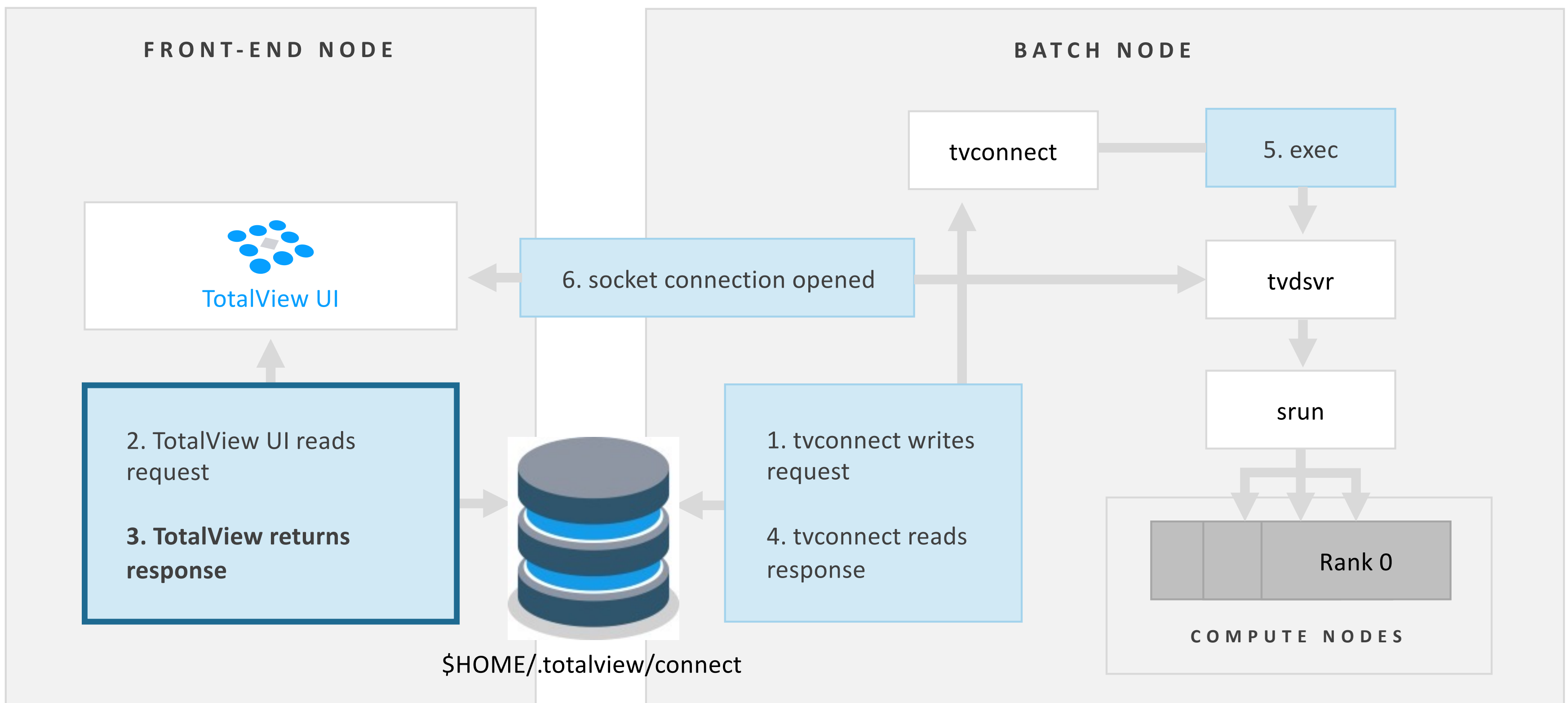
Reverse Connection Flow



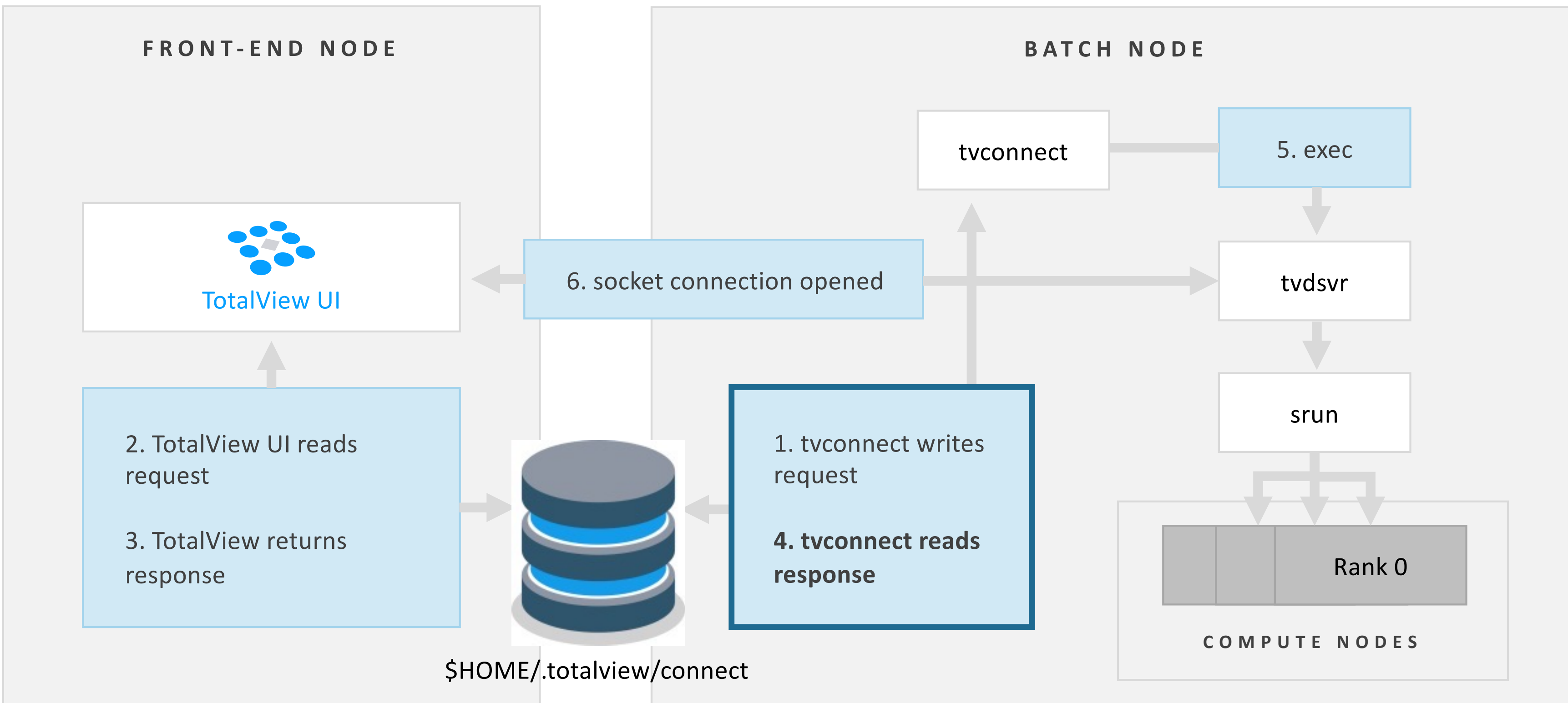
Reverse Connection Flow



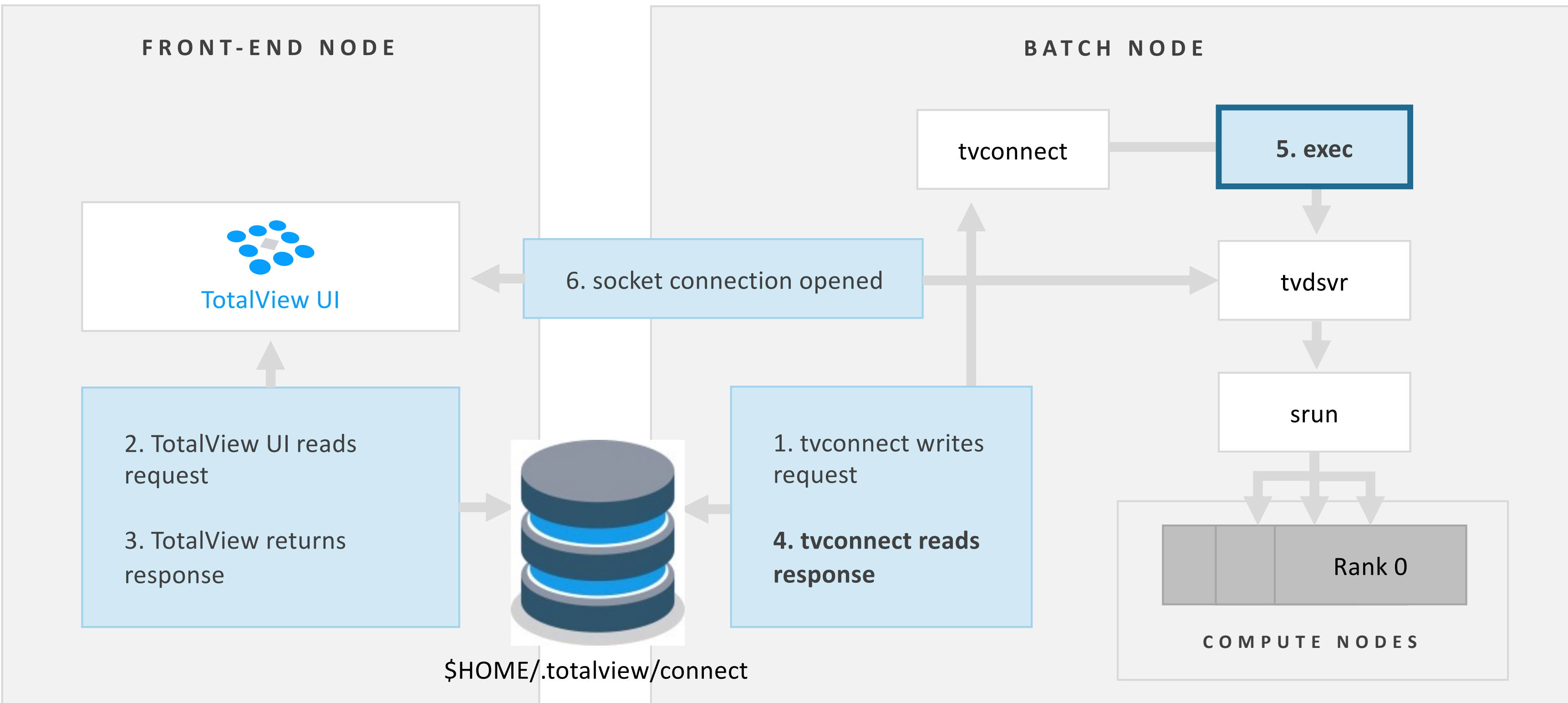
Reverse Connection Flow



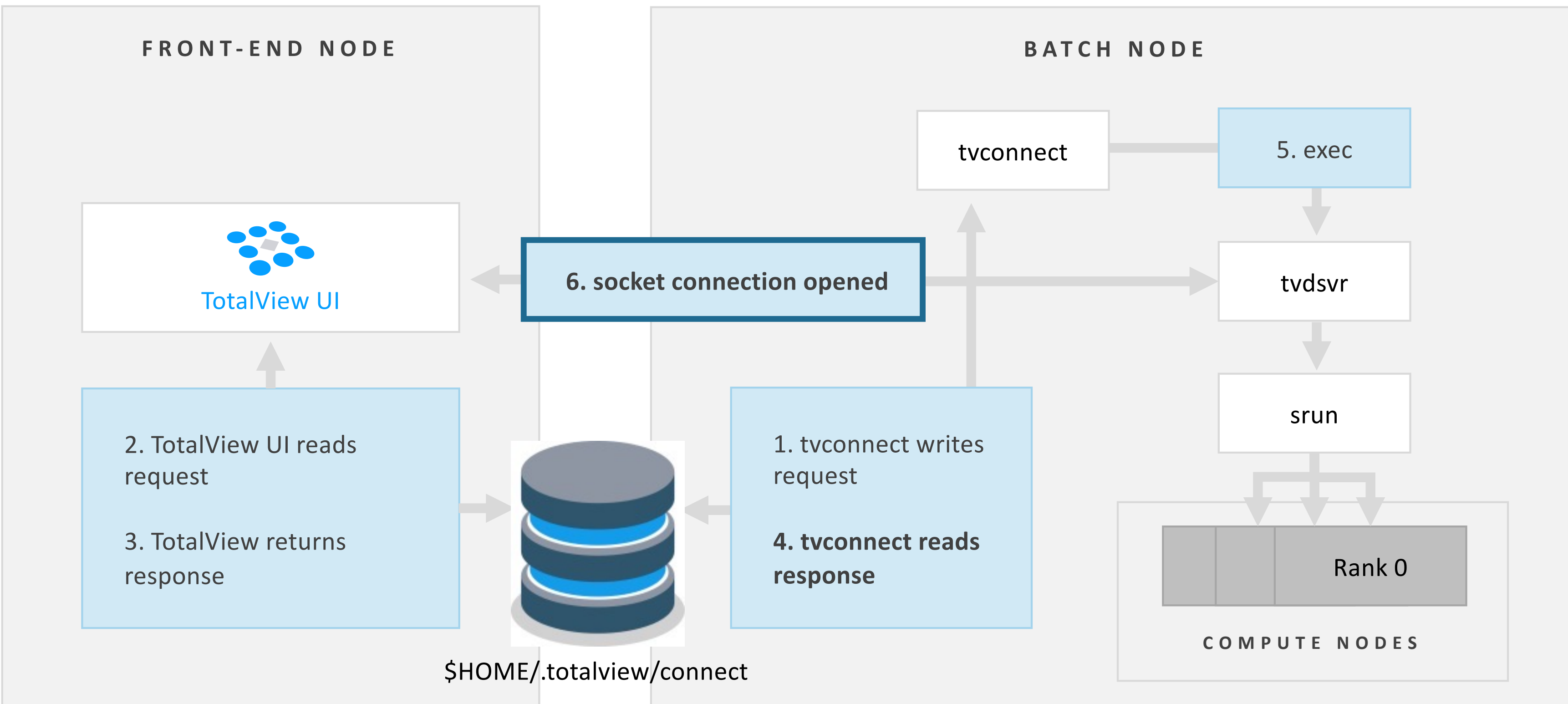
Reverse Connection Flow



Reverse Connection Flow



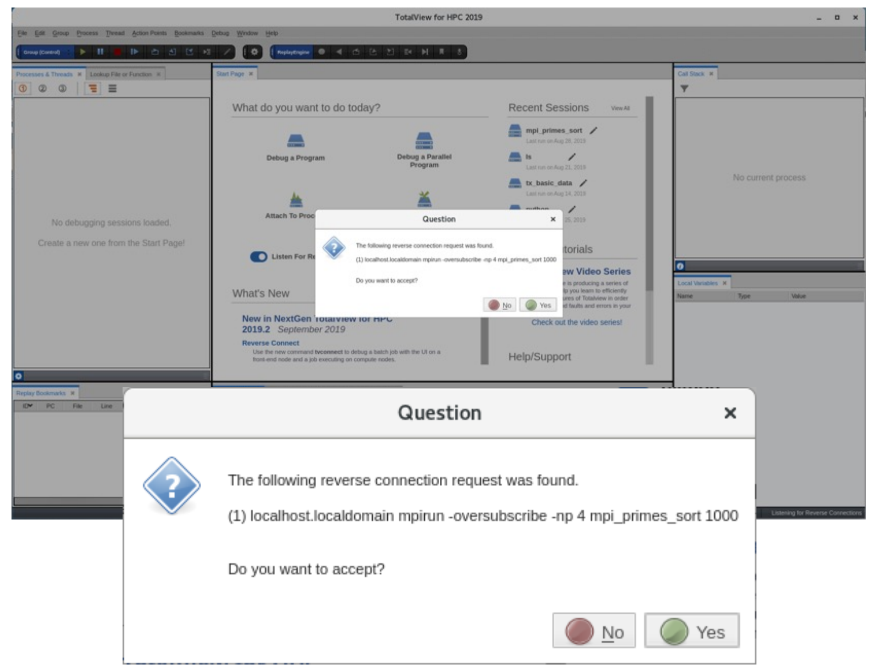
Reverse Connection Flow



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

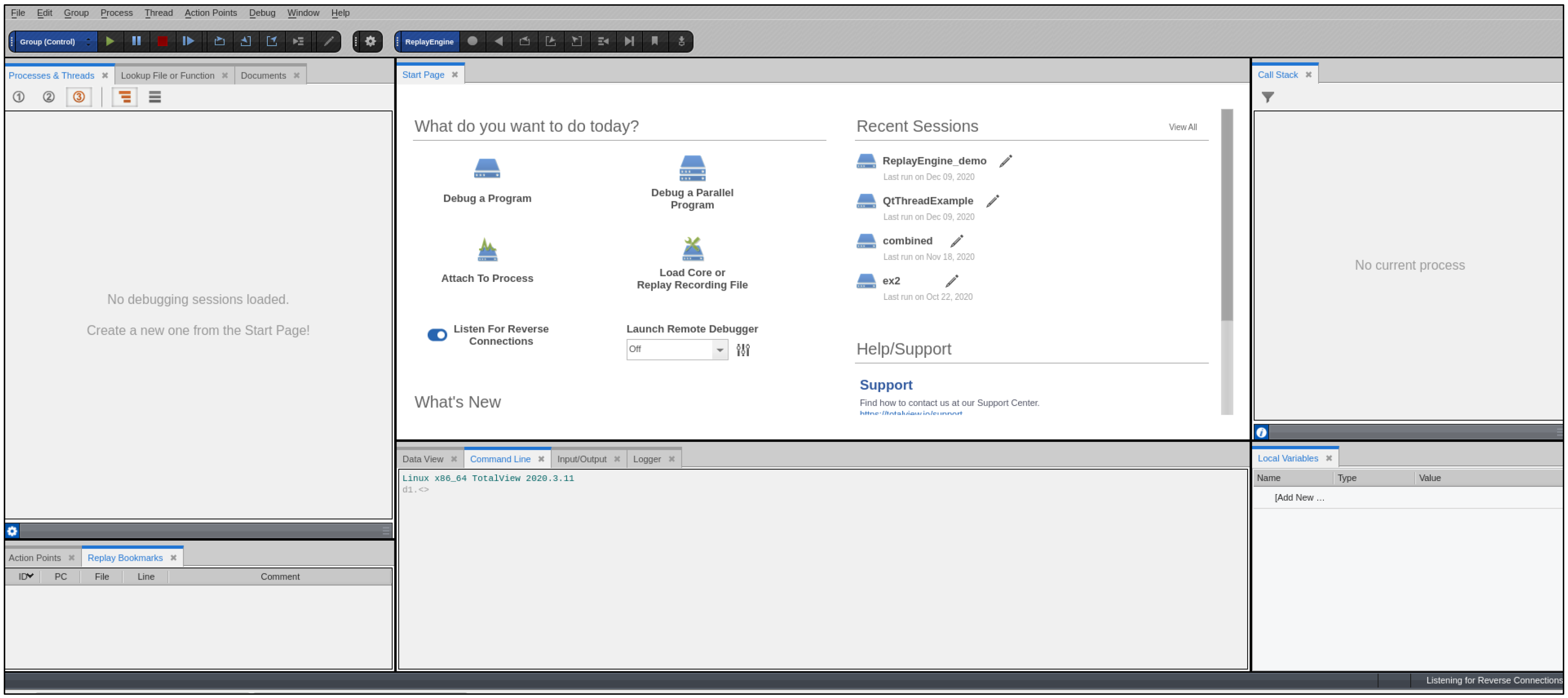
Debugging on Cori

- 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>

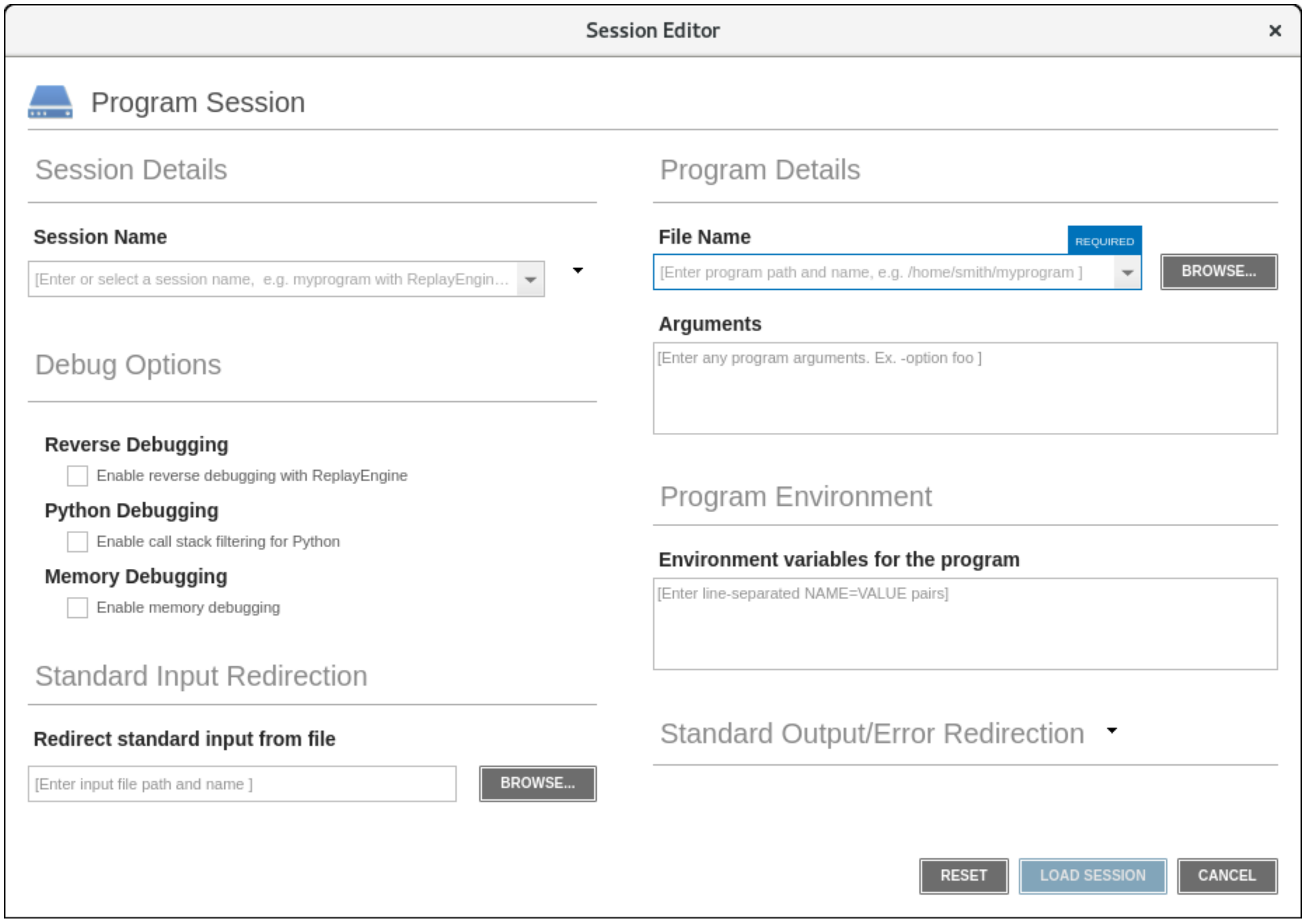
Startup

Start Page

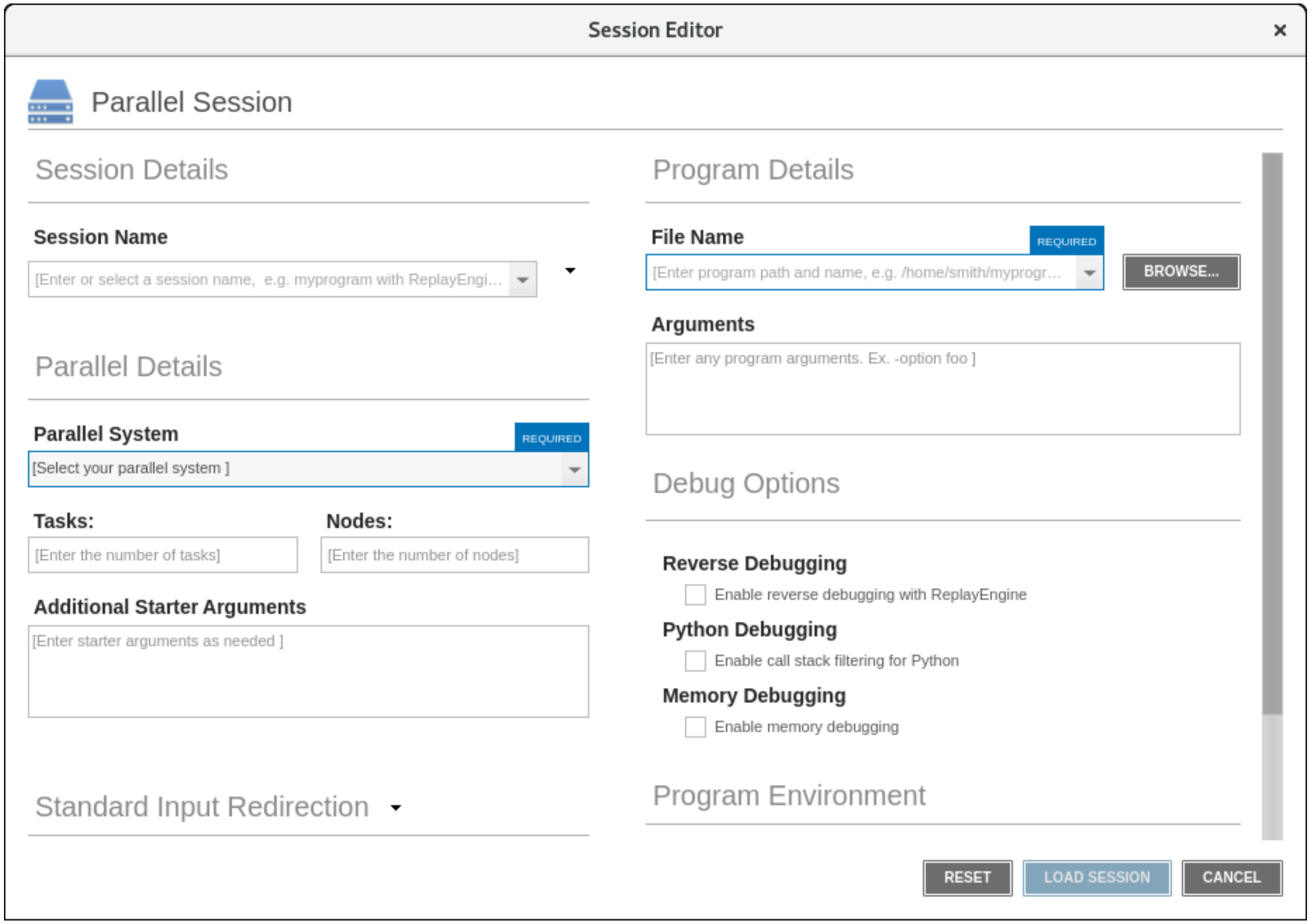
The Start Page is the place to start new debugging sessions, restore recent sessions and learn about the latest new TotalView features.



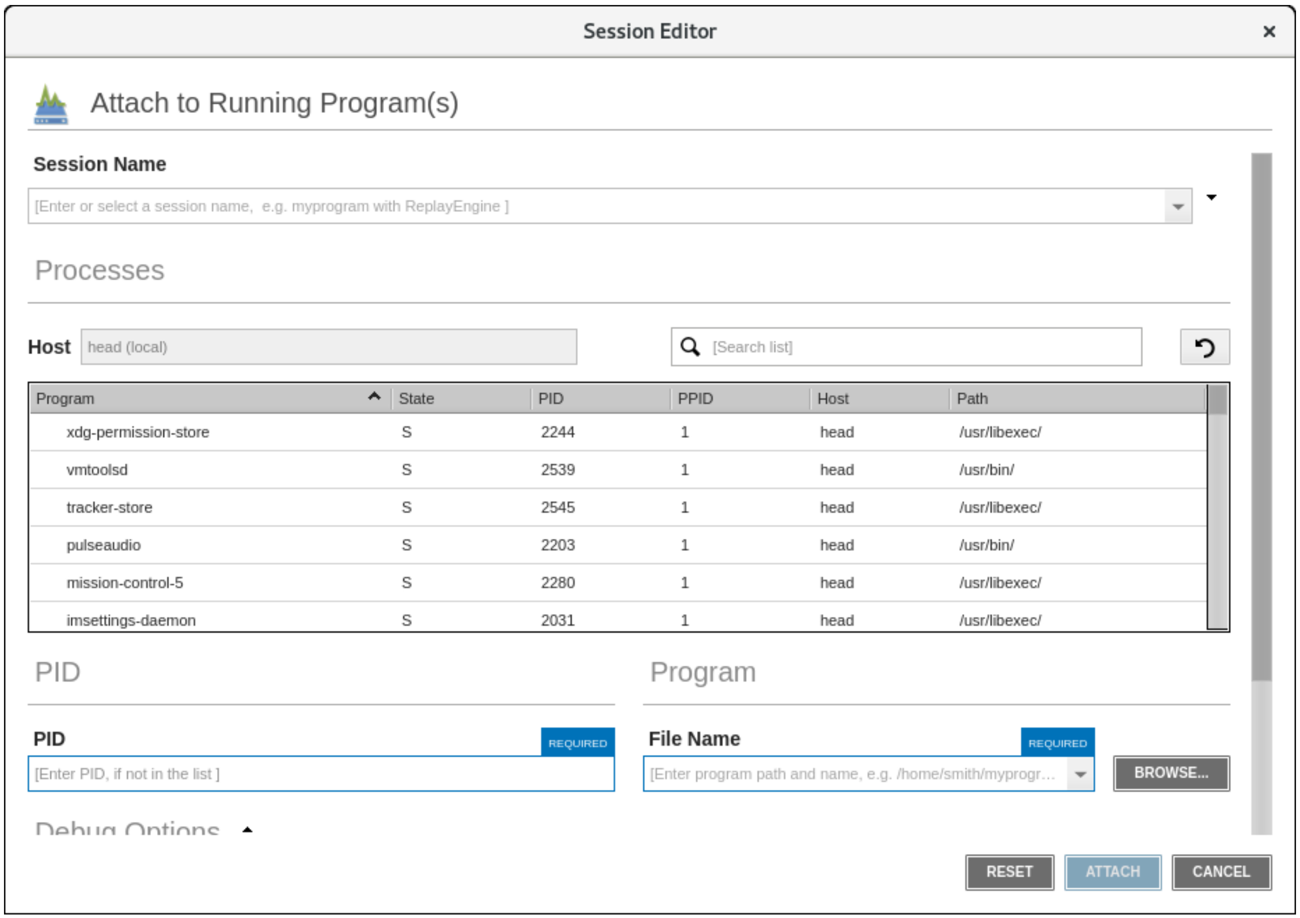
Debugging a New Program



Debugging a Parallel Program



Attach to a Running Program



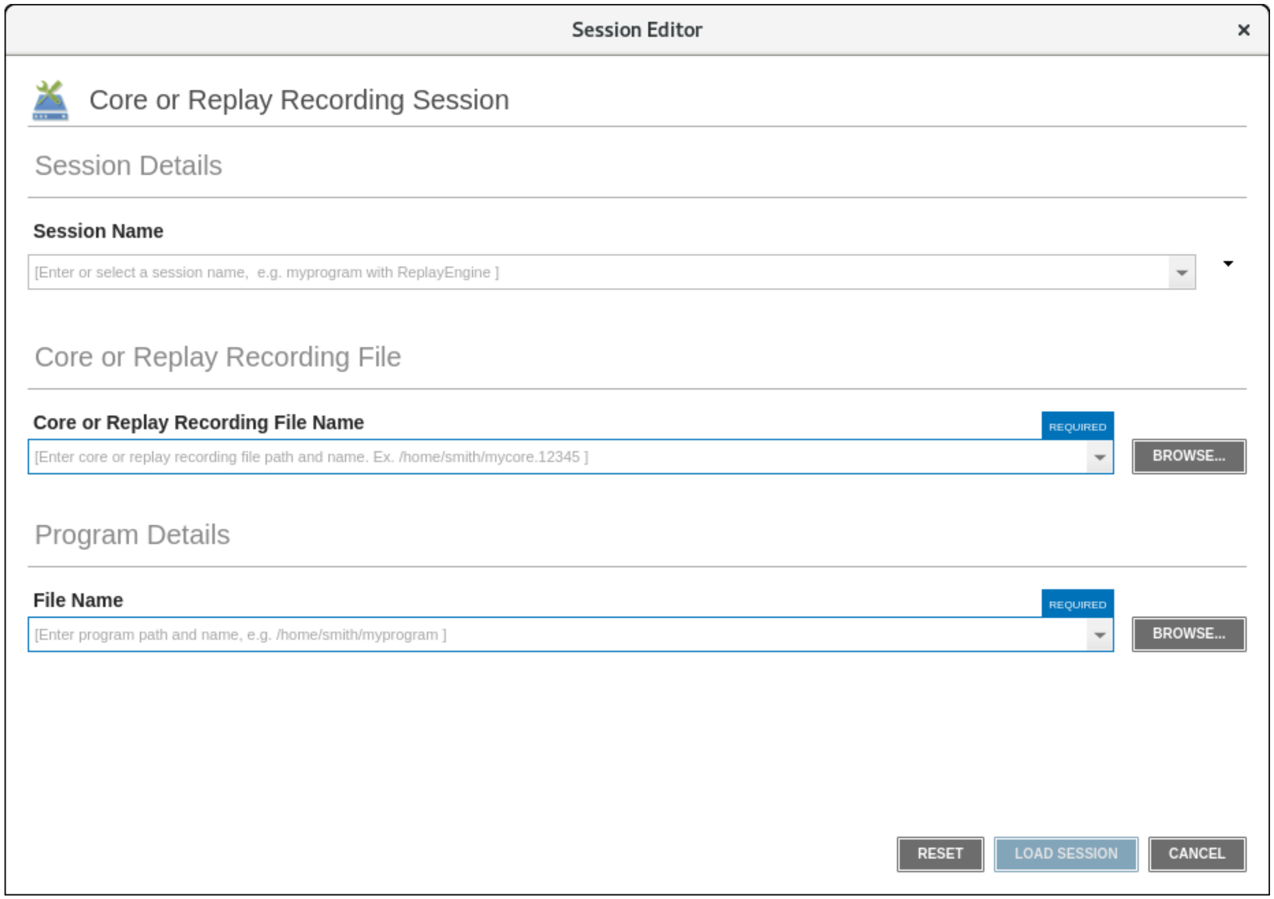
TotalView Power Tip

- Launch a remote TotalView debug server to attach to programs on a remote system:

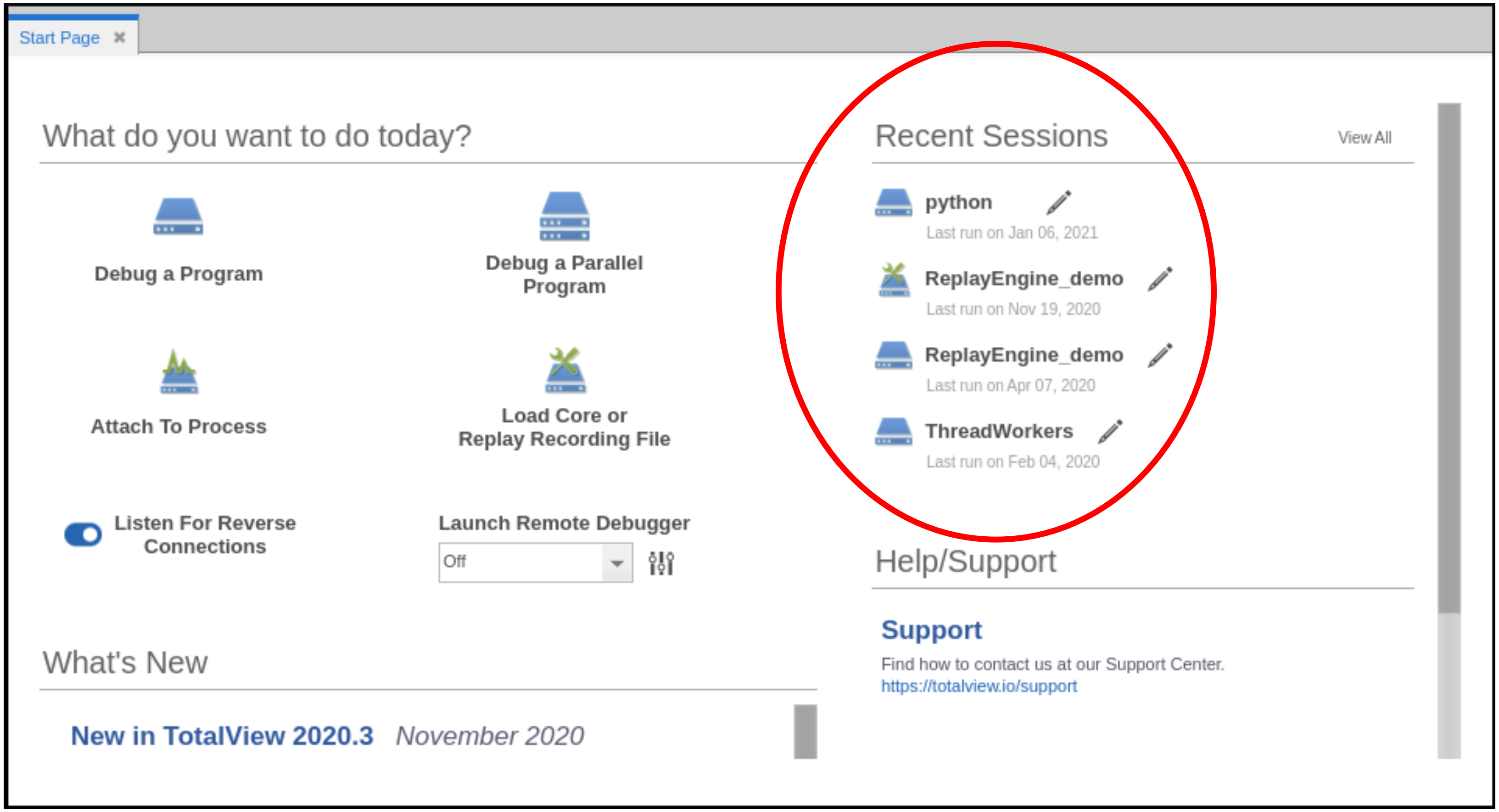
  `totalview executable –r hostname[:port]`

- This will be available in the UI in a coming release.

Open a Core File or Replay Recording Session



Starting a Previous Debugging Session



UI Navigation and Process Control

TotalView's Default Views

Processes & Threads View

Lookup File or Function

Documents

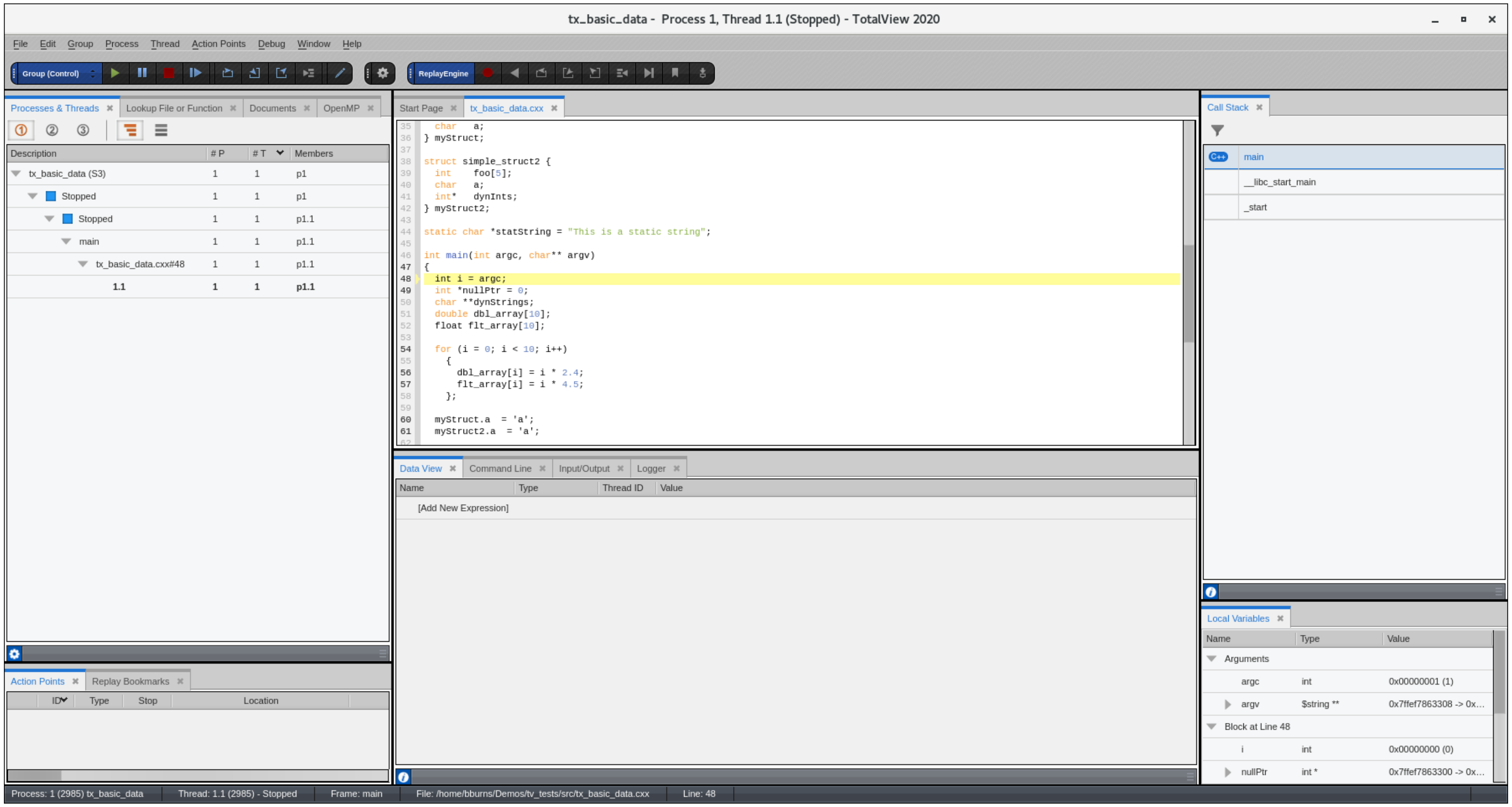
Source View

Call Stack View

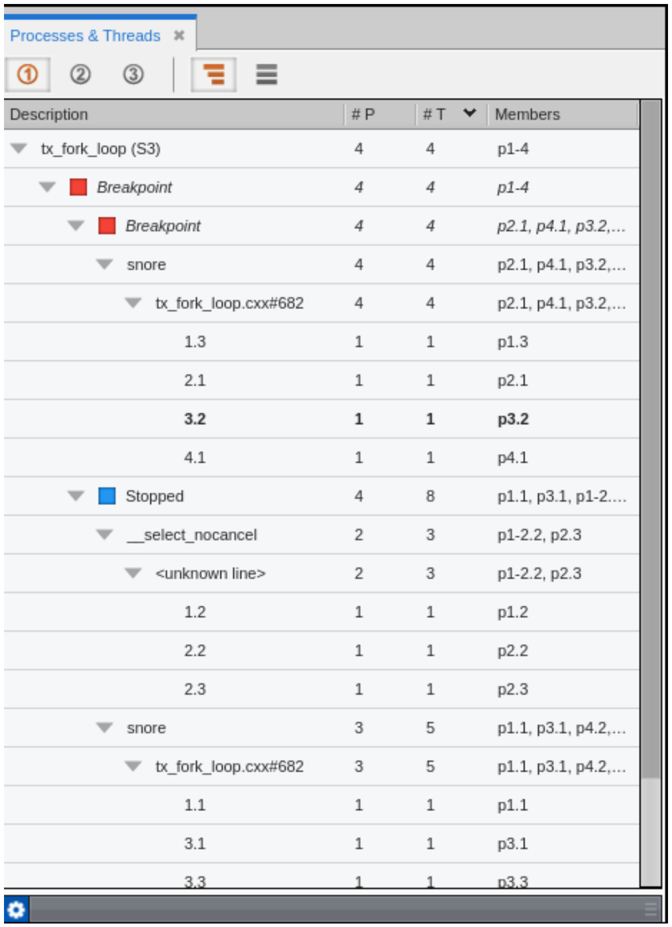
Local Variables View

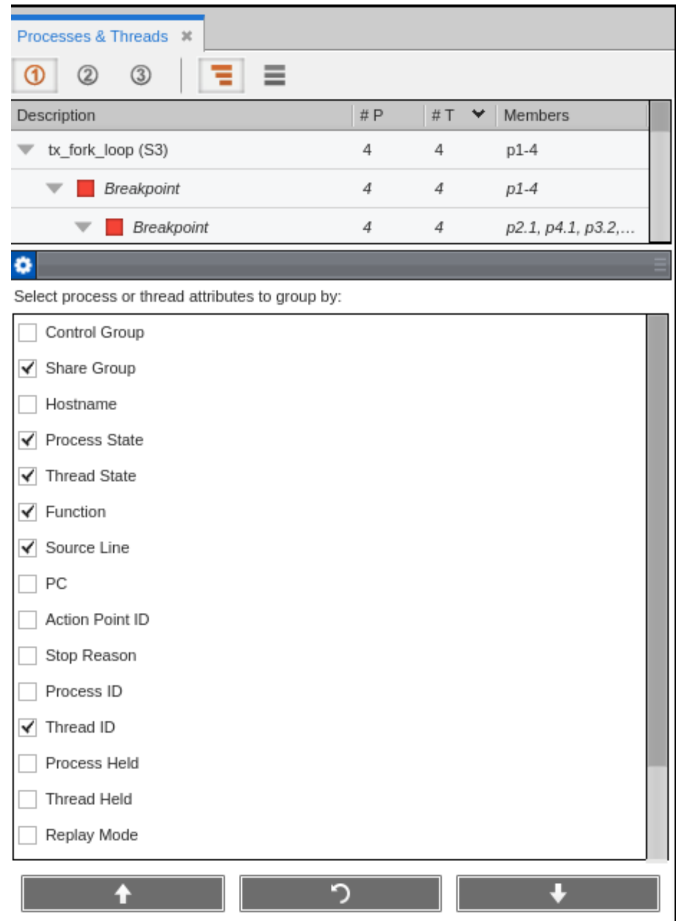
Data View, Command Line, Input/Output

Action Points, Replay Bookmarks

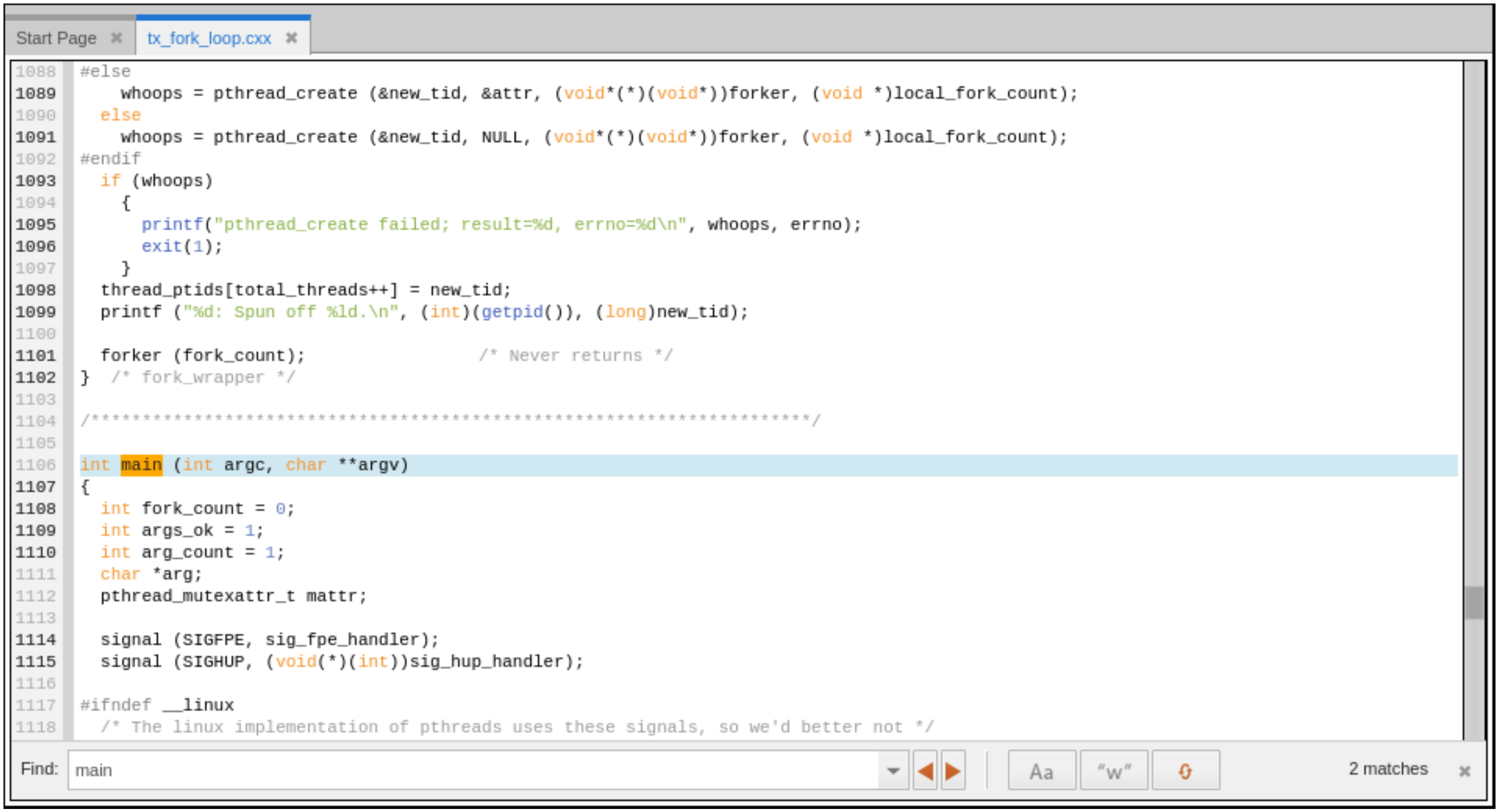


Process and Threads View

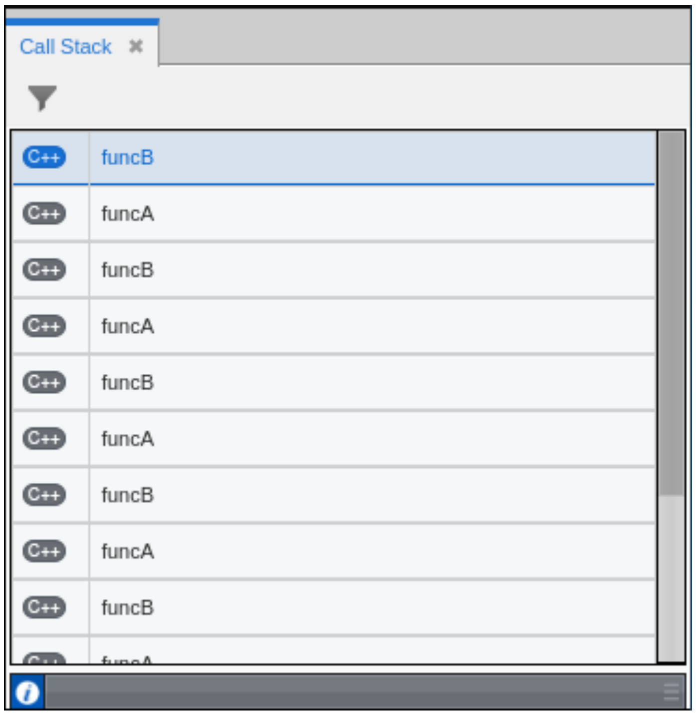


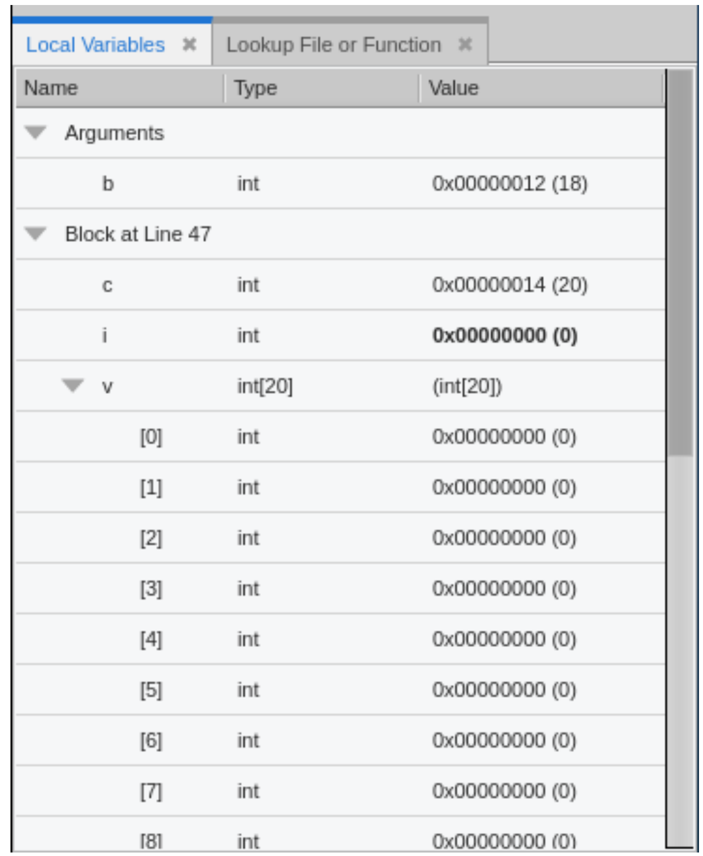


Source View

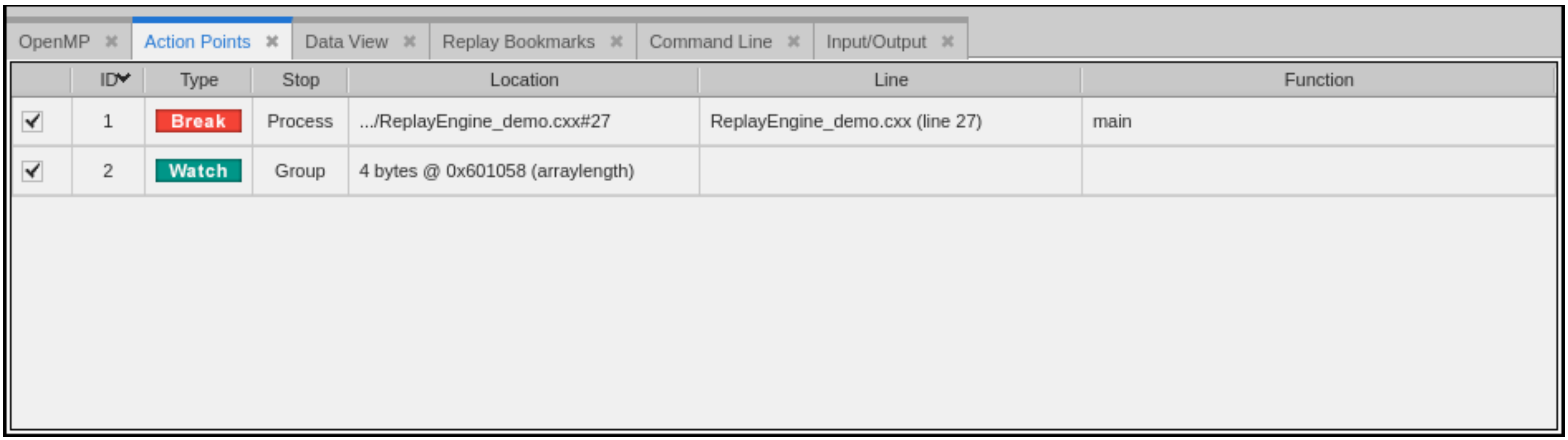

Start Page
tx_fork_loop.cxx
#else
whoops = pthread_create (&new_tid, &attr, (void*(*)(void*))forker, (void *)local_fork_count);
else
whoops = pthread_create (&new_tid, NULL, (void*(*)(void*))forker, (void *)local_fork_count);
#endif
if (whoops)
{
printf("pthread_create failed; result=%d, errno=%d\n", whoops, errno);
exit(1);
}
thread_ptids[total_threads++] = new_tid;
printf ("%d: Spun off %ld.\n", (int)(getpid()), (long)new_tid);
forker (fork_count); /* Never returns */
} /* fork_wrapper */
/**/
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);
#ifndef __linux
/* The linux implementation of pthreads uses these signals, so we'd better not */
Find:
main
Aa
"w"
2 matches

Call Stack View and Local Variables View

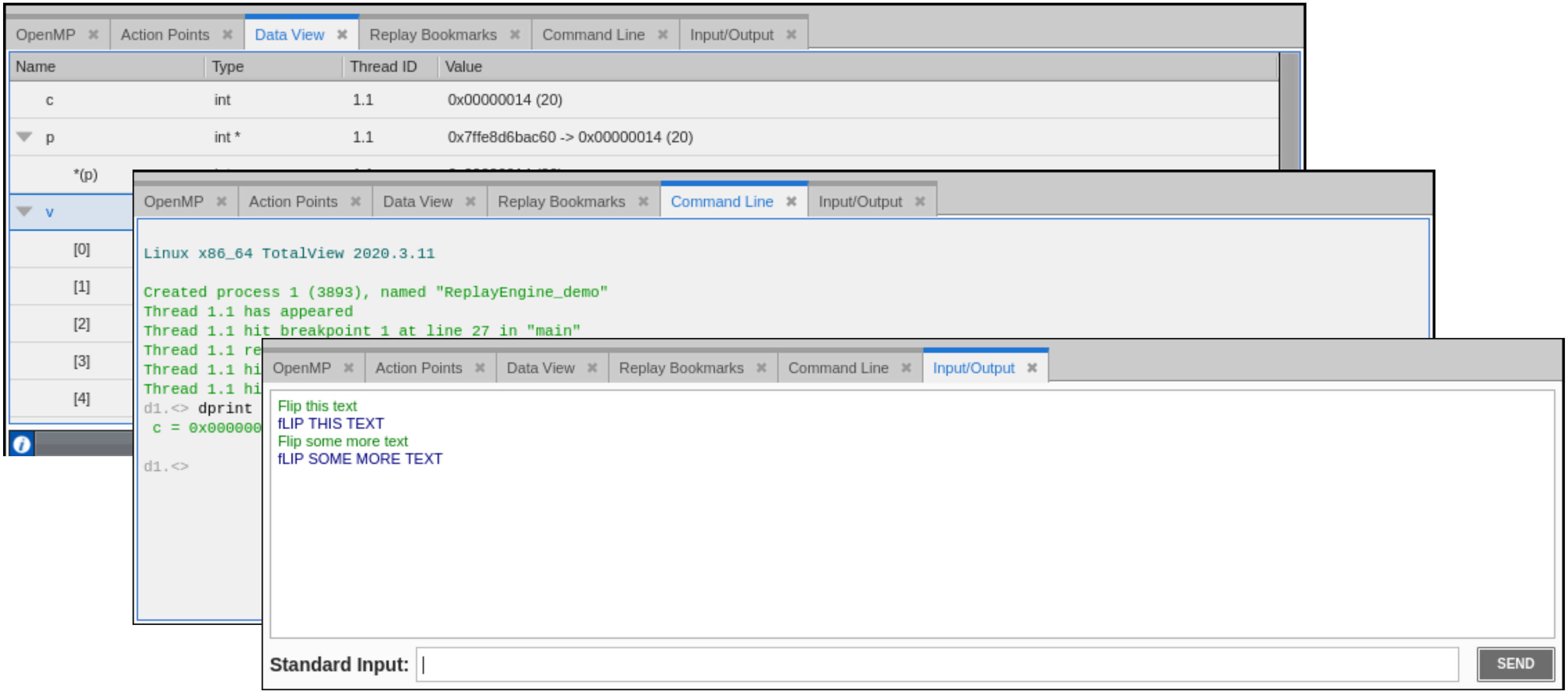




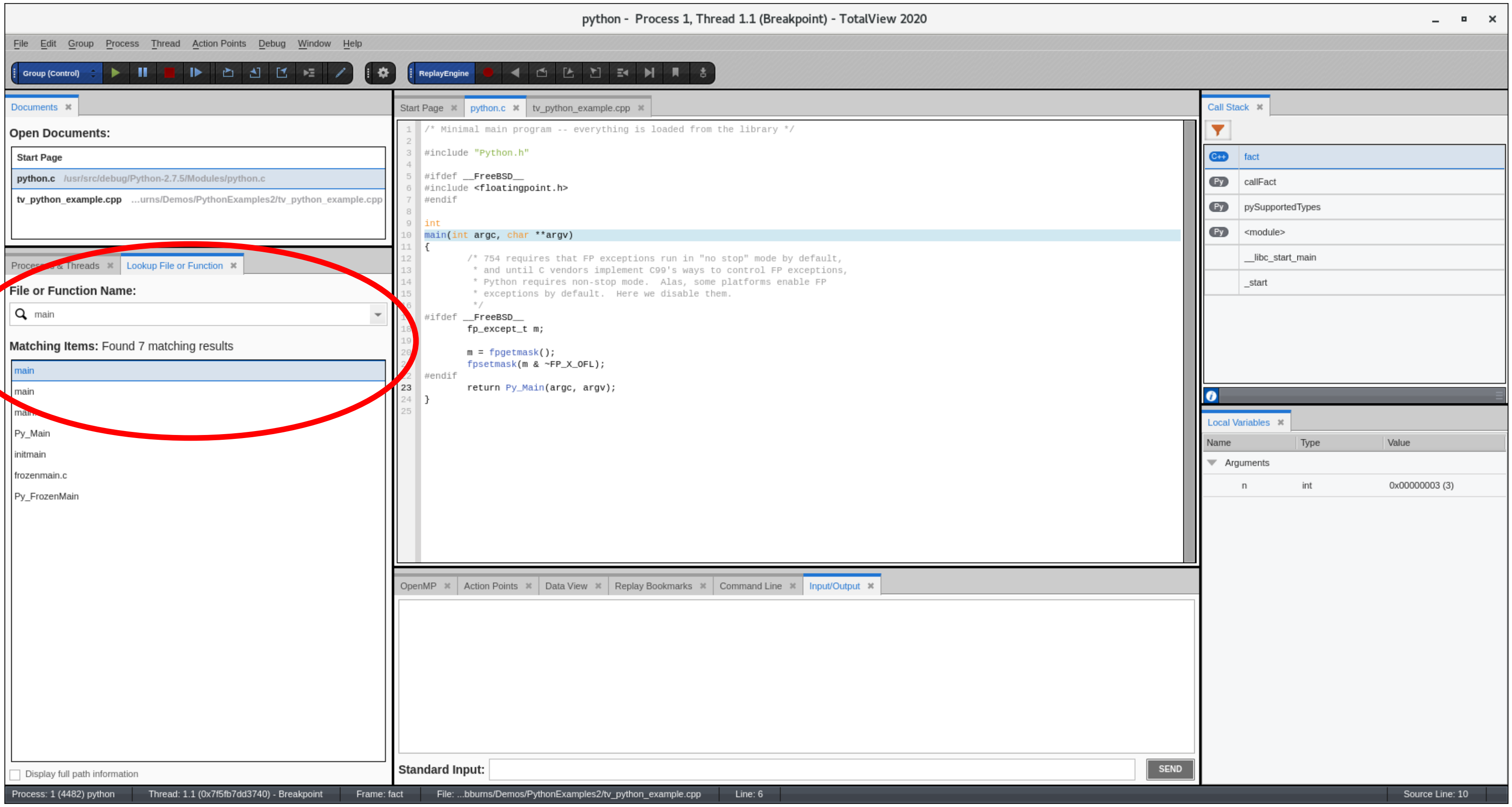
Action Points View



Data View, Command Line View and Input/Output View



Lookup File or Function

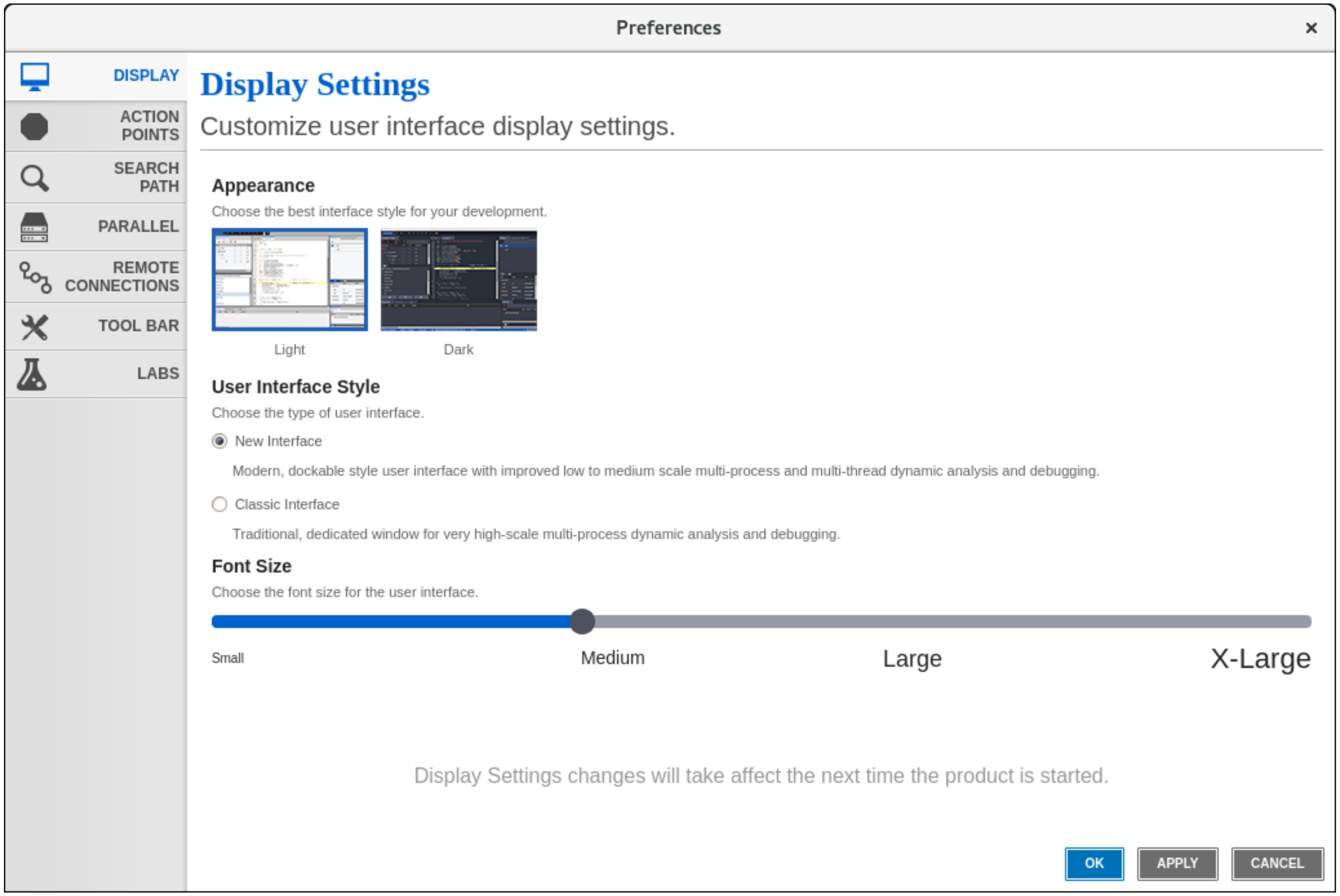


Preferences

File > Preferences Menu

Or

“Gear” Toolbar Item



TotalView Toolbar

Command	Description
Go	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.
Halt	Stops the thread at its current execution point.
Kill	Stops program execution. Existing breakpoints and other settings remain in effect.
Restart	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.
Next	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.
Step	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.
Out	If the thread is in a block of execution, runs the thread to the first line of execution beyond that block.
Run To	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.

Stepping Commands

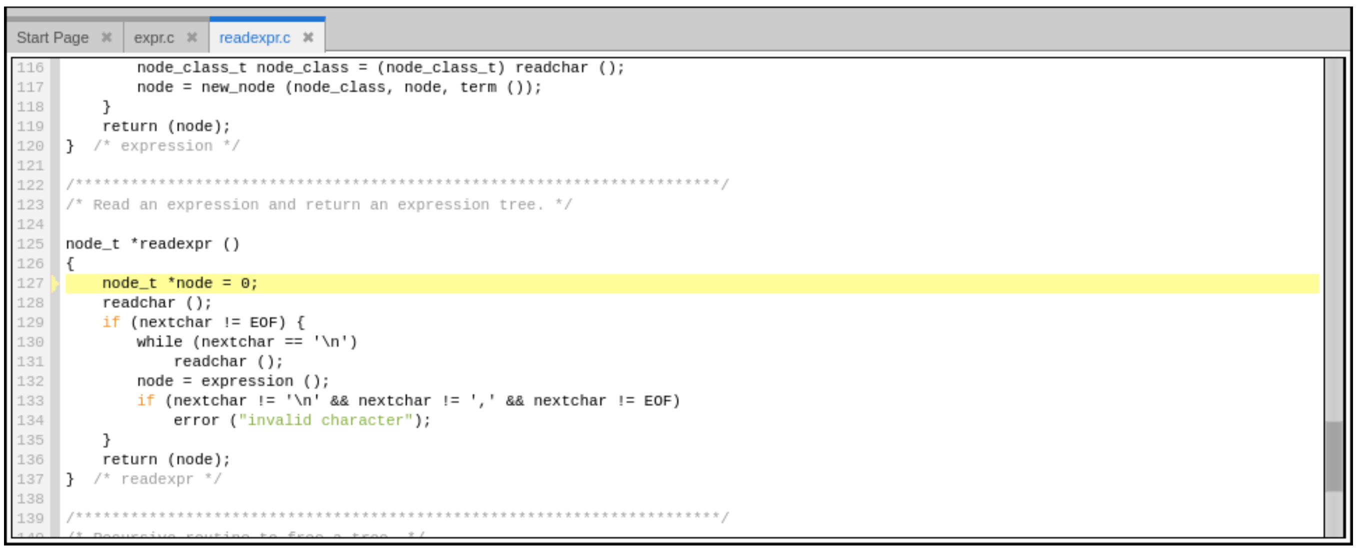
Select Step in the toolbar. TotalView stops the program just before the first executable statement, the call to setjmp (context);

Start Page expr.c

```c
    longjmp (context, 1);
}  /* error */

/***********************************************************************/
/* Read an expression, build a tree, evaluate the tree and print it. */

node_t *readexpr ();
double evaluate ();
void freetree (node_t *);

int main (int argc, char **argv)
{
    node_t *node;
    setjmp (context);
    while (node = readexpr ()) {
        previous = evaluate (node);
        printf ("%g %ld (0x%lx)\n", previous, (long) previous, (long) previous);
        fflush (stdout);
        freetree (node);
    }
    return (0);
}  /* main */
```

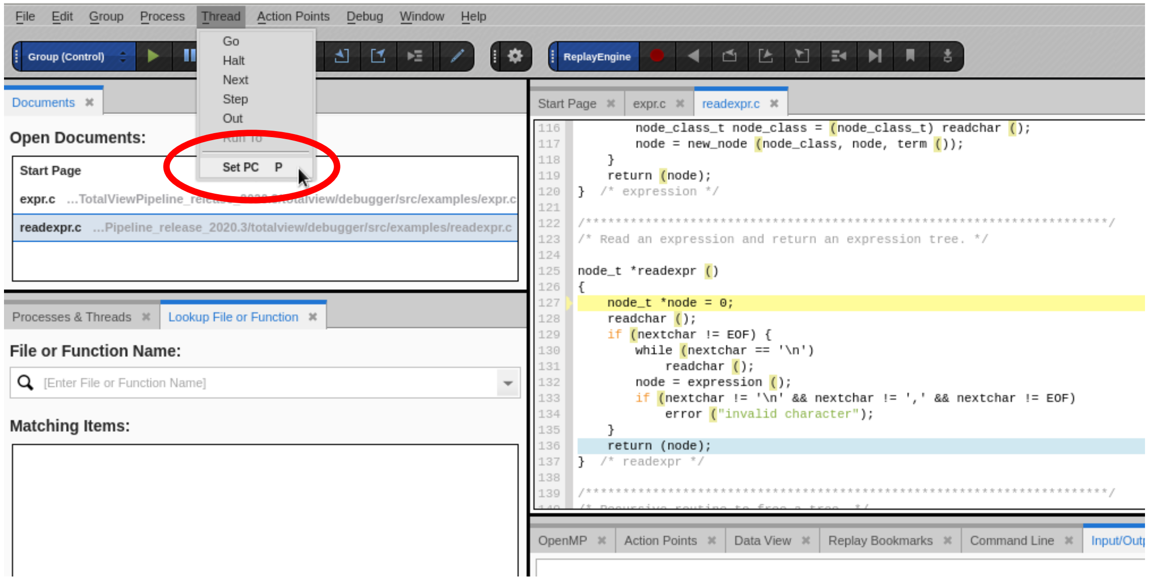
Stepping Commands

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

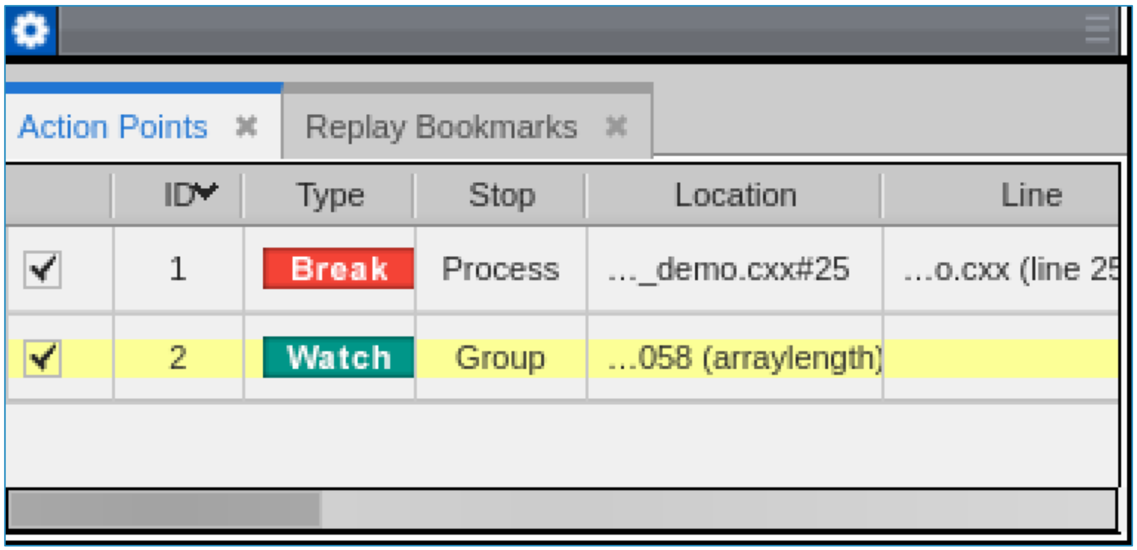


Demo

- TotalView threads demo (txdining)

Action Points

Action Points



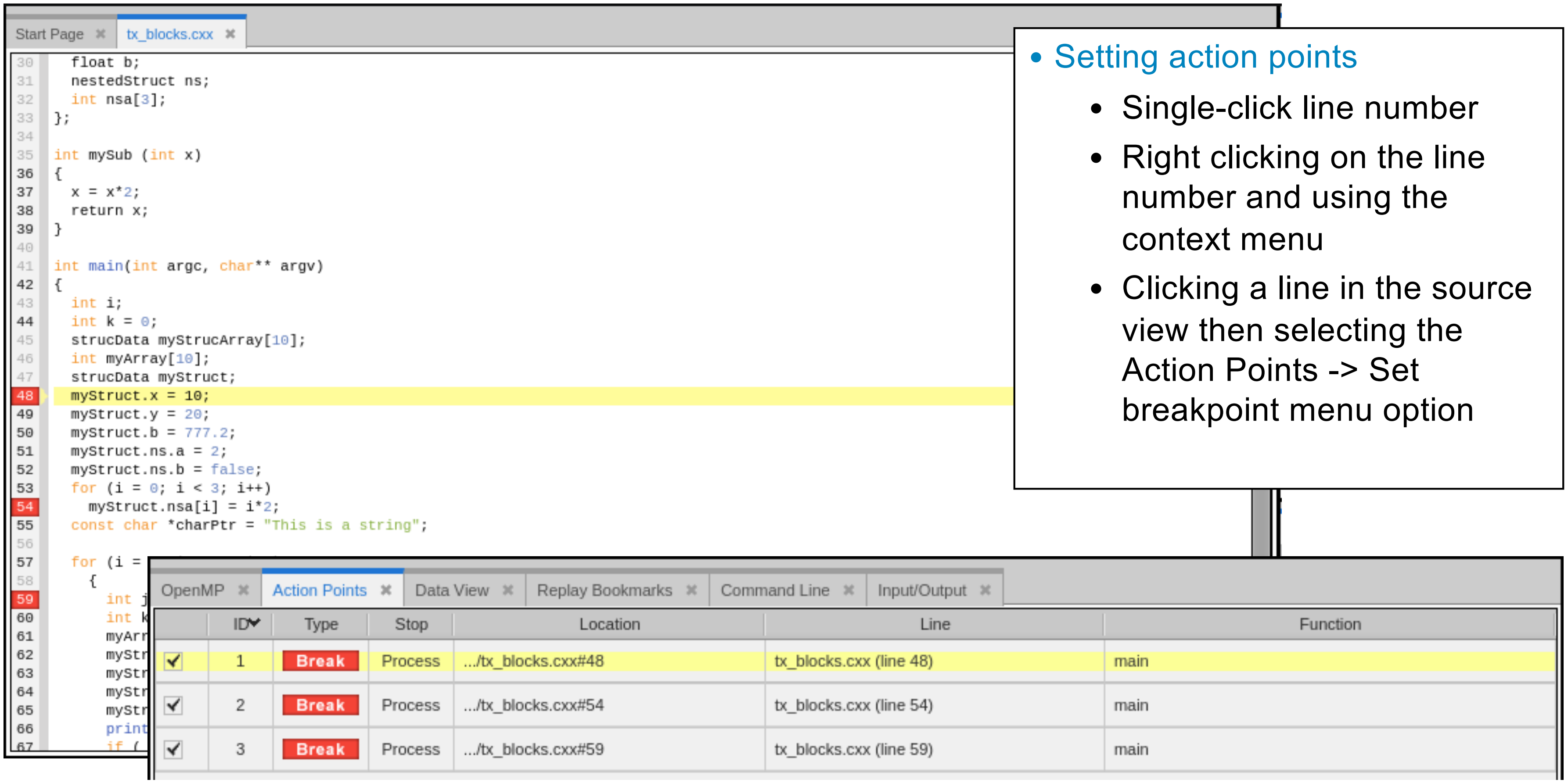
Breakpoint

Evaluation Point (Evalpoint)

Watchpoint

Barrierpoint

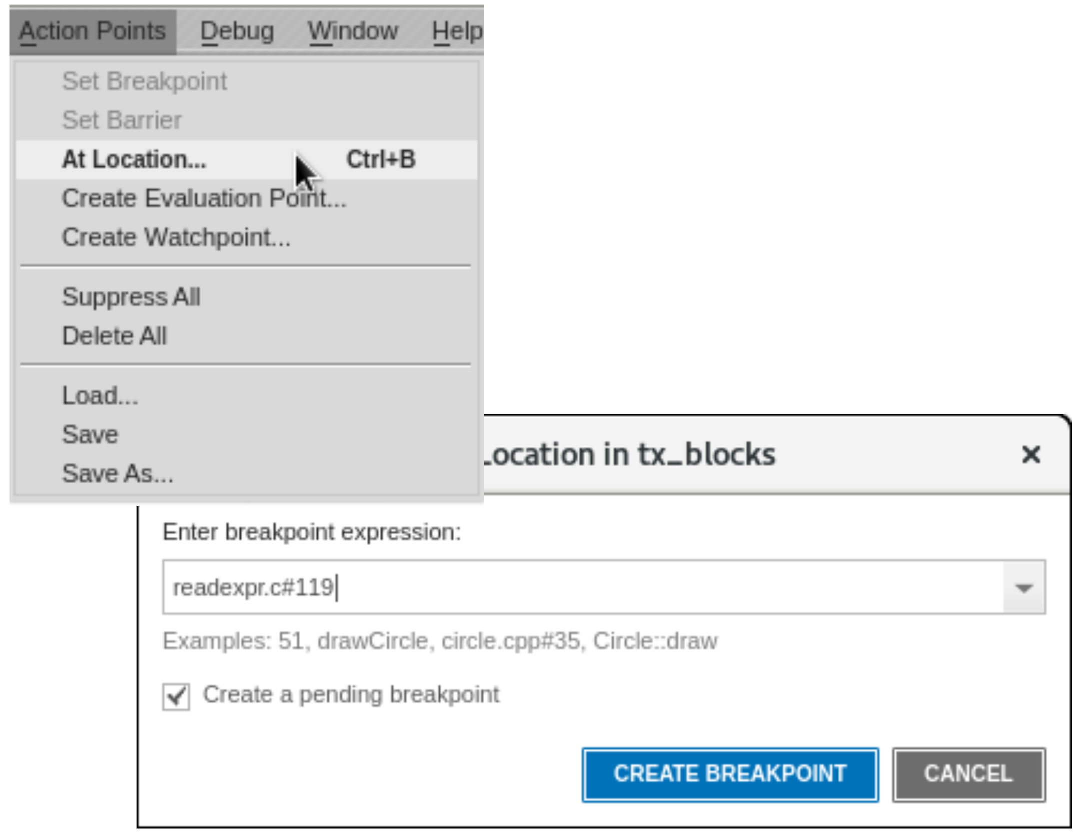
Setting Breakpoints



- 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

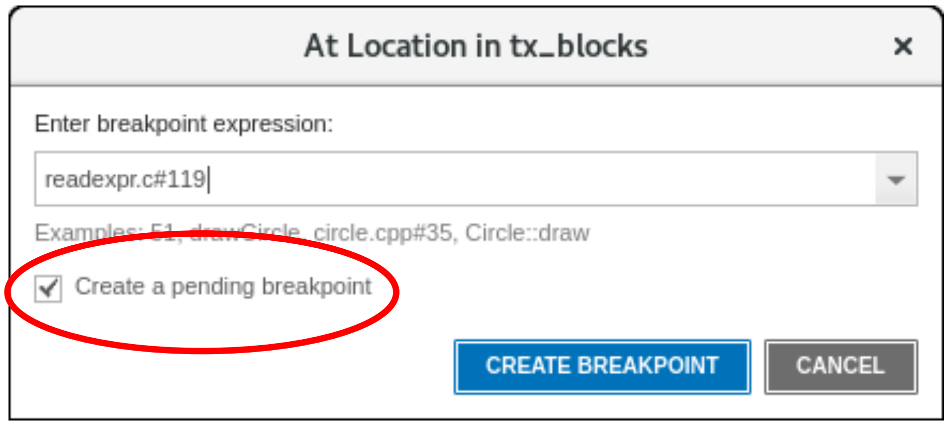
Setting Breakpoints

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

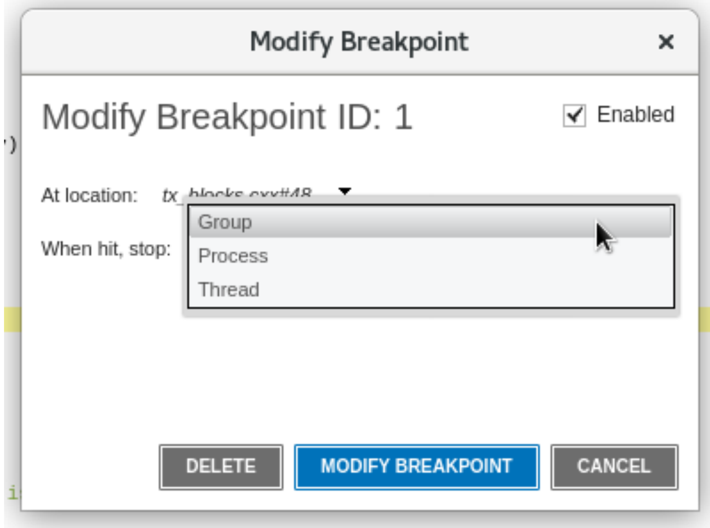


Pending Breakpoints

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



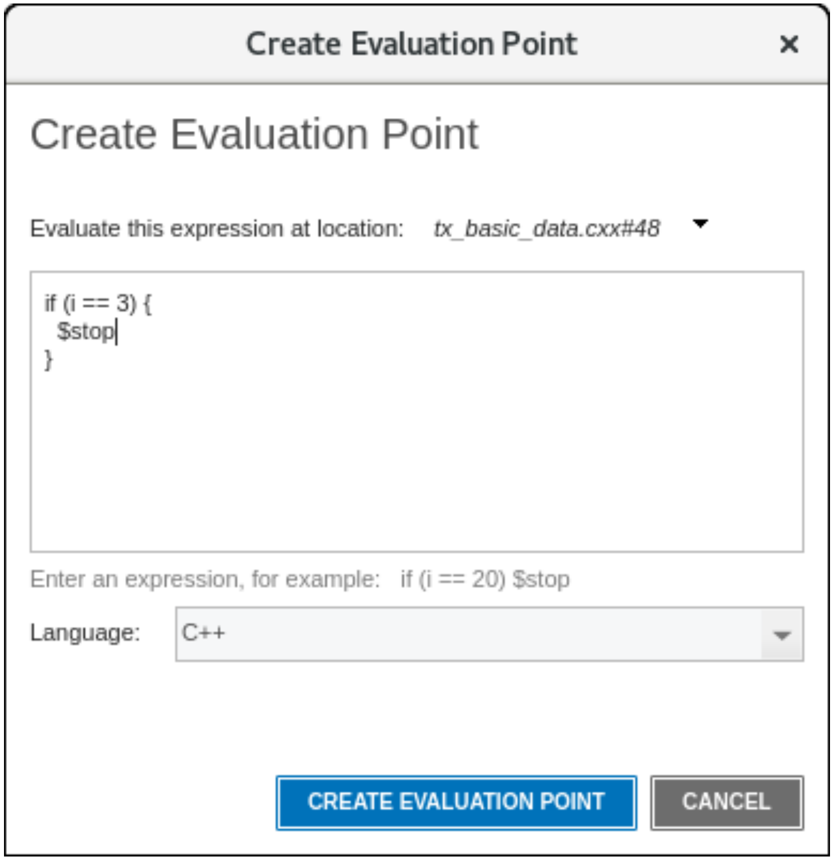
Modifying Breakpoints



- 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
Create Evaluation Point
Evaluate this expression at location: tx_basic_data.cxx#48
if (i == 3) {
$stop
}
Enter an expression, for example: if (i == 20) $stop
Language:
C++
CREATE EVALUATION POINT
CANCEL

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

```
printf("The value of result is %d\n", result);
```

- Skip some code

```
goto 63;
```

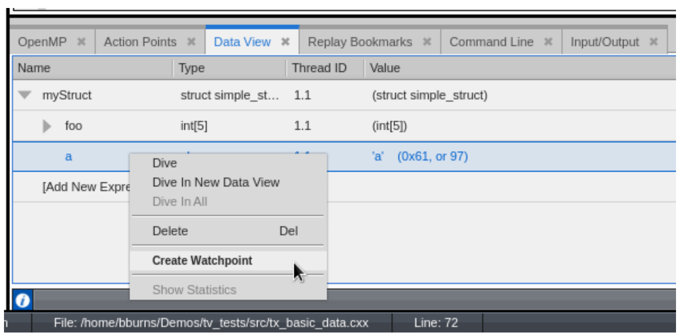
- Stop a loop after a certain number of iterations

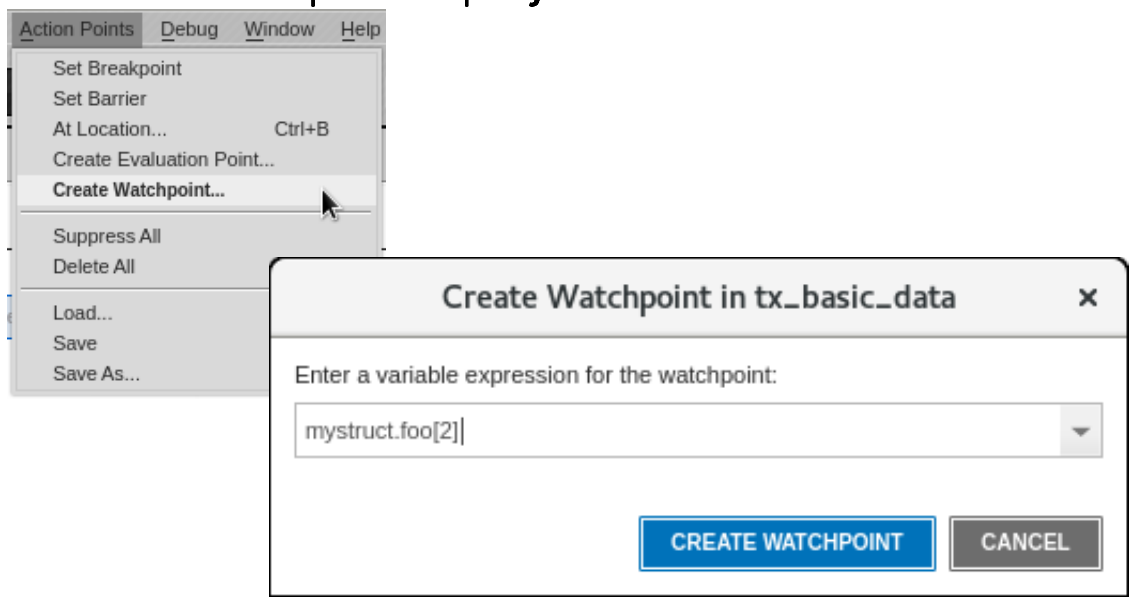
```
if ( (i % 100) == 0) {
   printf("The value of i is %d\n", i);
   $stop;
}
```

See “Using Built-in Statements” in Appendix A of the User Guide for more information on “$” expressions: https://help.totalview.io/current/HTML/index.html#page/TotalView/BuiltInStatments.html#ww1894979

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

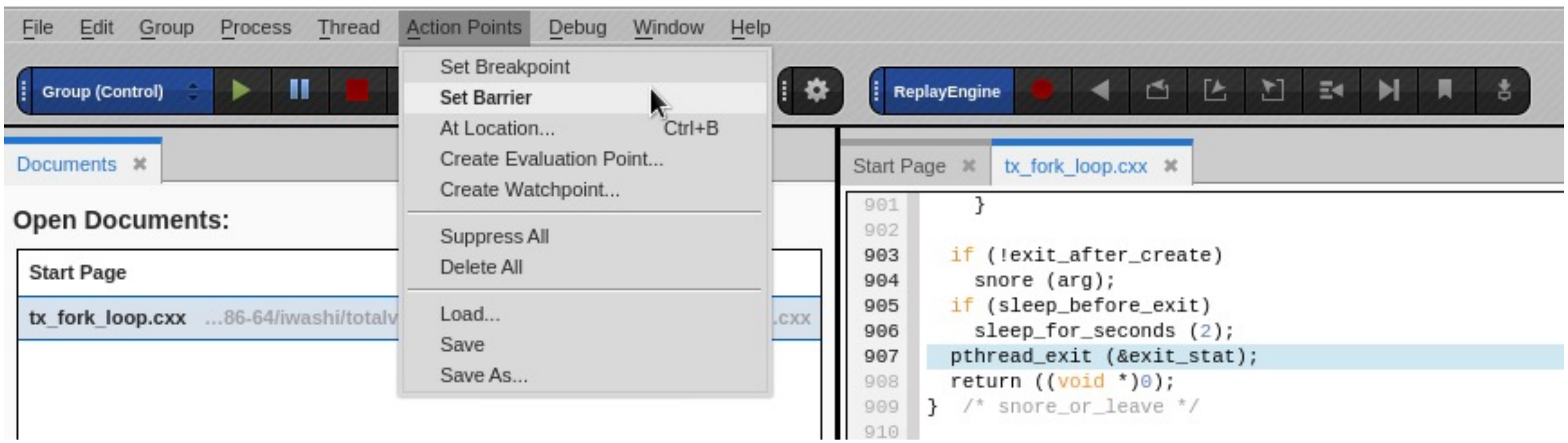
```
if (iValue != 42 && iValue != 44) {
    iNewValue = $newval; iOldValue = $oldval; $stop;
}
```

- Example 2

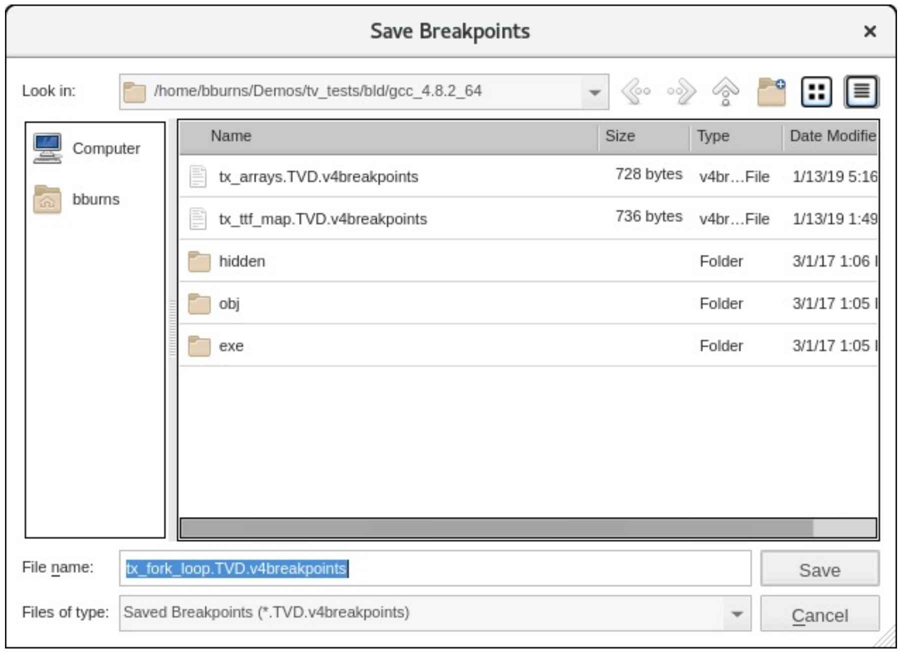
```
if ($oldval >= 0 && $newval < 0) $stop
```

Barrier Breakpoints

- Used to synchronize a group of threads or processes defined in the action point
- Threads or processes are held at barrierpoint 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



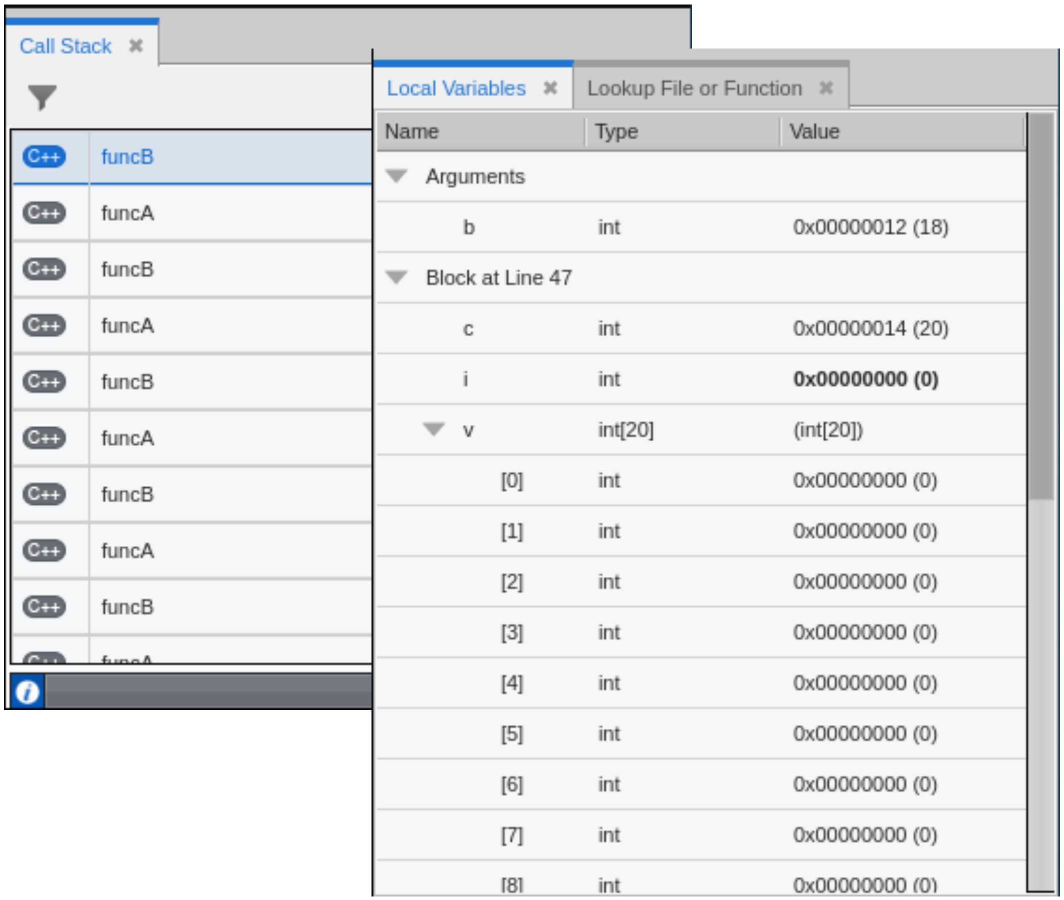
From the Action Points menu select Save or Save As to save breakpoints
Turn on option to save action points on exit

Demo

- TotalView evaluation point demo (Combined)

Examining and Editing Data

Call Stack and Local Variables



Call Stack View

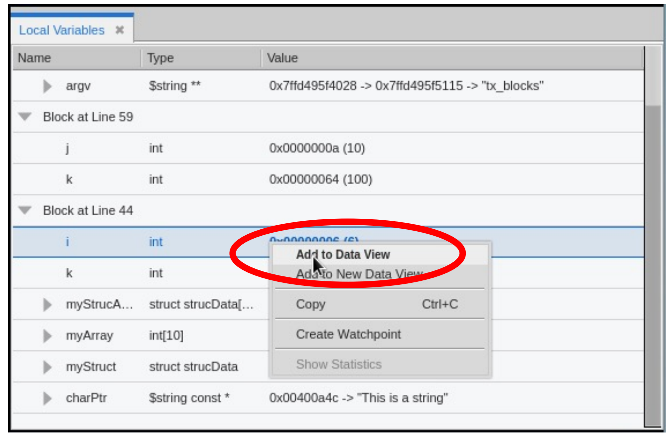
- Lists the set of call frames as the program calls from one function or method to another
- Filter button used to turn on or off filtering of frames.

Local Variables View

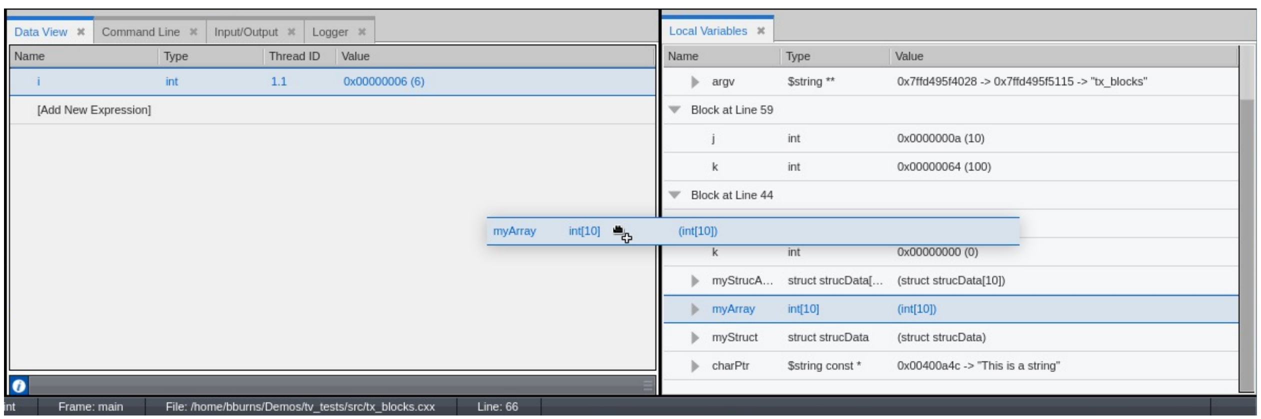
- Displays local variables relative to the current thread of interest and the selected stack frame
- Organized by arguments and blocks
- To edit values, add variable to the Data View

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



Context menu

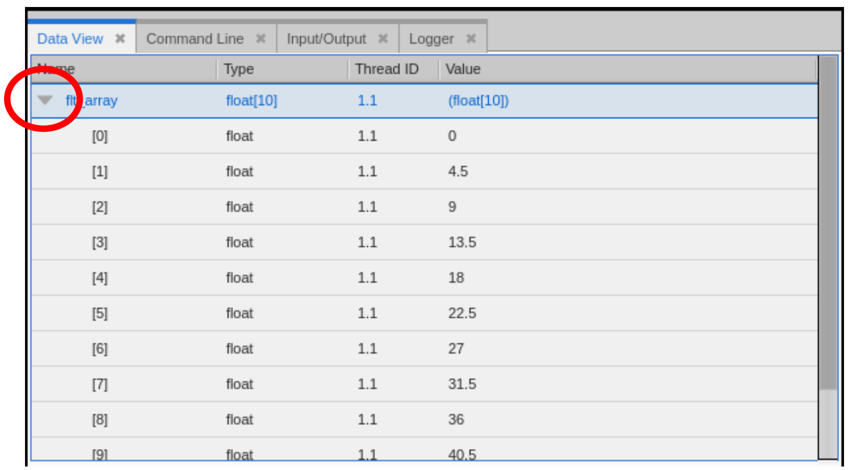


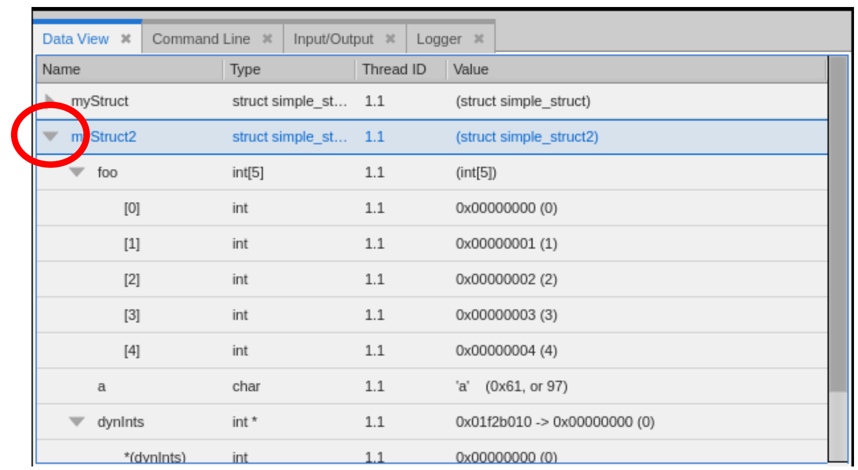
Drag and drop

The Data View Panel – Expanding Arrays and Structures

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

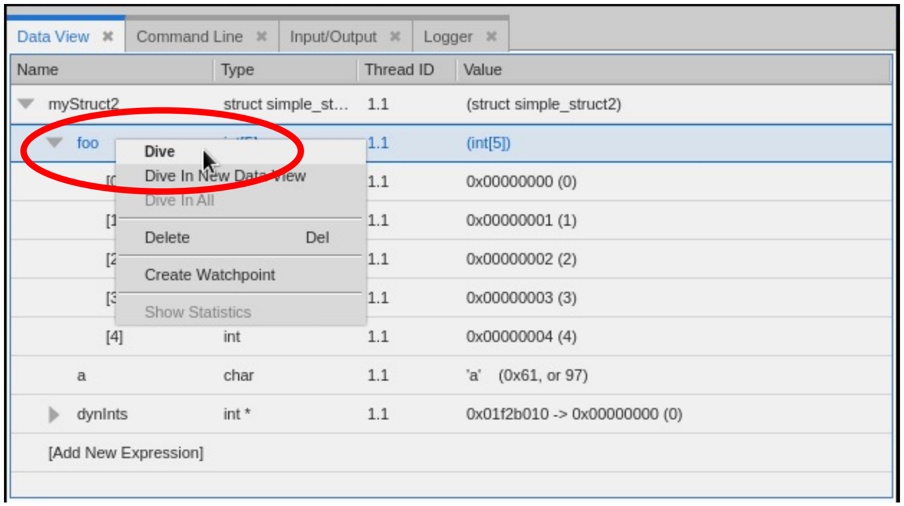
Any nested structures are displayed in the data view

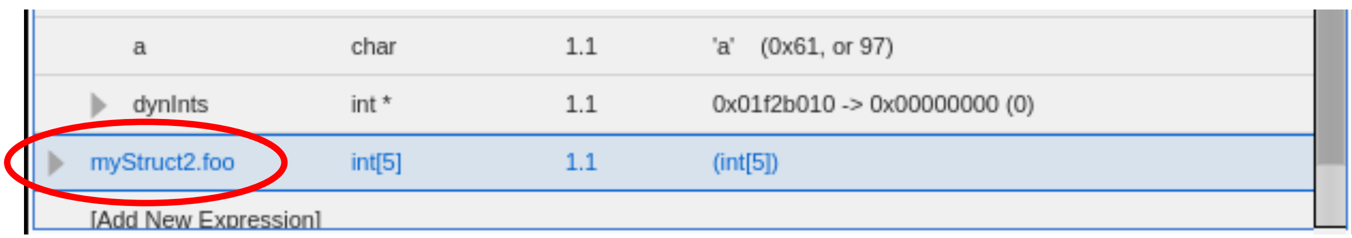




The Data View Panel – Diving on Data

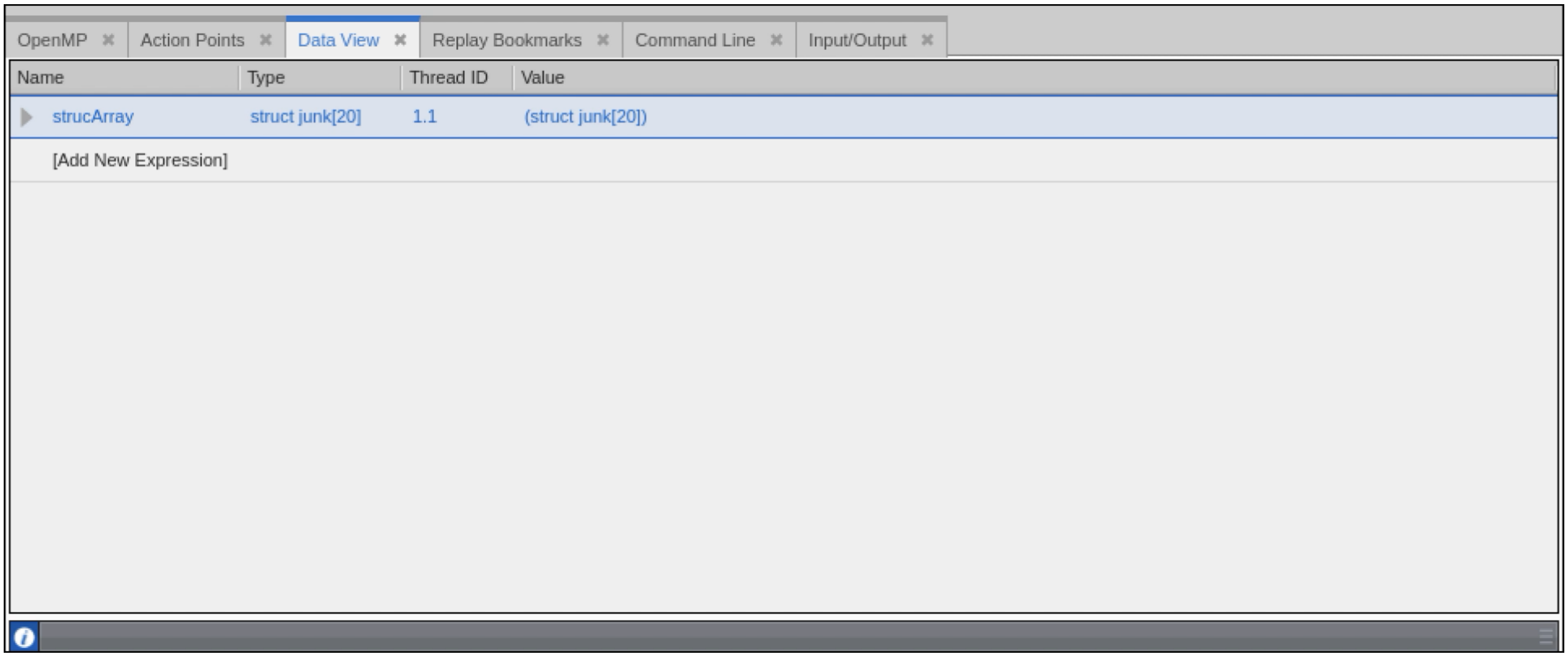
Dive on a single element to view individual data in the Data View





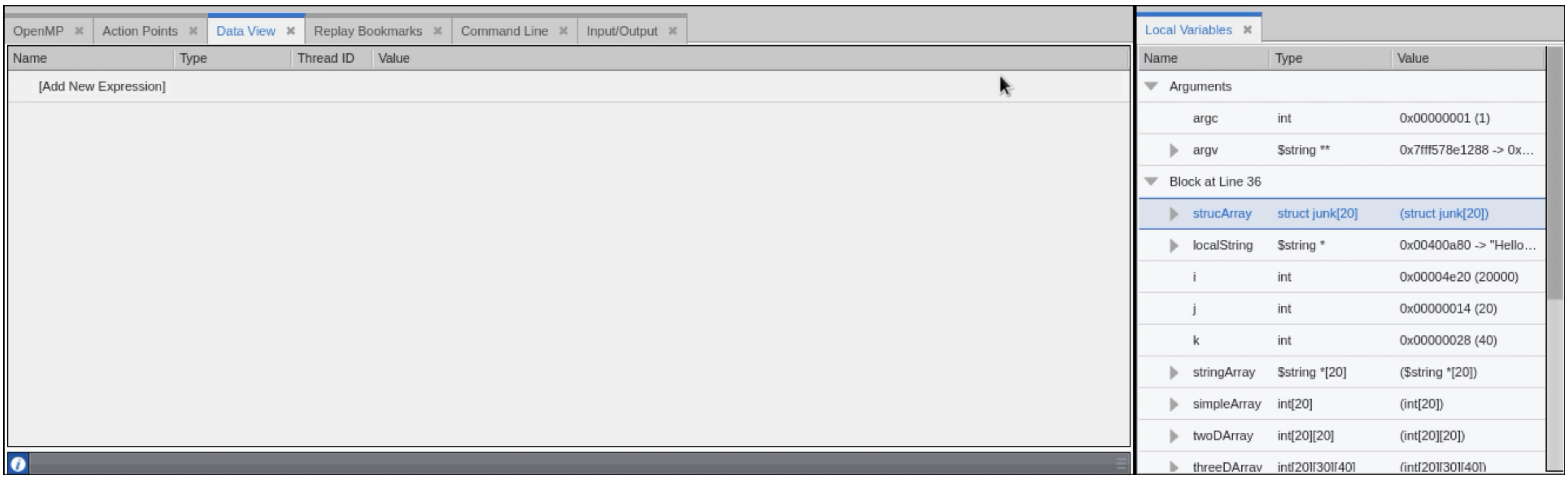
The Data View – Dive in All

- Dive in All
  - Use Dive in All to easily see each member of a data structure from an array of structures



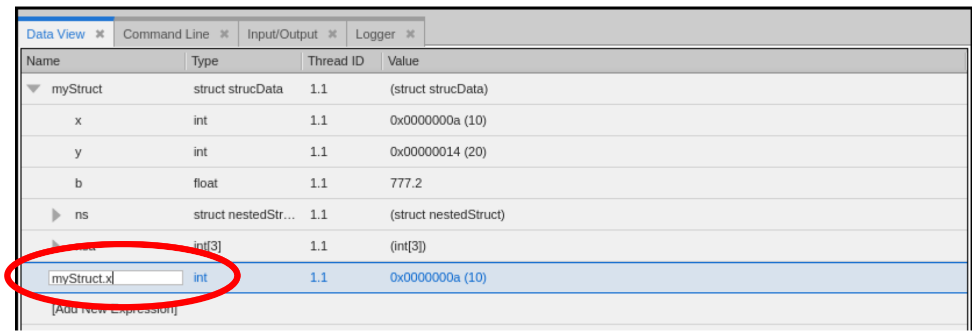
The Data View – Dive in New Data Window

- Dive in New Data Window
    - Use Dive in New Data Window add data structures to new Data Views for focused data debugging

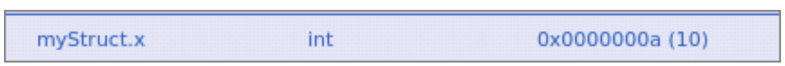


The Data View Panel – Entering Expressions

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



Type the expression in the [Add New expression] field

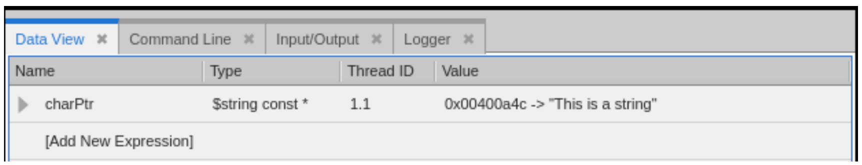


A new expression is added

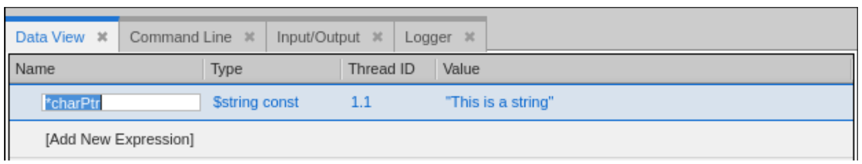
Increment a variable

The Data View Panel – Dereferencing a Pointer

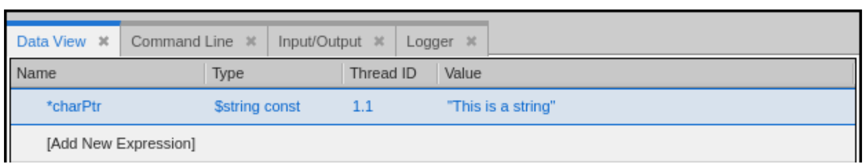
Dereferencing a pointer



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

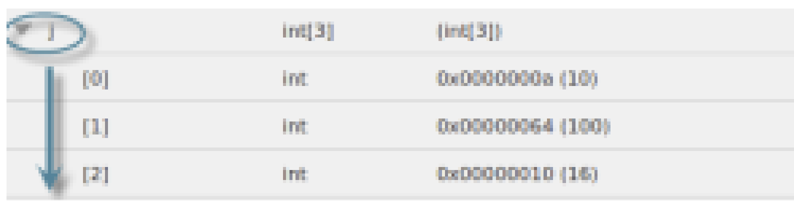


The Data View displays the variables value

The Data View Panel - Casting

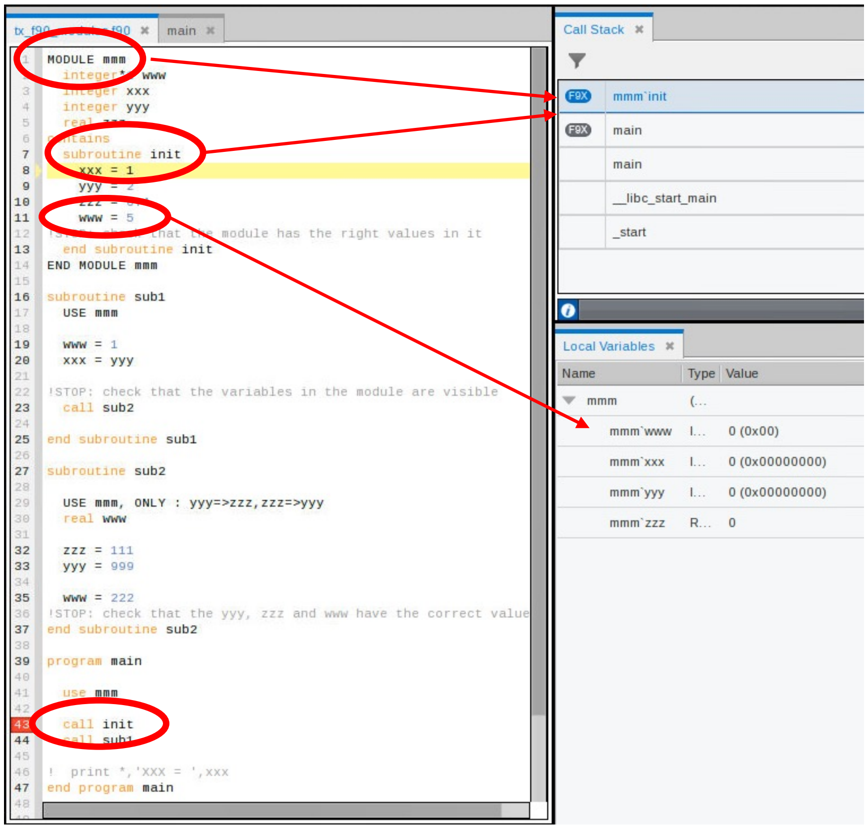
Casting to another type

Cast a variable into an array by adding the array specifier



TotalView displays the array

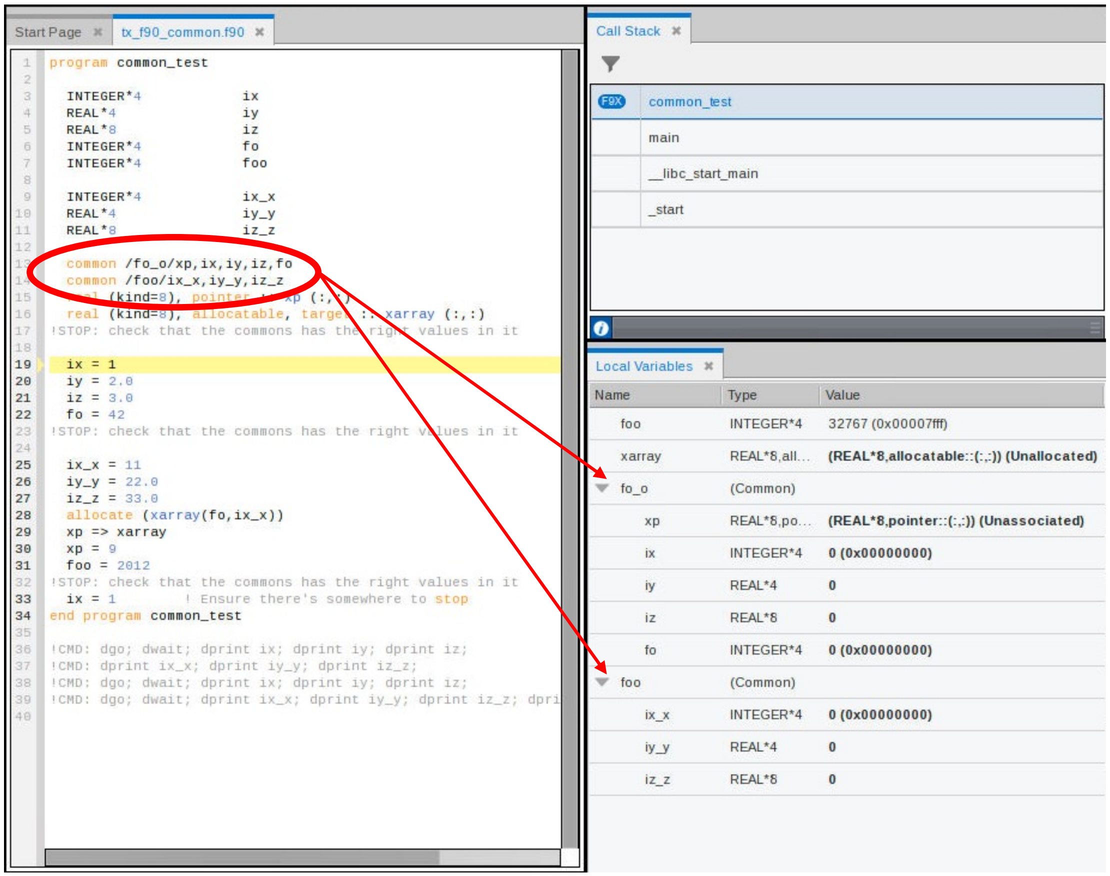
Viewing Data in Fortran



The qualified subroutine name appears in the Call Stack view.

The qualified variable names appear in the Local Variable panel.

Fortran Common Blocks

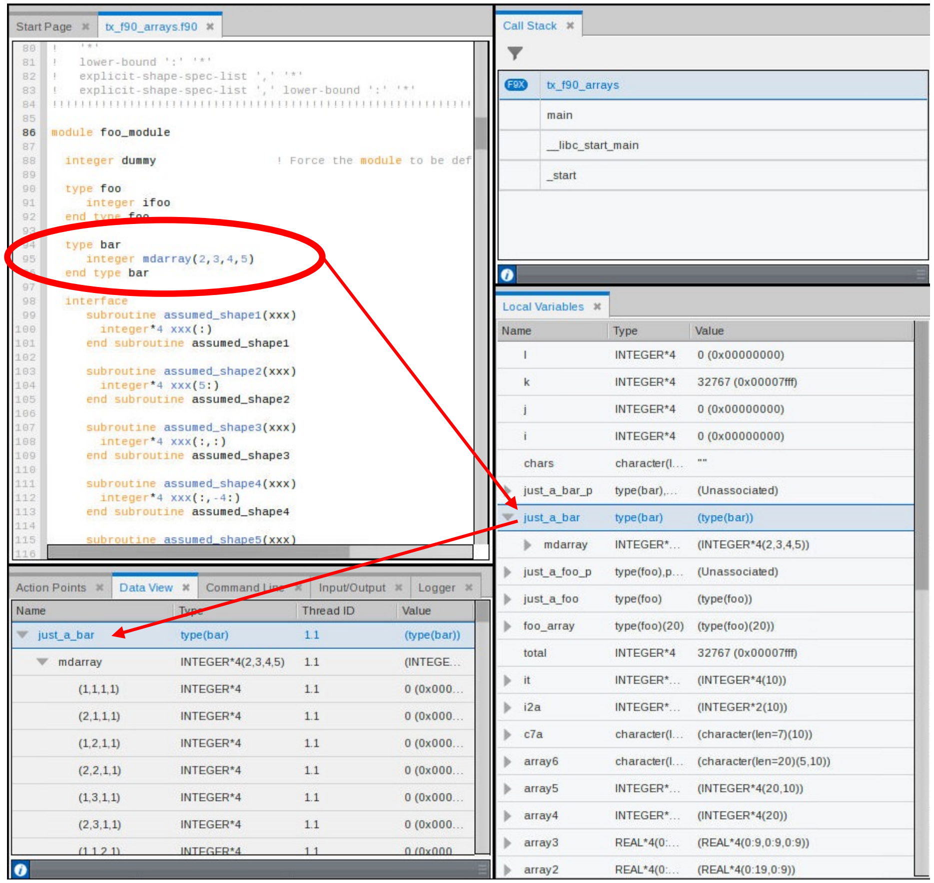


For each common block defined in the scope of a subroutine or function, TotalView creates an entry in that function's common block list.

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

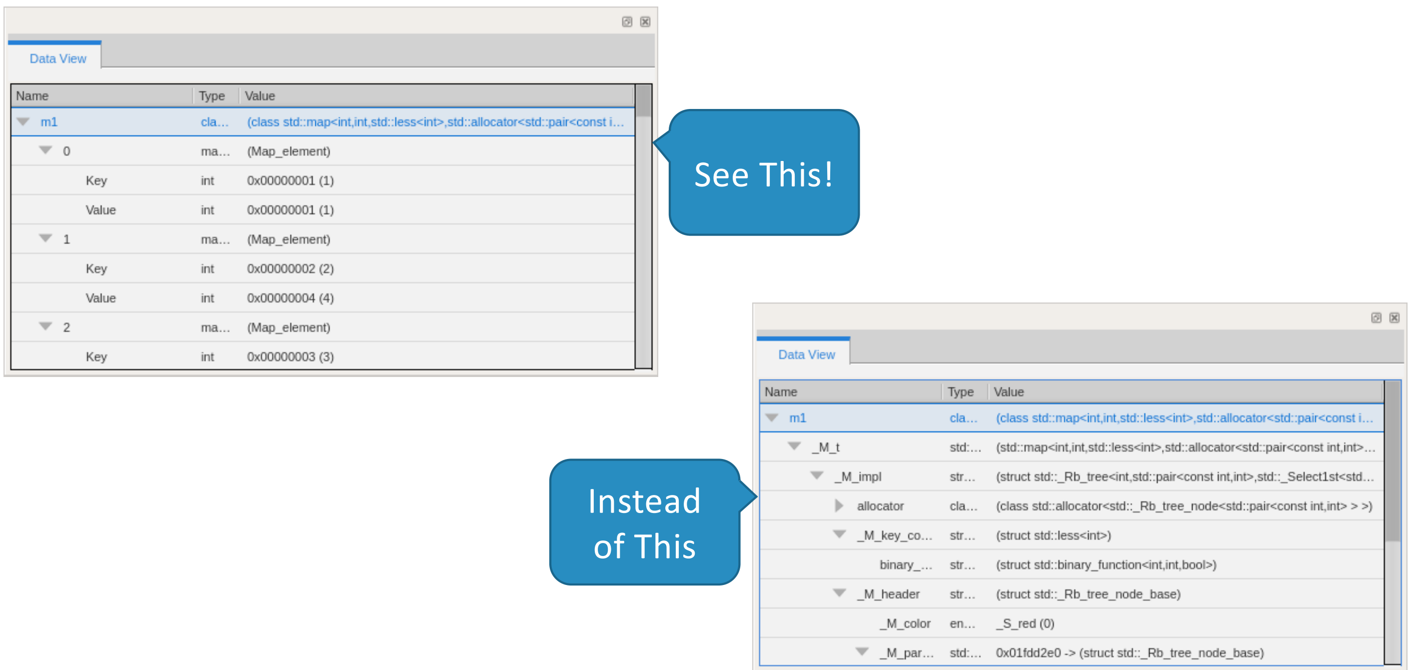


TotalView displays user-defined types in the Local Variables panel, which you can then add to the Data View for more detail

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.



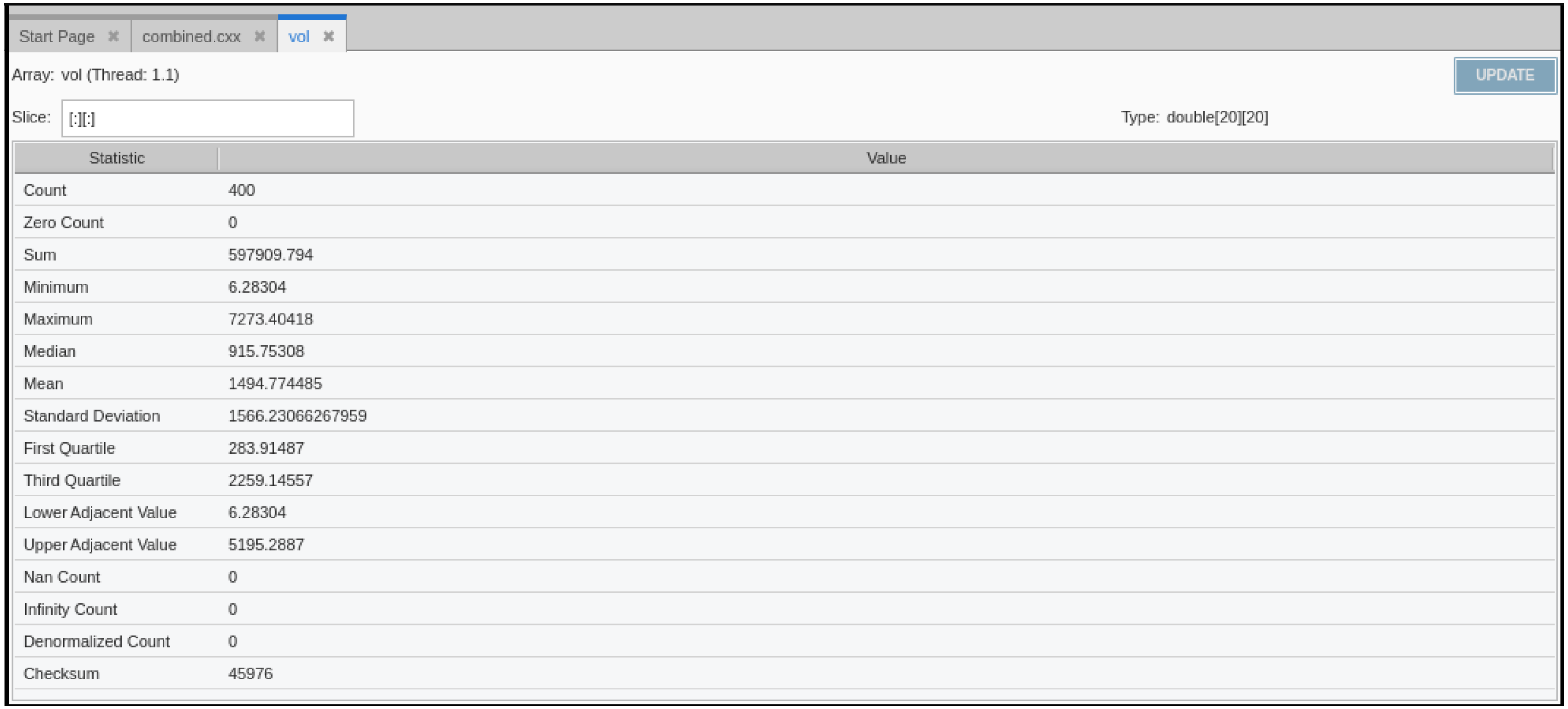
Advanced C++ Support

- 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

```
#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 glambda1 = [](int a, float b) { return a < b; };
   // Two line lambda
   auto glambda2 = [](int a, float && b) {
     if (a < b)
     return 1;
     if (b>a)
     return -1;
     return 0;
   };

   bool b = glambda1(3, 3.14);
   int i = glambda2(3, 3.14);
   for (int i=0; i<10;i++)
     b = glambda1(i, 3.14+i);


   std::function<double(double)> f0    = [](double x){
     return 1;};
   auto                          f1    = [](double x){
     return x;};
   decltype(f0)                  fa[3] = {f0,f1,[](double x){
```

Array Statistics

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



Statistic	Value
Count	400
Zero Count	0
Sum	597909.794
Minimum	6.28304
Maximum	7273.40418
Median	915.75308
Mean	1494.774485
Standard Deviation	1566.23066267959
First Quartile	283.91487
Third Quartile	2259.14557
Lower Adjacent Value	6.28304
Upper Adjacent Value	5195.2887
Nan Count	0
Infinity Count	0
Denormalized Count	0
Checksum	45976

Demo

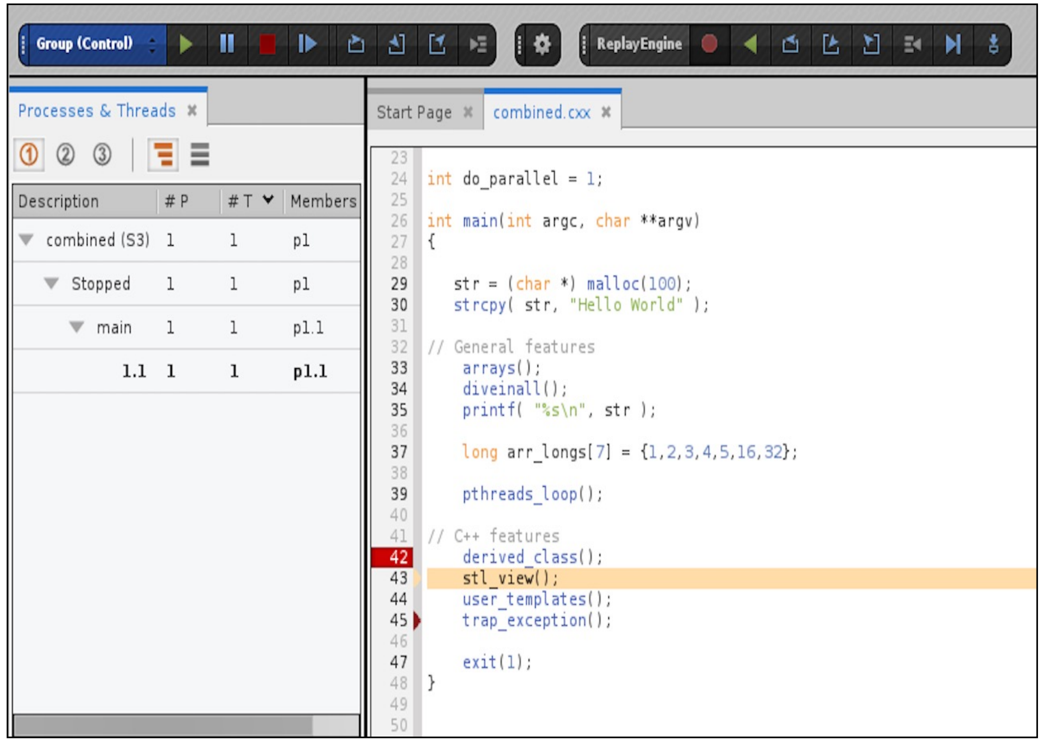
- TotalView STL types (Combined)
- TotalView dive in all demo (Combined)

Q&A

TotalView Reverse Debugging

Reverse debugging

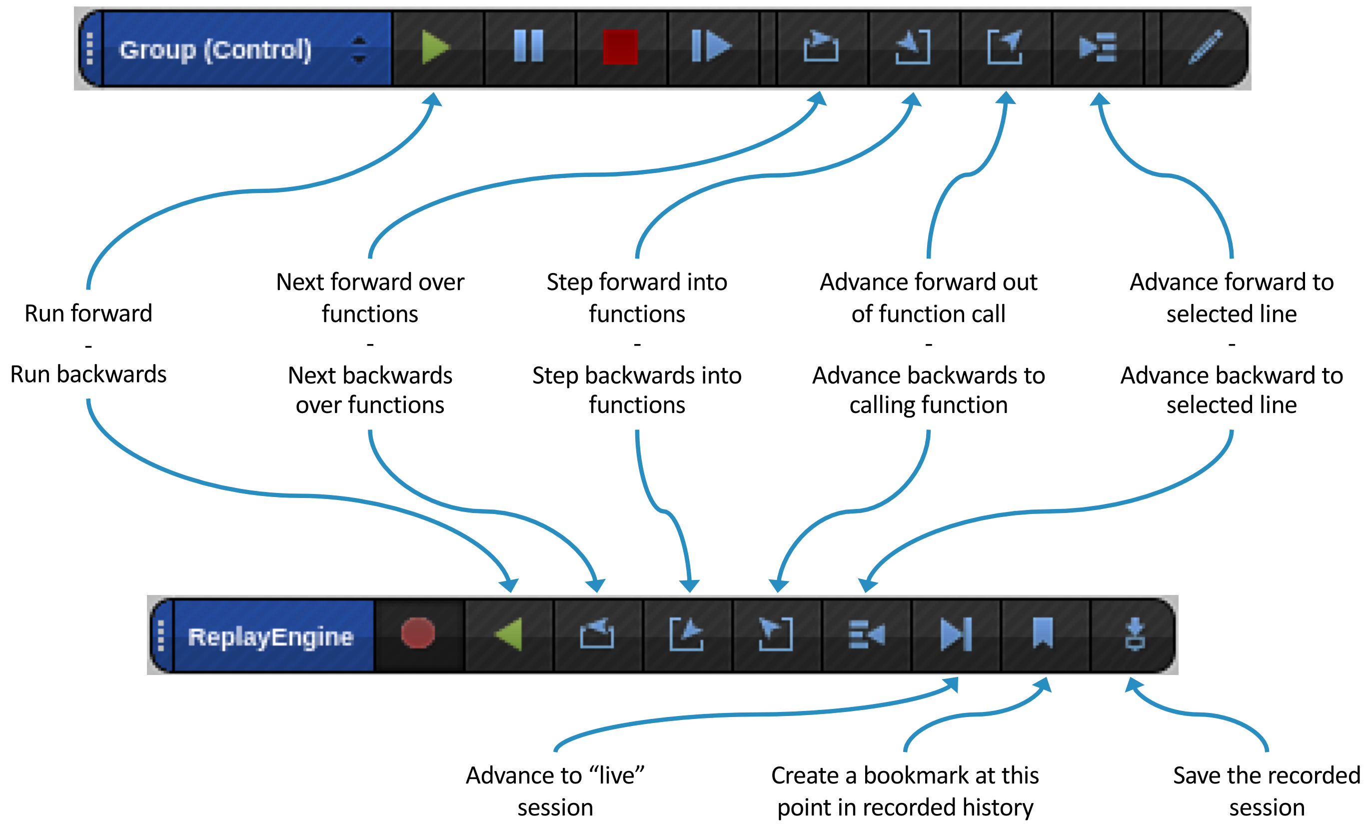
- How do you isolate an intermittent failure?
  - Without TotalView
    - Set a breakpoint in code
    - Realize you ran past the problem
    - Re-load
    - Set breakpoint earlier
    - Hope it fails
    - Keep repeating
  - With TotalView
    - Set a breakpoint
    - Start recording
    - See failure
    - 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, ether 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

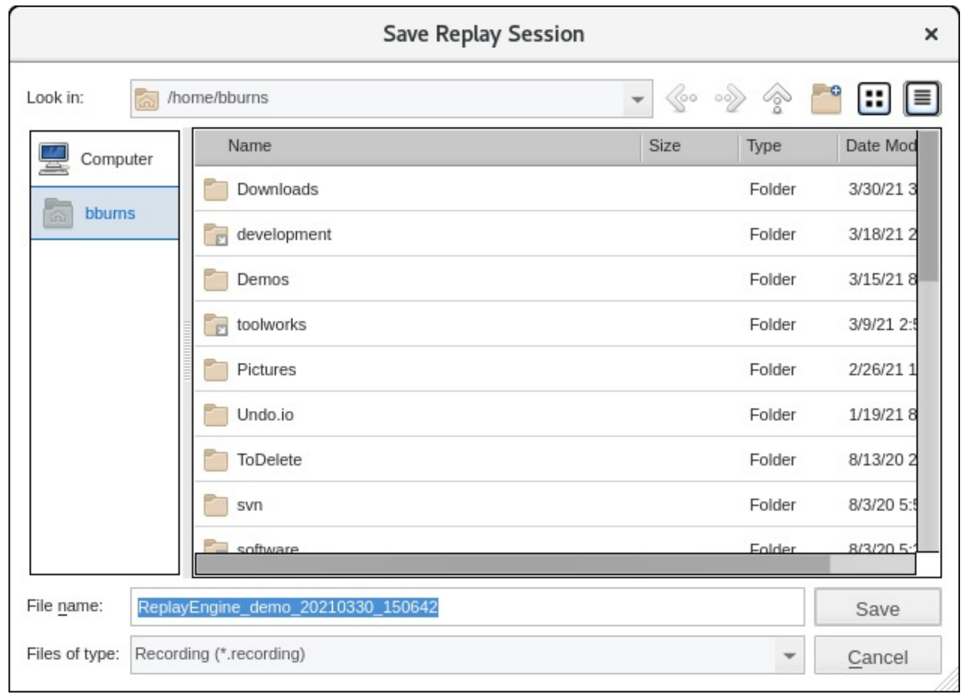
Reverse Debugging Controls



Saving and Loading Execution History

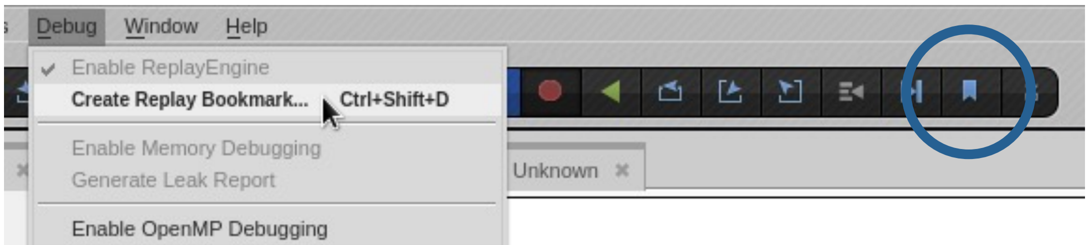
- TotalView can save the current ReplayEngine execution history to file at any time
- The saved recording can be loaded into TotalView using any of the following:
  - At startup, using the same syntax as when opening a core file:

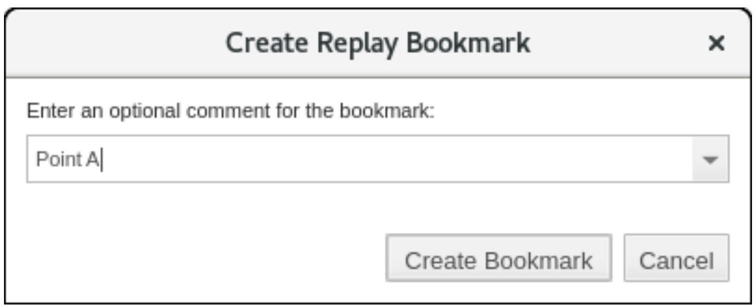
    totalview -newUI *executable recording-file*
  - On the Start Page view by selecting Load Core File or Replay Recording File



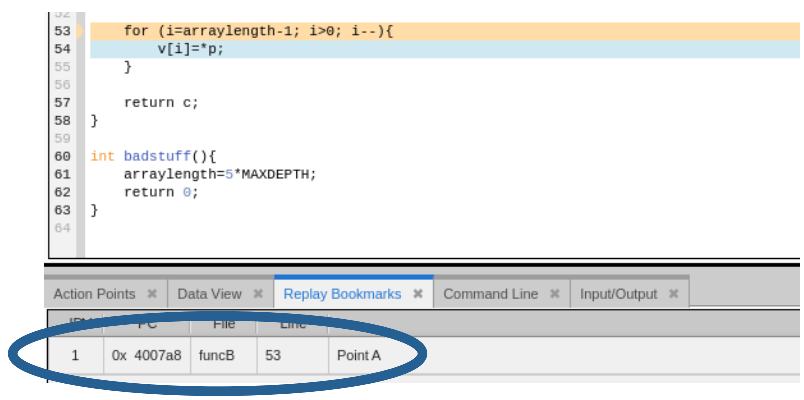
Replay Bookmarks

- 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.

dset TV::replay_history_size *value*

e.g. dset TV::replay_history_size 1024M

- Setting the preferred behaviour. By default, the oldest history is discarded so that recording can continue

dset TV::replay_history_mode **1** (Discard oldest history and continue recording)

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.

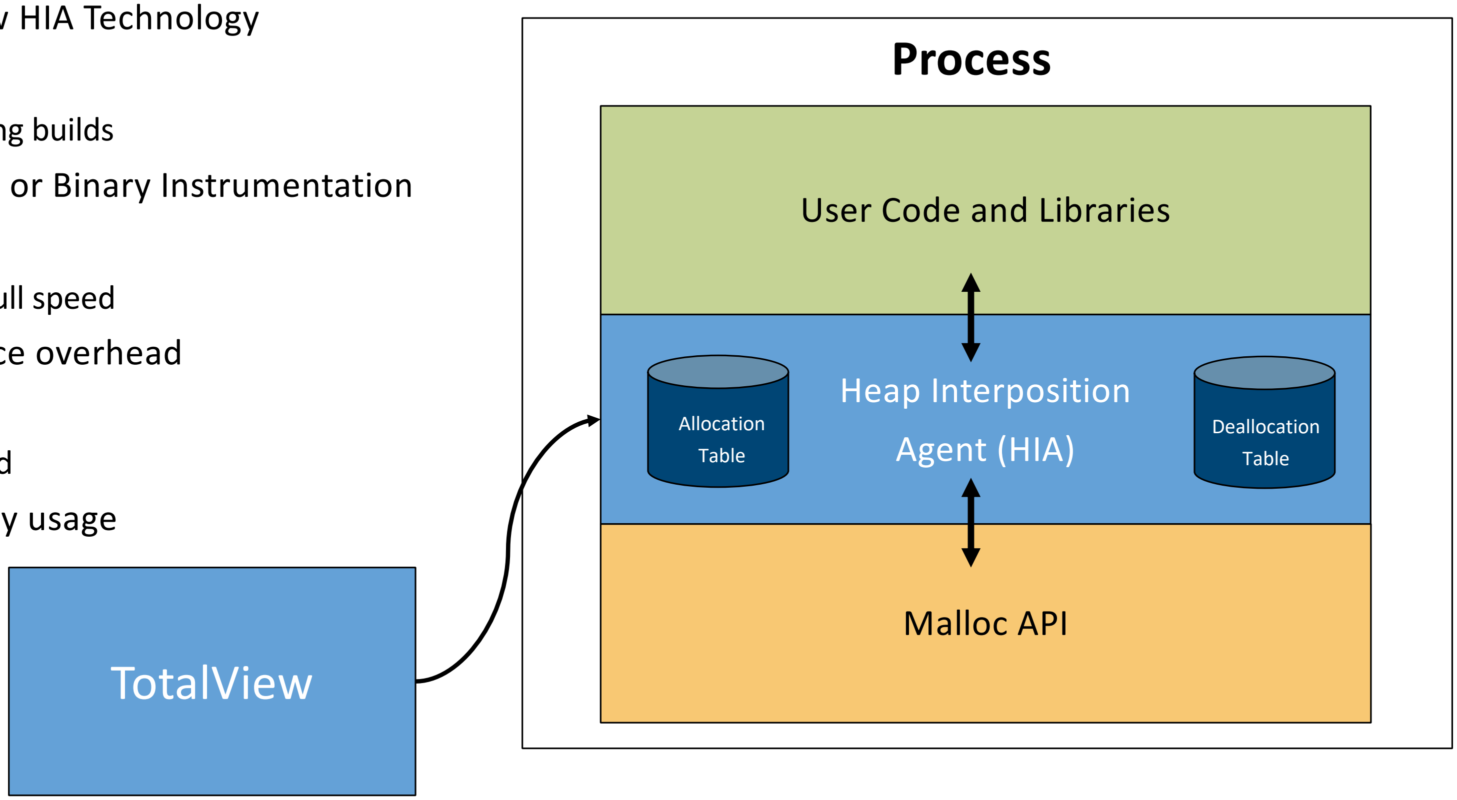
TotalView Memory Debugging

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

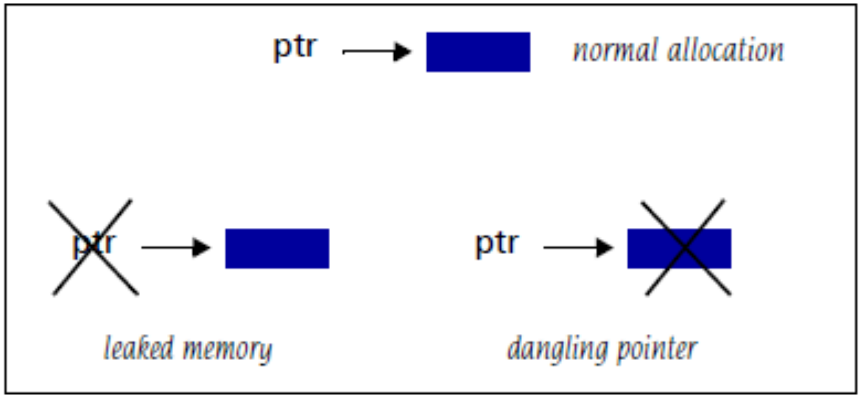
- Advantages of TotalView HIA Technology
  - Use it with your existing builds
    - No Source Code or Binary Instrumentation
  - Programs run nearly full speed
    - Low performance overhead
  - Low memory overhead
    - Efficient memory usage



Memory Debugging Features

TotalView Memory Debugging Features

- Leak detection
- Heap usage
- Dangling pointer detection



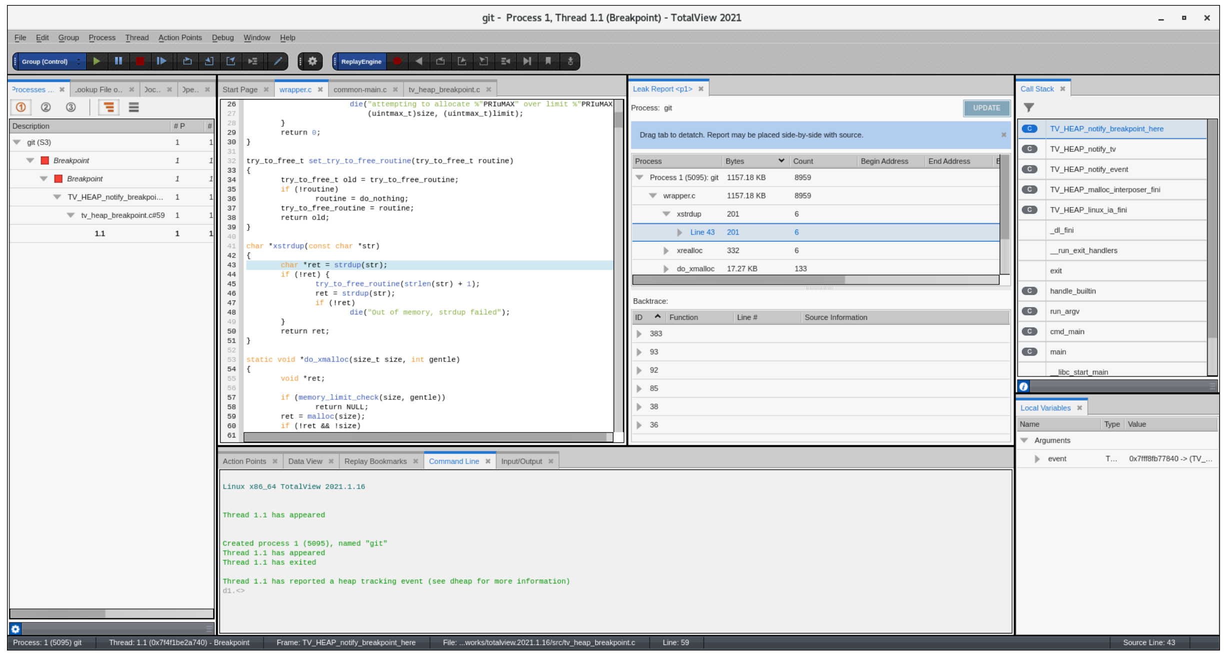
Coming Features

- 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

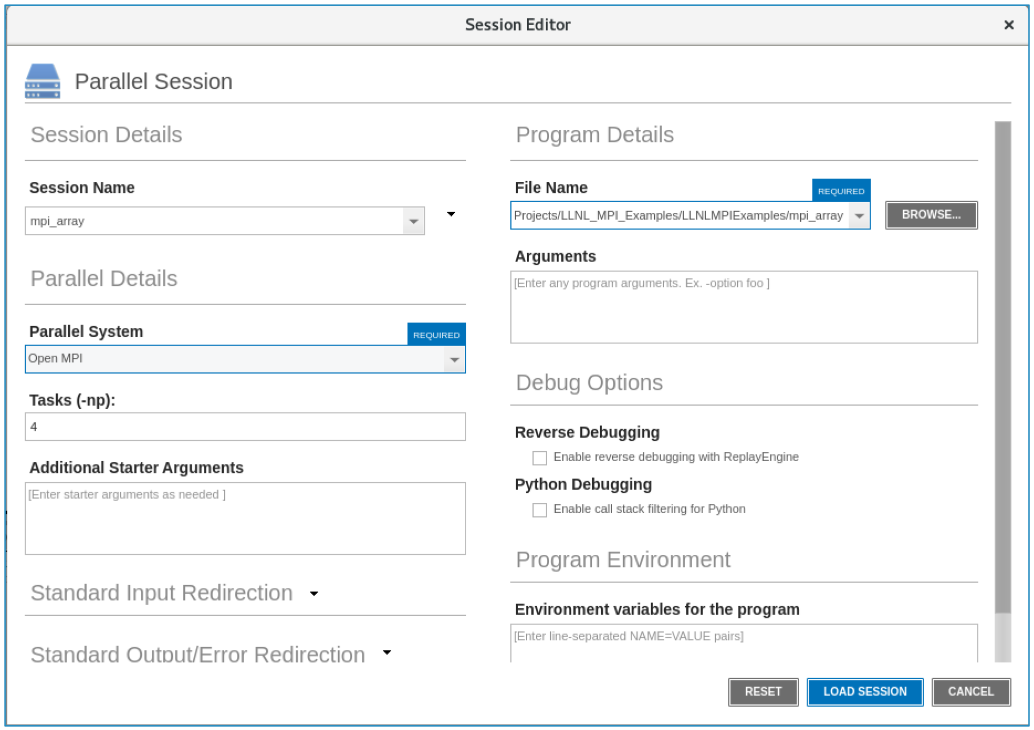
Debugging Parallel Applications

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

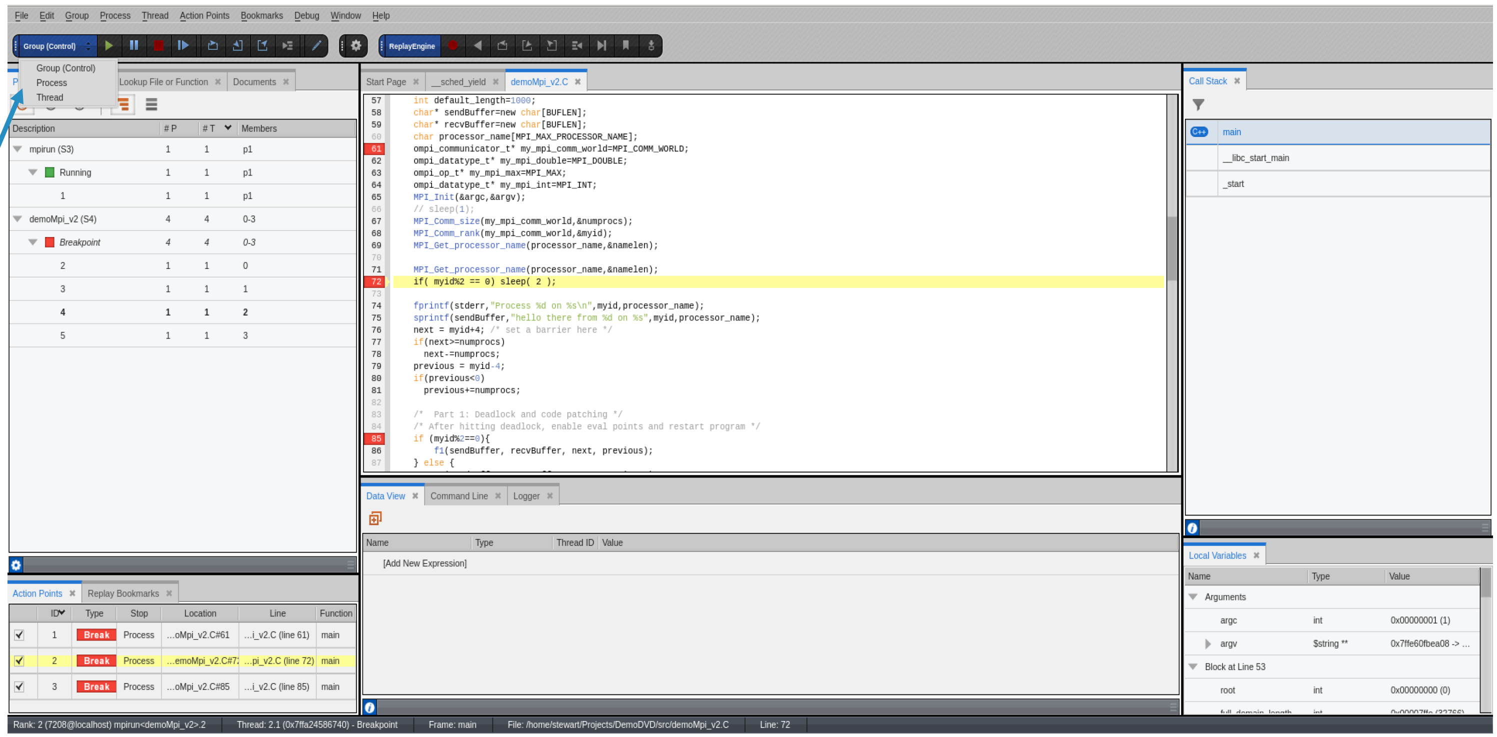
General Command Line: `totalview --args <starter> -n ## <partition> myprogram`

MPI	Startup Command
IBM PowerLE (@LLNL)	totalview --args lrun -n 16 myprog
Linux under SLURM	totalview --args srun -n 16 -p pdebug myprog
Open MPI / MPICH / Intel MPI	totalview --args mpirun -np 16 myprog

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



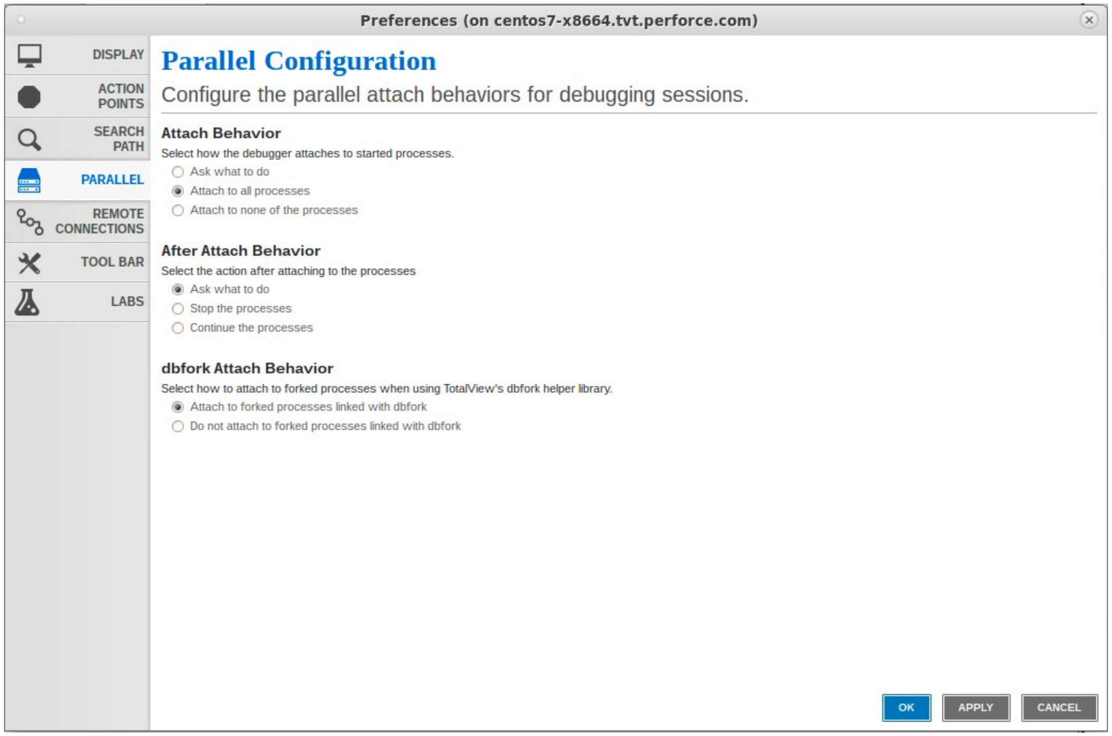
Select either

- 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)

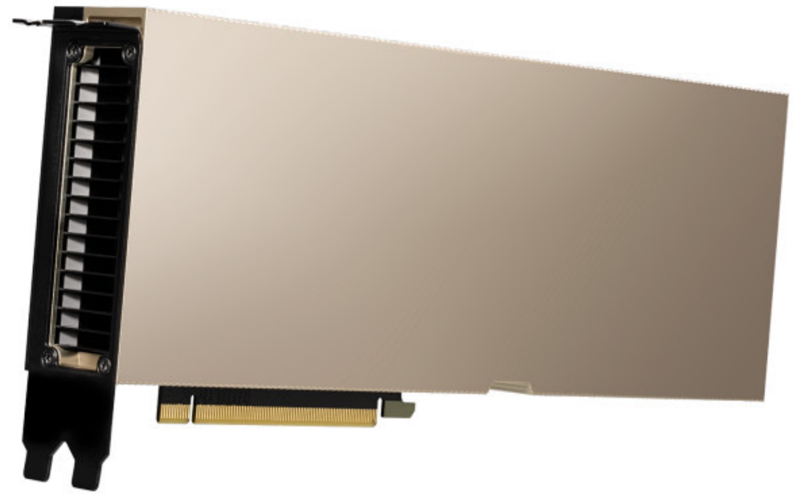
  shalverson@nvidia.com

- Andrew Gontarek (NVIDIA Software Engineer – Devtools Compute Debugger)

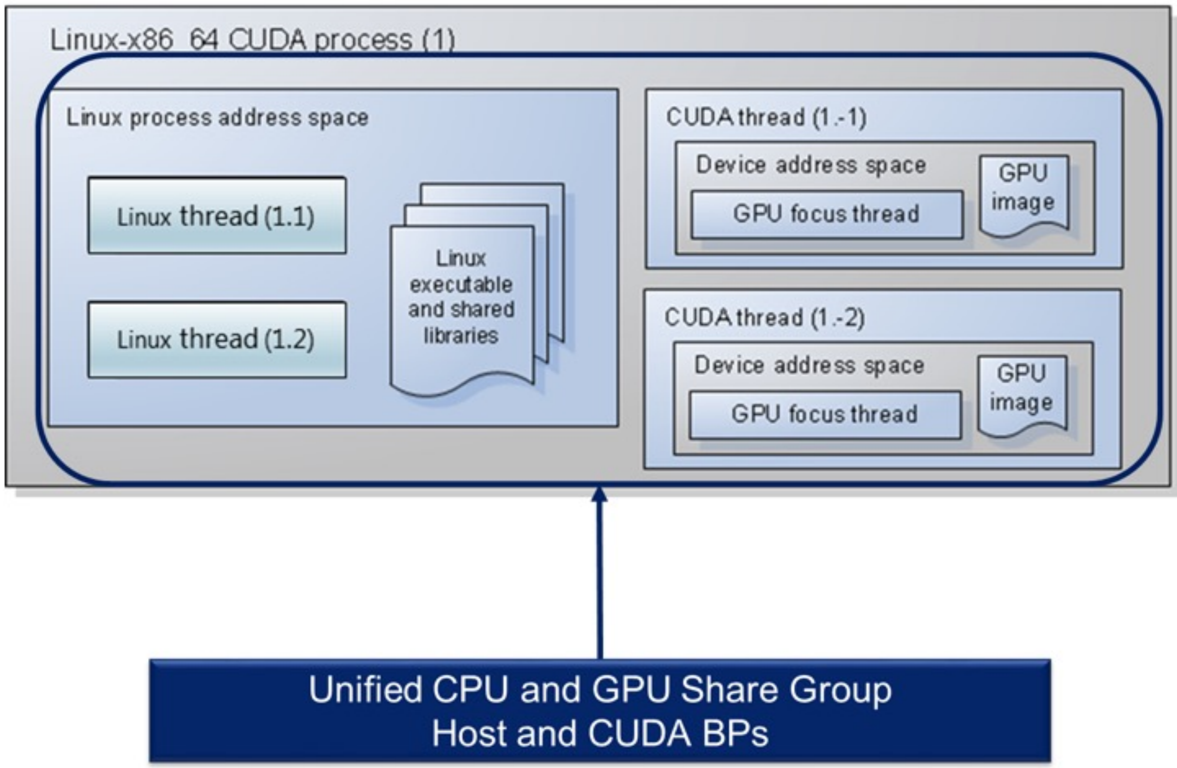
  agontarek@nvidia.com

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

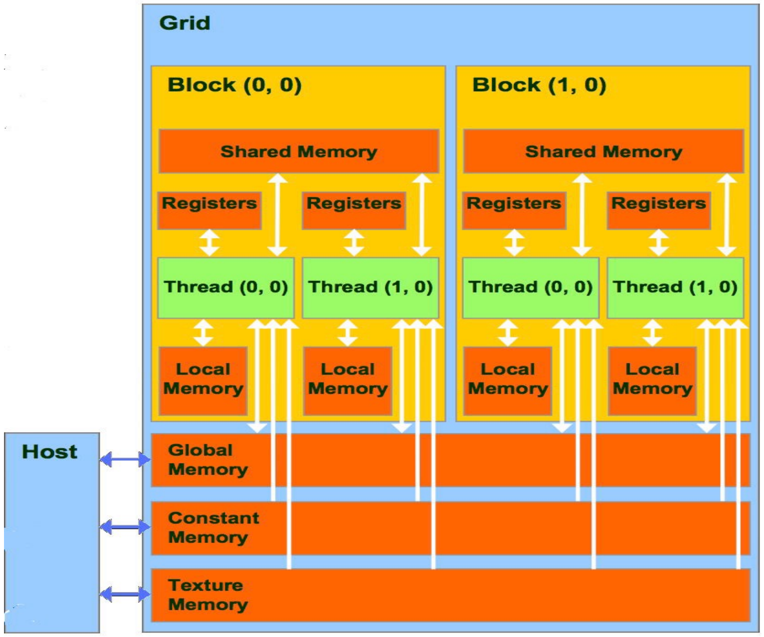


TotalView CUDA Debugging Model



GPU Memory Hierarchy

- Hierarchical memory
  - Local (thread)
    - Local
    - Register
  - Shared (block)
  - Global (GPU)
    - Global
    - Constant
    - Texture
  - System (host)



Supported Type Storage Qualifiers

@generic	An offset within generic storage
@frame	An offset within frame storage
@global	An offset within global storage
@local	An offset within local storage
@parameter	An offset within parameter storage
@iparam	Input parameter
@oparam	Output parameter
@shared	An offset within shared storage
@surface	An offset within surface storage
@texsampler	An offset within texture sampler storage
@texture	An offset within texture storage
@rtvar	Built-in runtime variables
@register	A PTX register name
@sregister	A PTX special register name

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.

```
% /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

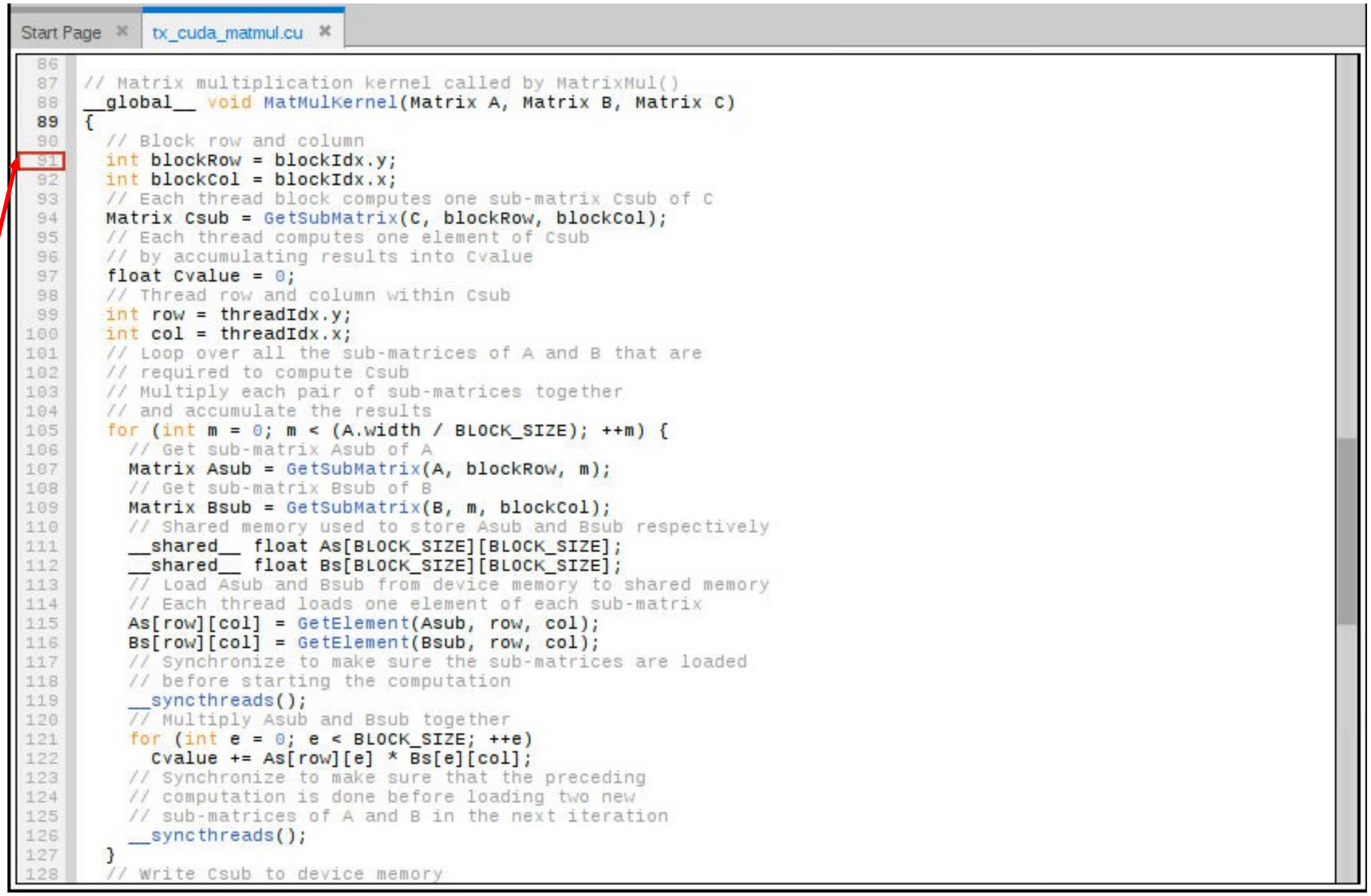
Start Page | tx_cuda_matmul.cu

```
  Matrix A;
  A.width = width_;
  A.height = height_;
  A.stride = width_;
  A.elements = (float*) malloc(sizeof(*A.elements) * width_ * height_);
  for (int row = 0; row < height_; row++)
    for (int col = 0; col < width_; col++)
      A.elements[row * width_ + col] = row * 10.0 + col;
  return A;
}

static void
print_Matrix (Matrix A, const char *name)
{
  printf("%s:\n", name);
  for (int row = 0; row < A.height; row++)
    for (int col = 0; col < A.width; col++)
      printf ("[%5d][%5d] %f\n", row, col, A.elements[row * A.stride + col]);
}

// Multiply an m*n matrix with an n*p matrix results in an m*p matrix.
// Usage: tx_cuda_matmul [ m [ n [ p ] ] ]
// m, n, and p default to 1, and are multiplied by BLOCK_SIZE.
int main(int argc, char **argv)
{
//  cudaSetDevice(0);
  const int m = BLOCK_SIZE * (argc > 1 ? atoi(argv[1]) : 1);
  const int n = BLOCK_SIZE * (argc > 2 ? atoi(argv[2]) : 1);
  const int p = BLOCK_SIZE * (argc > 3 ? atoi(argv[3]) : 1);
  Matrix A = cons_Matrix(m, n);
  Matrix B = cons_Matrix(n, p);
  Matrix C = cons_Matrix(m, p);
  MatMul(A, B, C);
  print_Matrix(A, "A");
  print_Matrix(B, "B");
  print_Matrix(C, "C");
  return 0;
}

/*
 * Update log
 *
 * Feb 25 2015 NYP: Removed __forceinline__, it is making cli too fast
```

Set Breakpoints in CUDA Kernel Code Before Launch

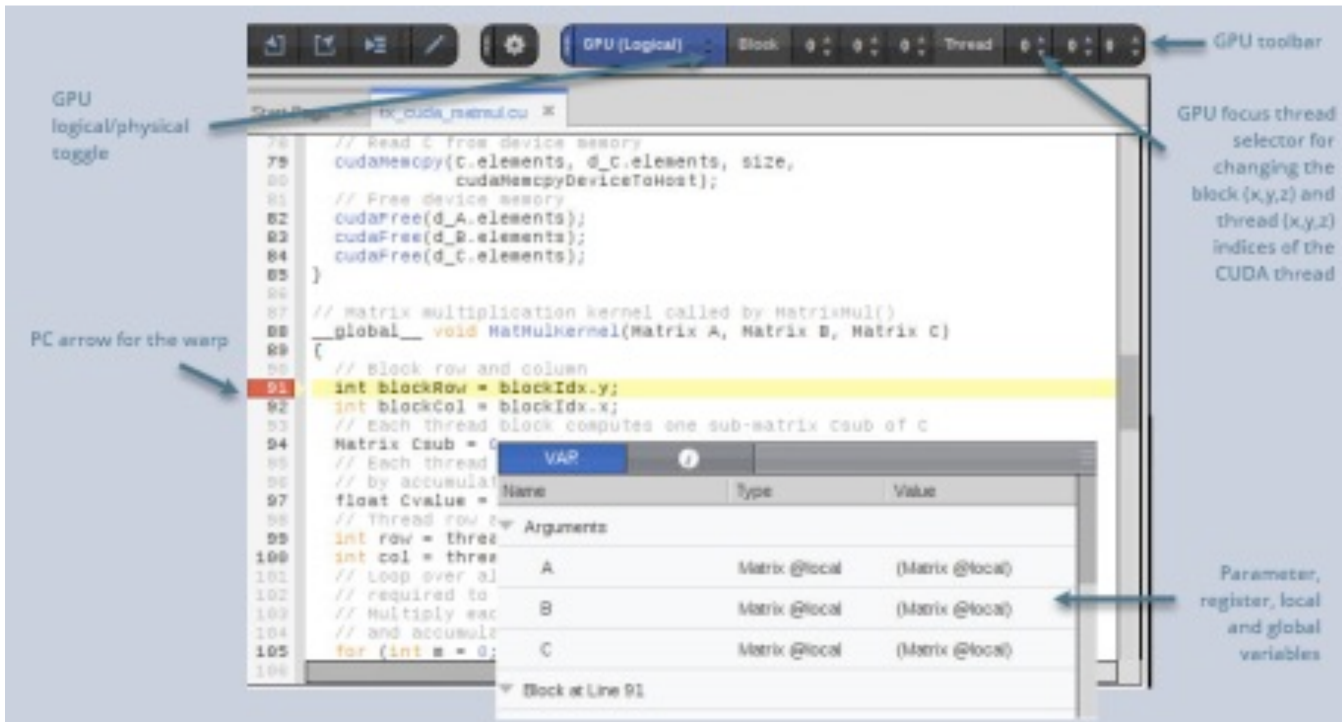
Set breakpoints in the CUDA or OpenMP TARGET region code before you start the process.

Hollow breakpoint indicates a breakpoint will be set when the code is loaded onto the GPU.

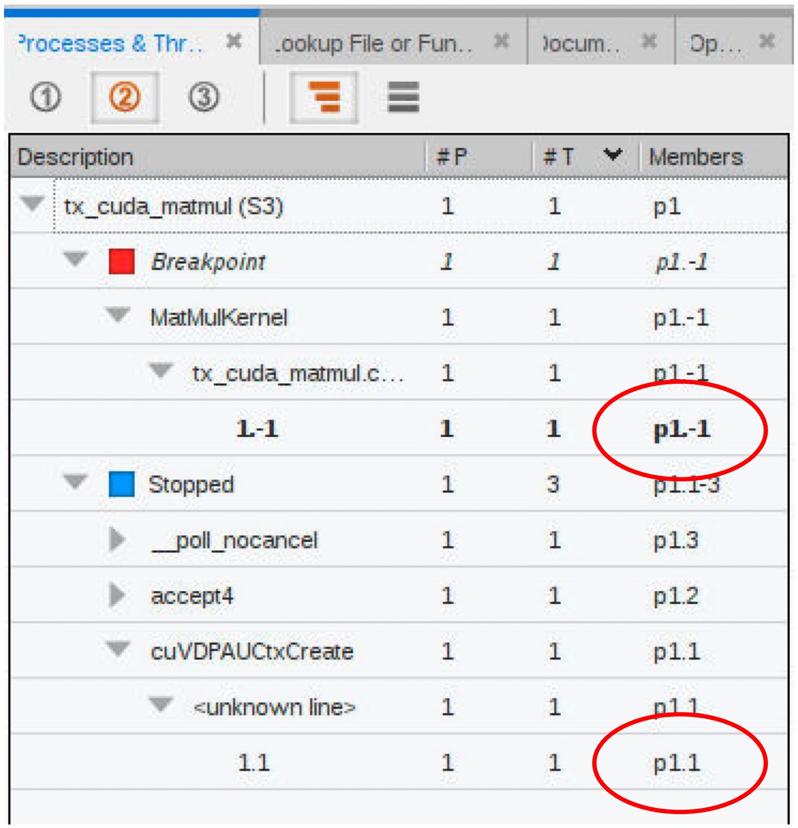


Stopped at a Breakpoint in CUDA Kernel Code

- 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

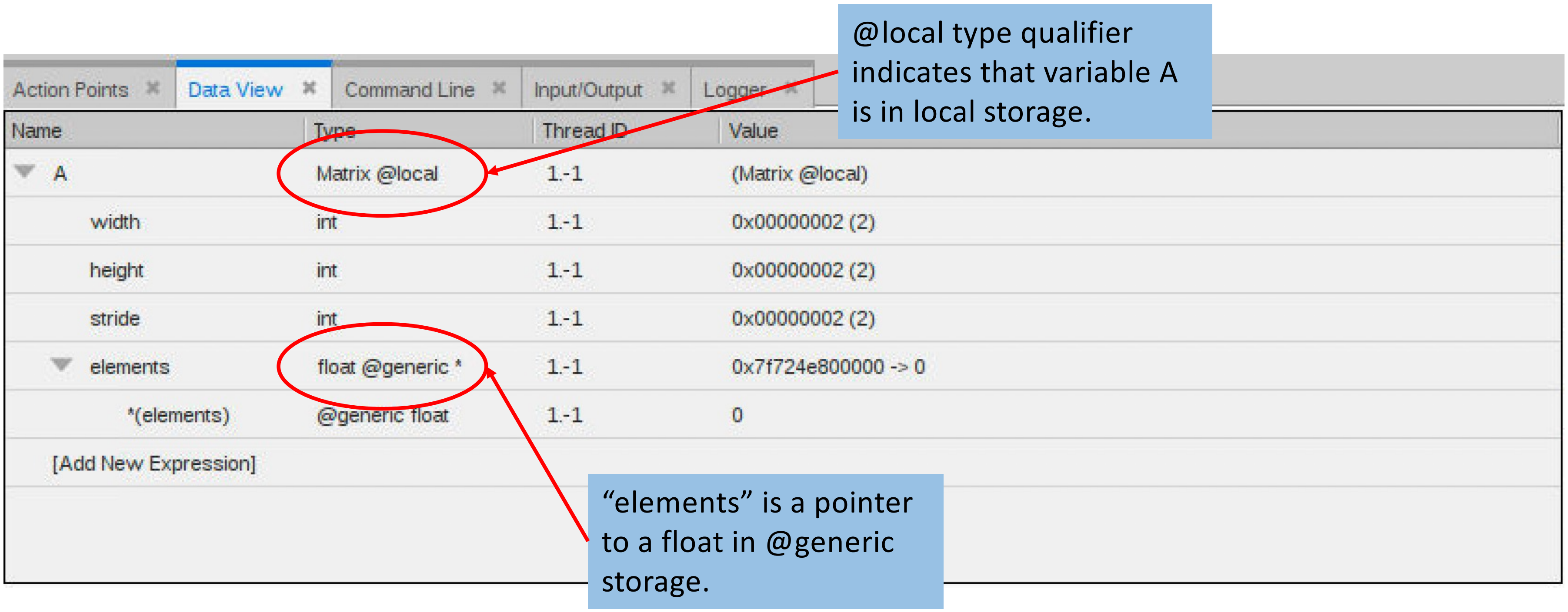
Logical toolbar displays the Block and Thread coordinates.

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.

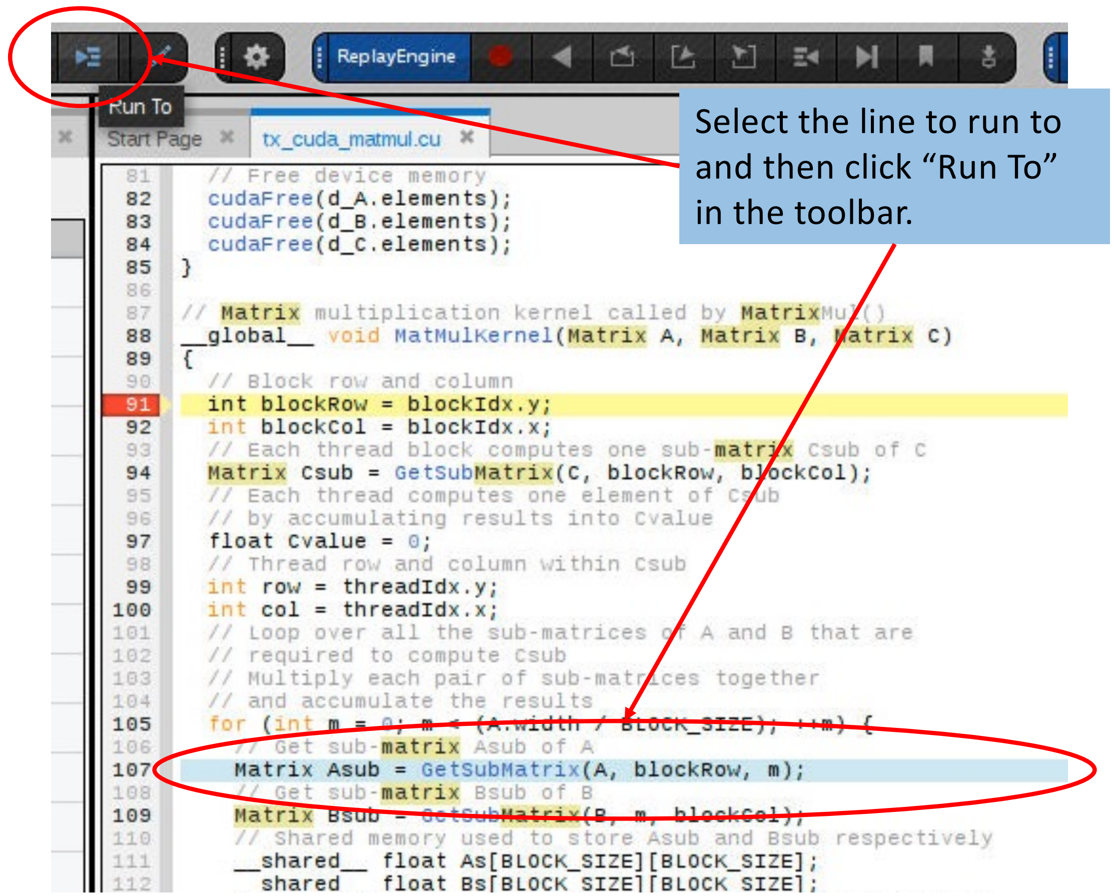
Displaying CUDA Program Variables



- 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

Stepping GPU Code

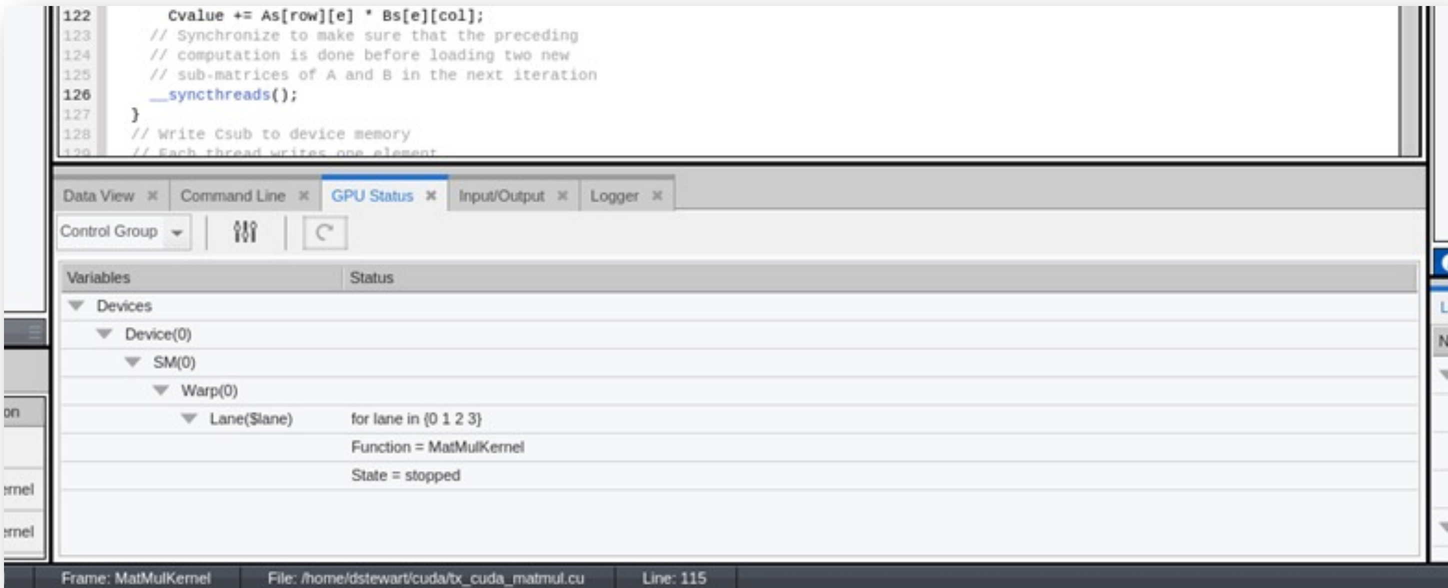
- Single-step operations advance all the GPU hardware lanes in the same warp
- To advance the execution of more than one warp, you may either:
    - Set a breakpoint and continue the process, or
    - Select a line number in the source pane and select "Run To".



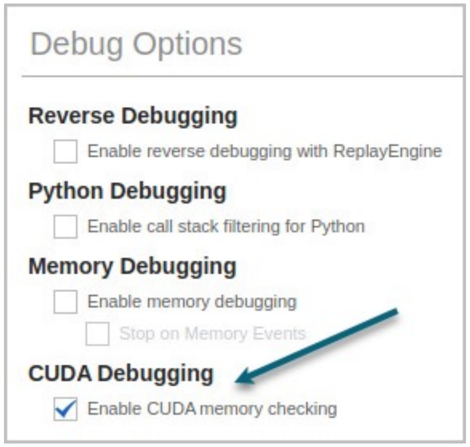
GPU Status View

Displays the state of all the GPUs being debugged.

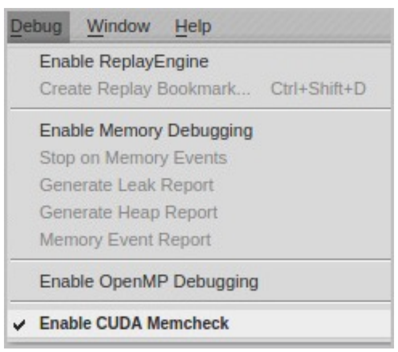
Fully configurable to allow aggregating, sorting and filtering based on physical or logical attributes.



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

Debugging on Perlmutter (Things to Know)

- 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 <u>not</u> 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
  - **`module load totalview`**
- Allocate some nodes, for example
  - **`salloc -A nvendor -C gpu -N 2 -G 8 -t 60 -q interactive_ss11`**
- An interactive shell (bash, csh, etc.) will start inside the allocation
- Start **totalview** on **srun**, for example
  - **`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**

```
#!/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**

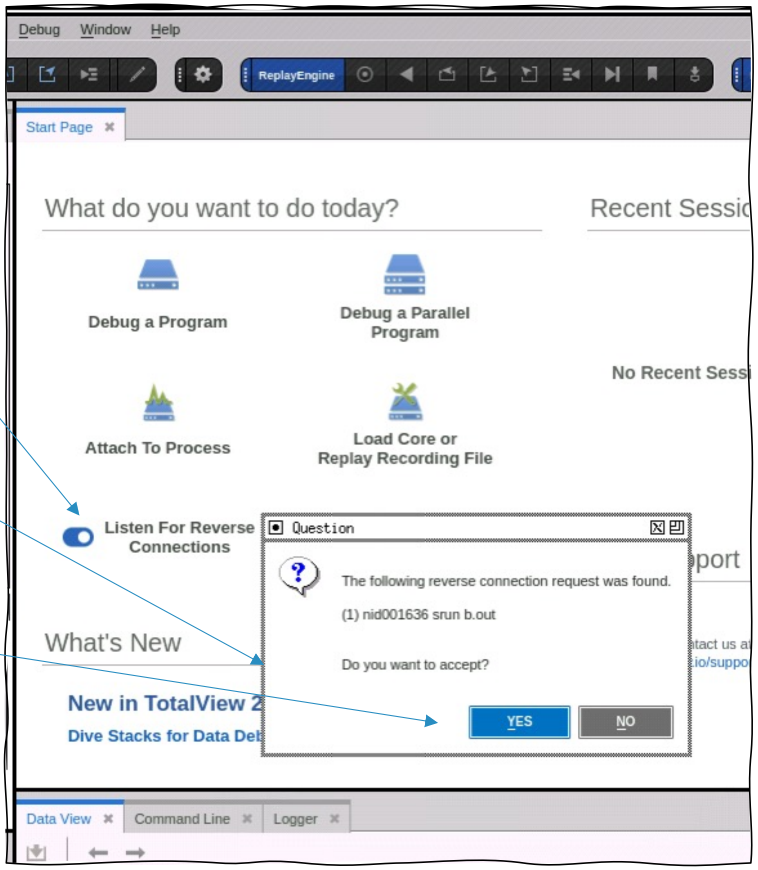
```
module load totalview
totalview
```


Debugging on Perlmutter (Batch Start-up)

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

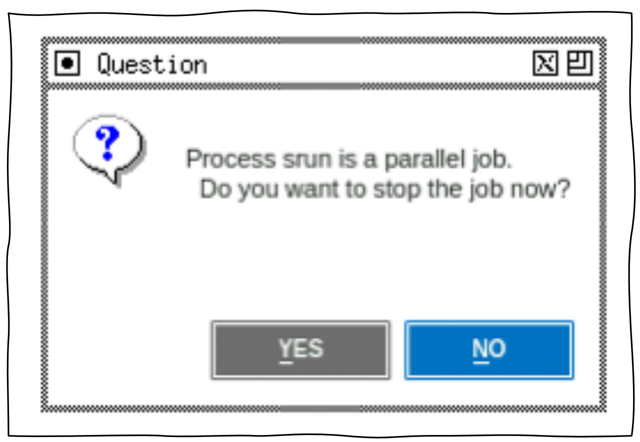
- When the batch script executes the **tvconnect** command, TotalView will post a dialog

- Select "Yes" to connect TotalView to the batch job



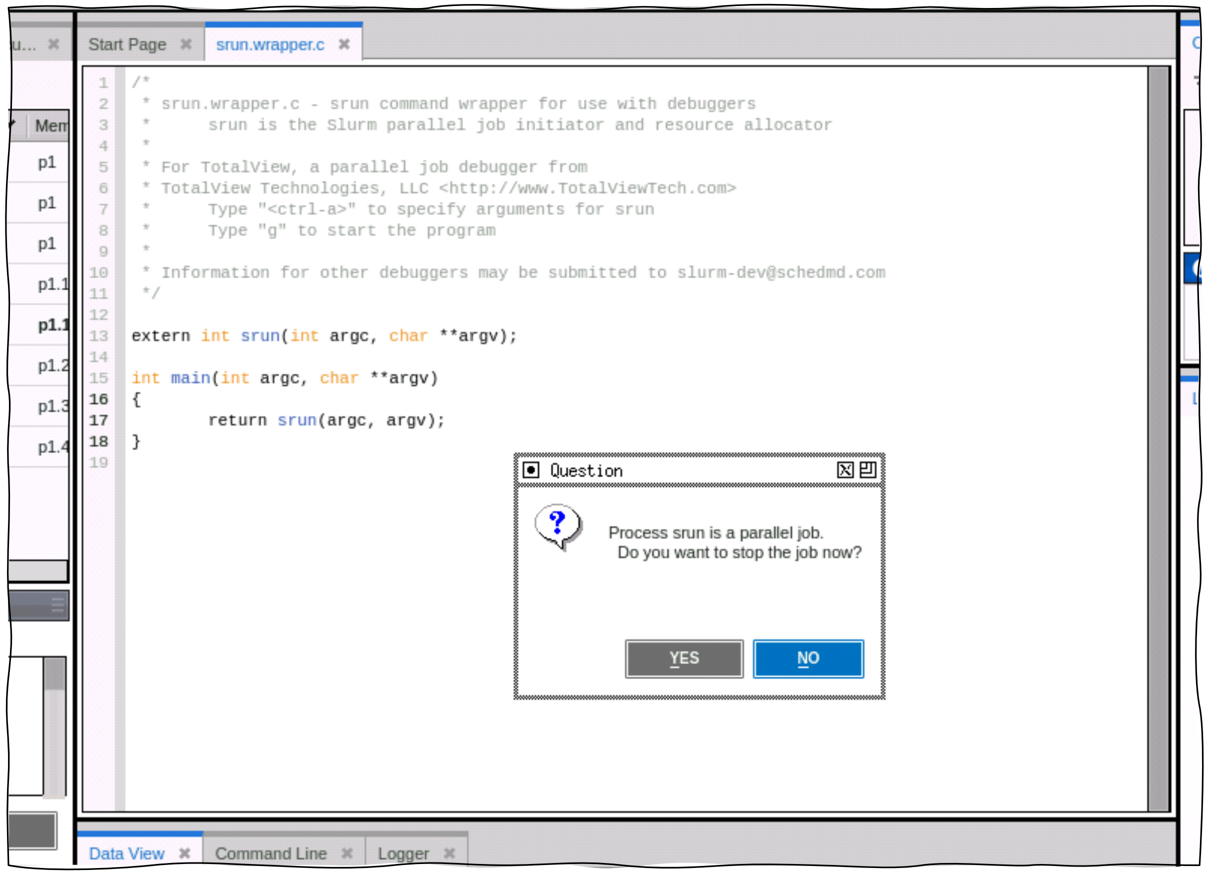
Debugging on Perlmutter (Common to Interactive/Batch)

- 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

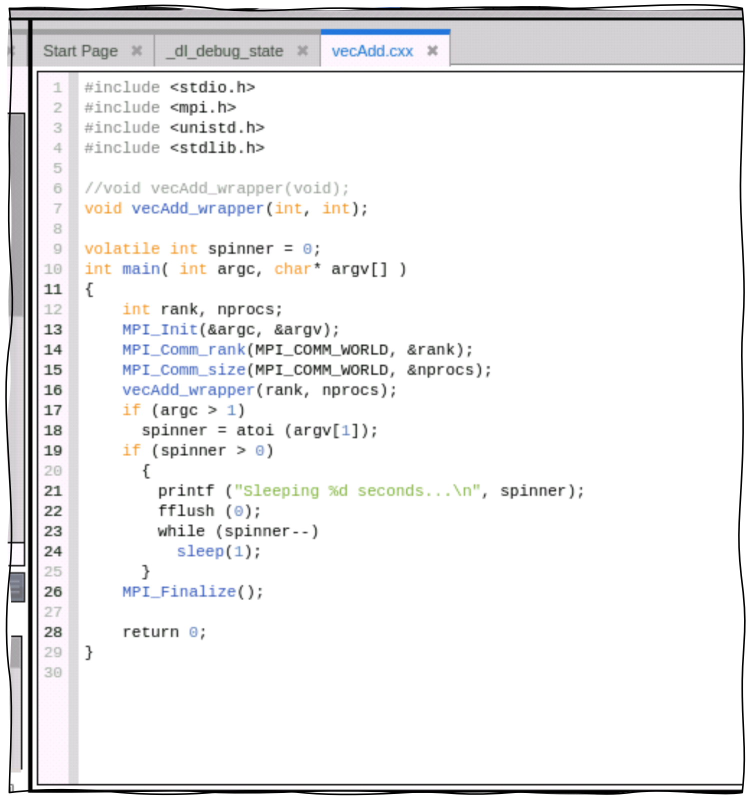


Stop the job when it goes parallel?

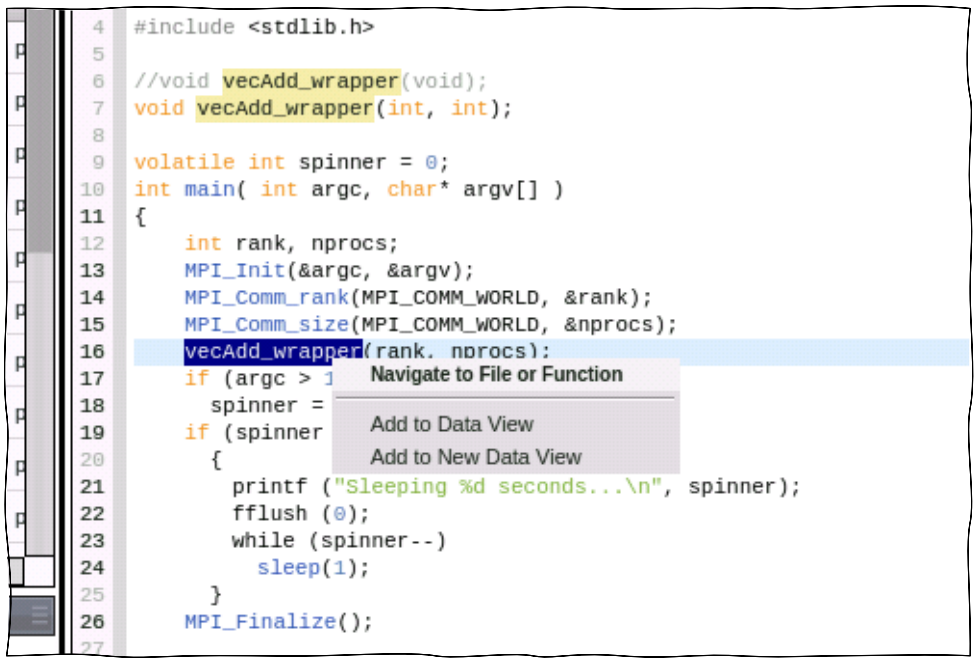
- Click “Yes” to stop the parallel job, which is useful if you want to
    - Navigate to source files / functions
    - Set breakpoints
- Click “No” to allow the job to run, which is useful if you
    - Had saved breakpoints from a previous session
    - Know the program is going to crash (SEGV, etc.)



TotalView will focus on main() in rank 0

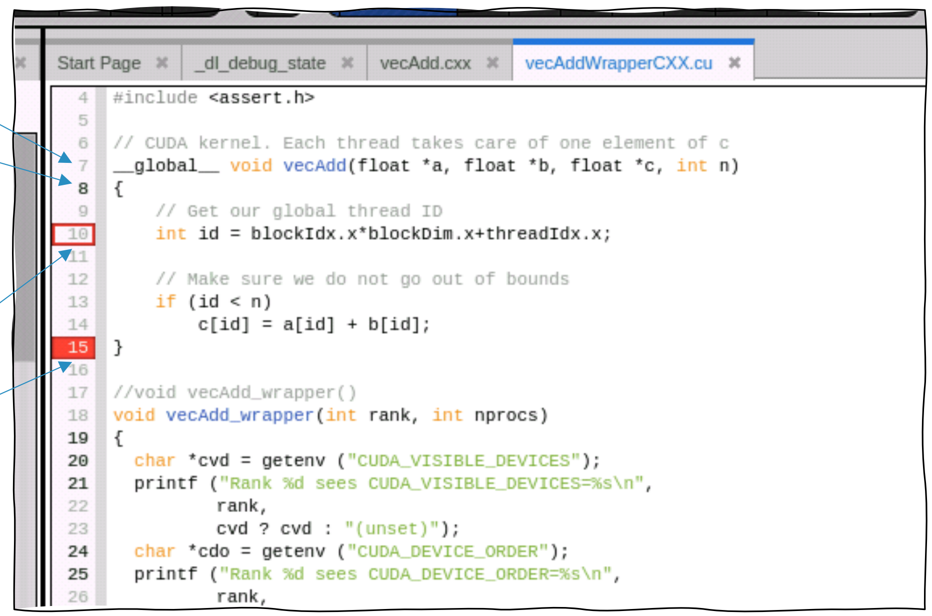


Navigate to a file or function you want to debug

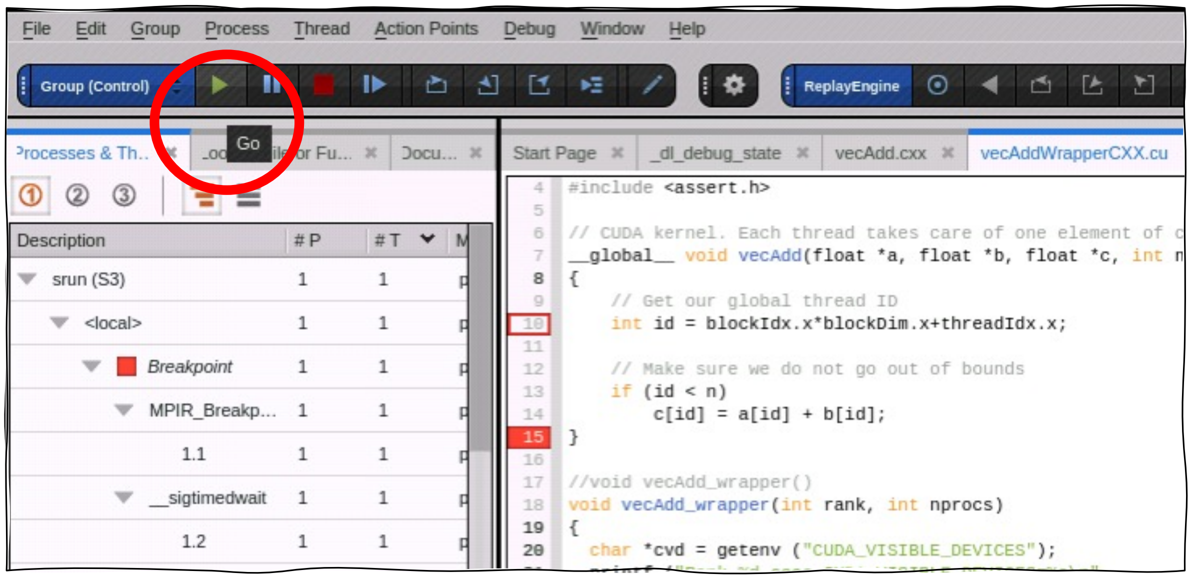


Find the CUDA kernel and select a line number to plant a breakpoint

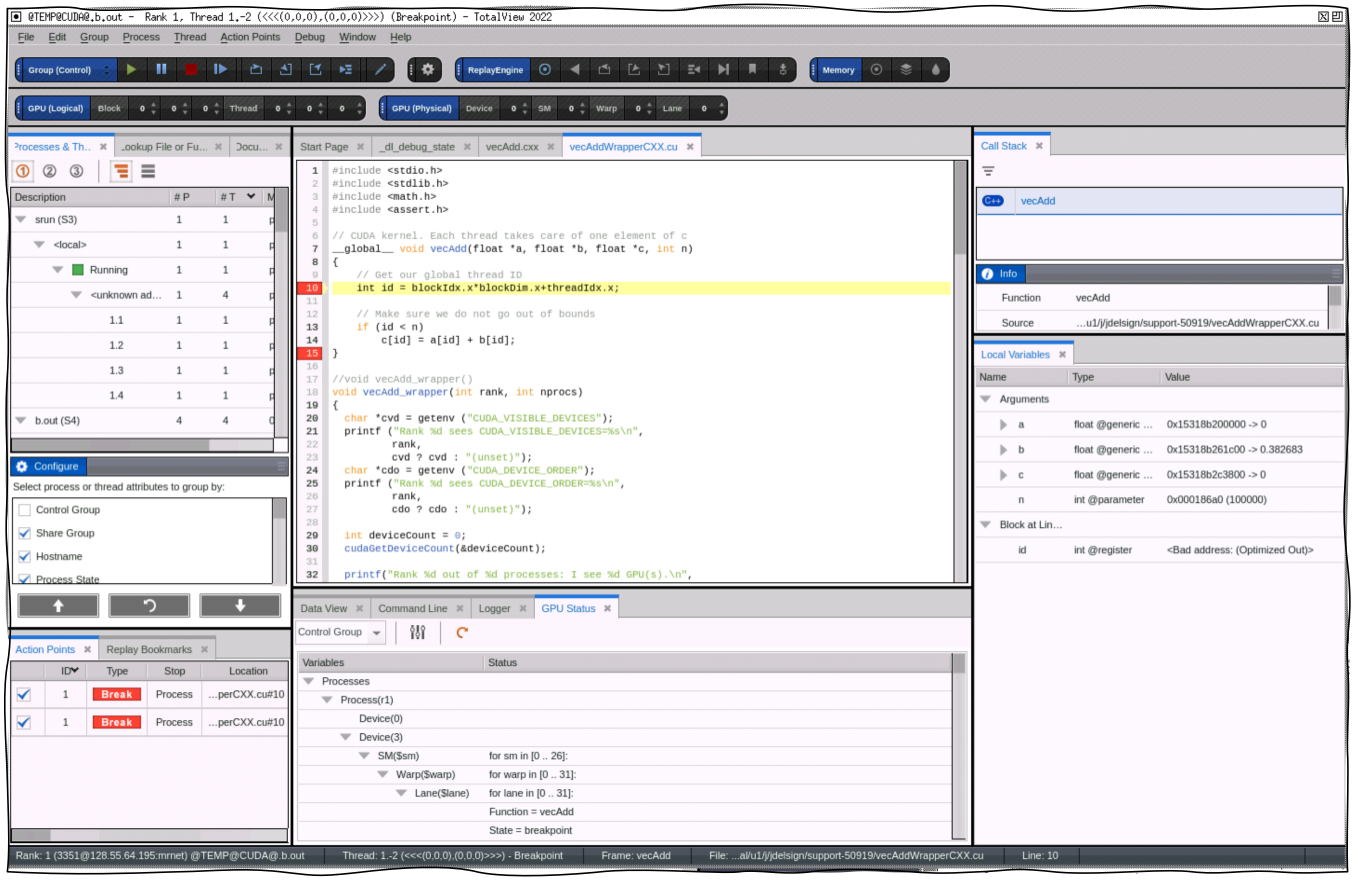
- Line numbers indicate if there's code at that line
  - Pale line numbers indicate no code (yet)
  - Bold line numbers indicate code
- CUDA code is *dynamically* loaded at runtime, so TotalView does not have any debug information *until* the CUDA kernel is launched
- Select a line number in the CUDA 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

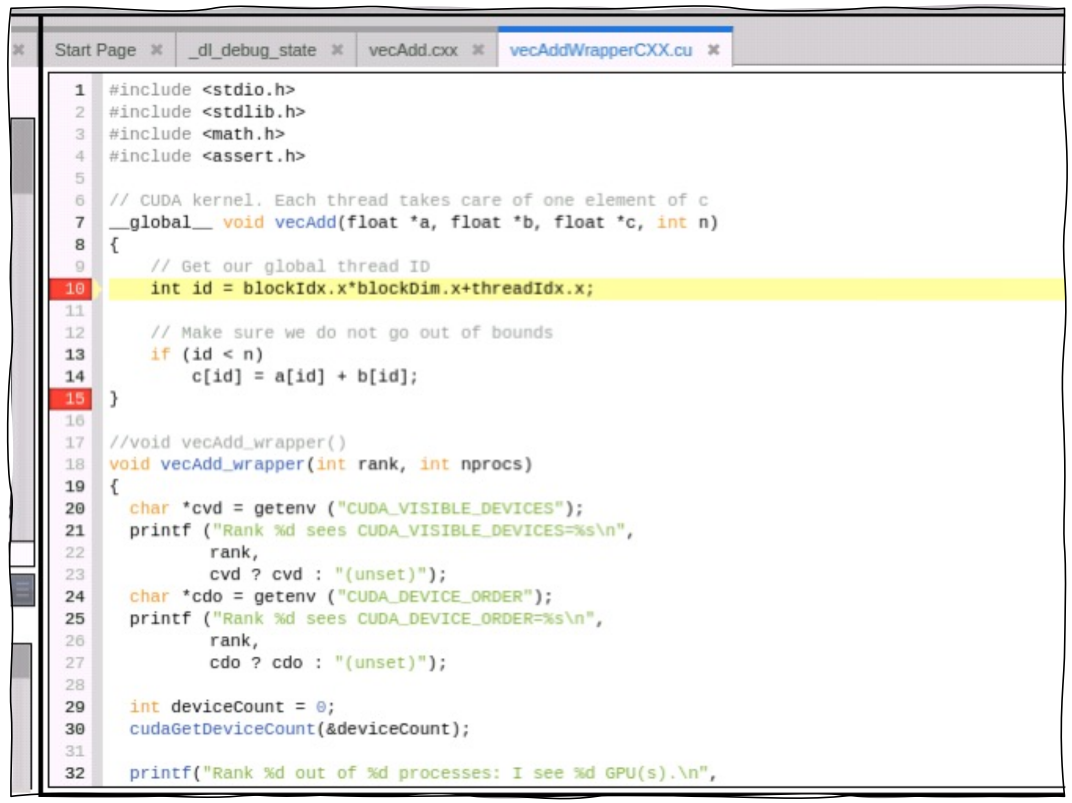


Stopped at a breakpoint in the CUDA kernel



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



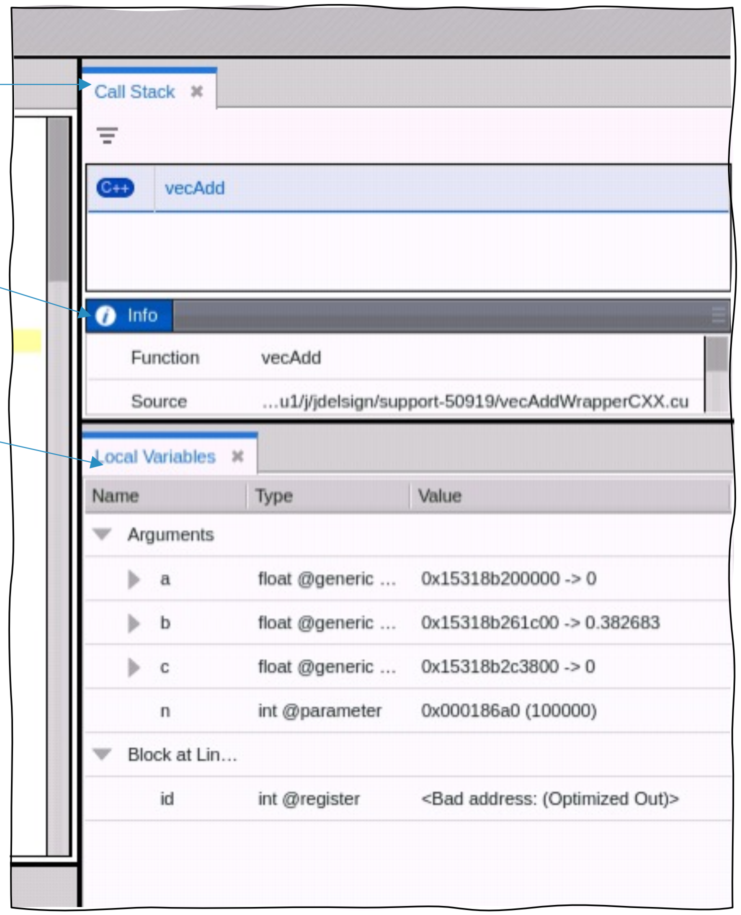
GPU thread focus and navigation controls

- “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

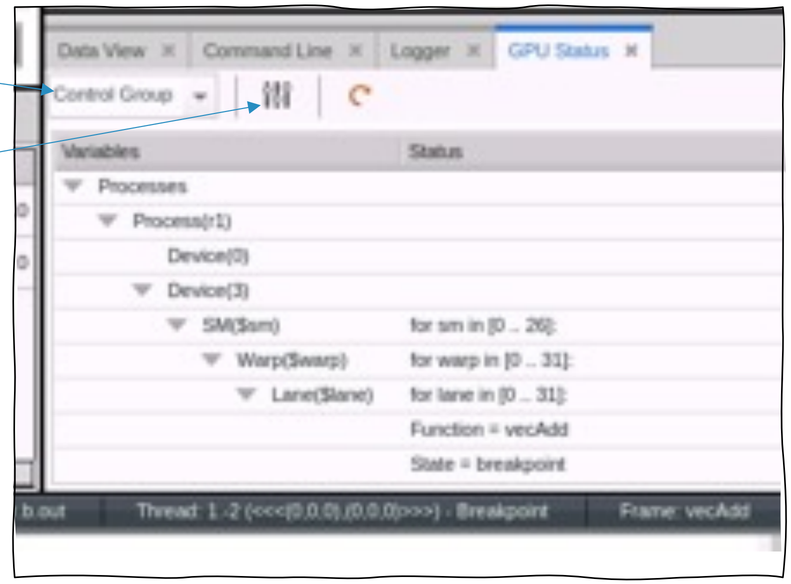
CUDA stack backtrace and local variables

- Call Stack
- Open the drawer for details
- Local Variables

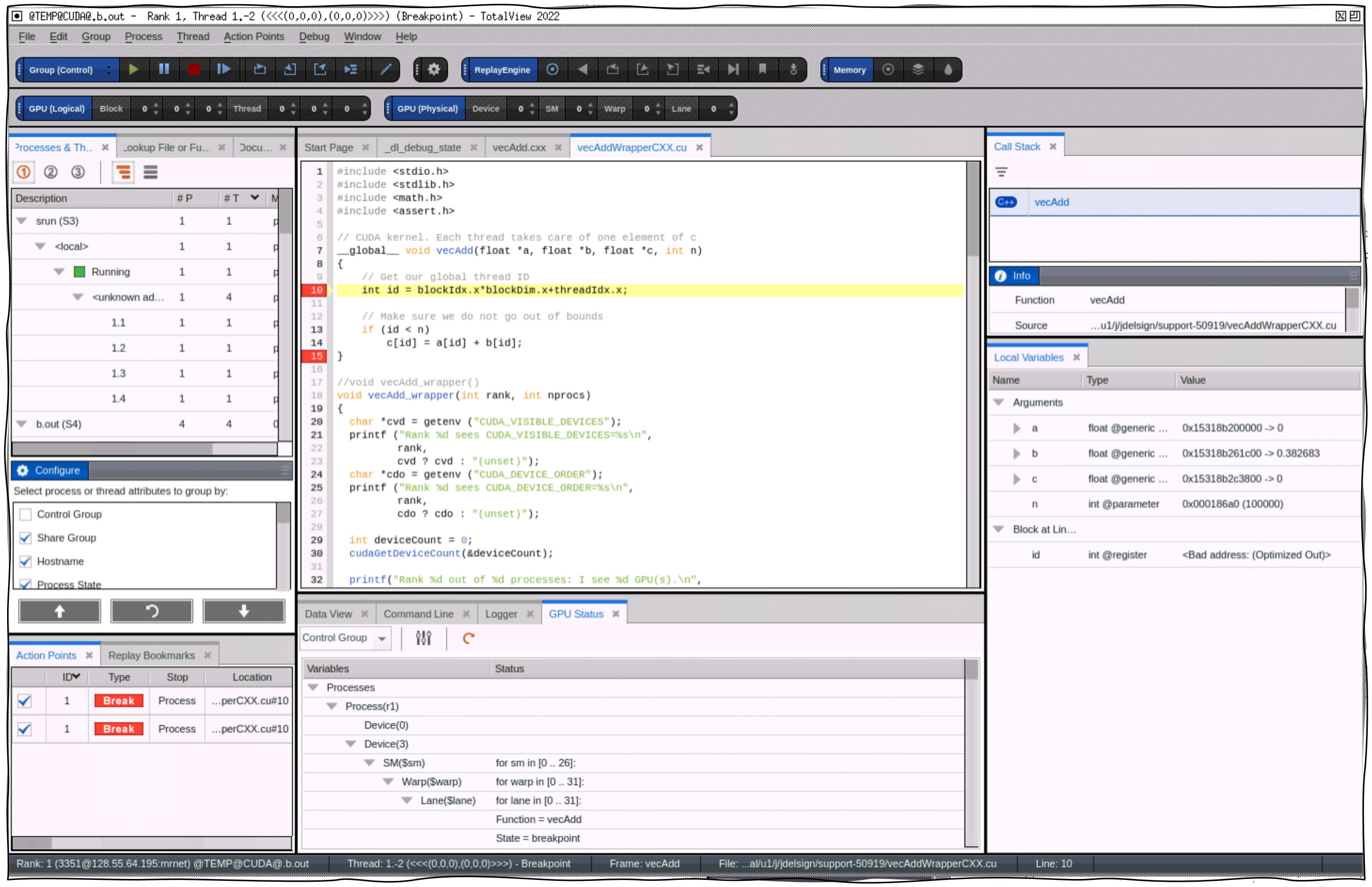


GPU Status view

- 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



Demo


@TEMP@CUDA@.b.out - Rank 1, Thread 1.-2 (<<<(0,0,0),(0,0,0)>>>) (Breakpoint) - TotalView 2022
File Edit Group Process Thread Action Points Debug Window Help
Group (Control)
ReplayEngine
Memory
GPU (Logical) Block Thread
GPU (Physical) Device SM Warp Lane
Start Page
_dl_debug_state
vecAdd.cxx
vecAddWrapperCXX.cu
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(float *a, float *b, float *c, int n)
// Get our global thread ID
int id = blockIdx.x*blockDim.x+threadIdx.x;
// Make sure we do not go out of bounds
if (id < n)
c[id] = a[id] + b[id];
//void vecAdd_wrapper()
void vecAdd_wrapper(int rank, int nprocs)
char *cvd = getenv ("CUDA_VISIBLE_DEVICES");
printf ("Rank %d sees CUDA_VISIBLE_DEVICES=%s\n",
char *cdo = getenv ("CUDA_DEVICE_ORDER");
printf ("Rank %d sees CUDA_DEVICE_ORDER=%s\n",
int deviceCount = 0;
cudaGetDeviceCount(&deviceCount);
printf("Rank %d out of %d processes: I see %d GPU(s).\n",
Call Stack
vecAdd
Info
Function vecAdd
Source ...u1/j/jdelsign/support-50919/vecAddWrapperCXX.cu
Local Variables
Name Type Value
Arguments
a float @generic ... 0x15318b200000 -> 0
b float @generic ... 0x15318b261c00 -> 0.382683
c float @generic ... 0x15318b2c3800 -> 0
n int @parameter 0x000186a0 (100000)
Block at Lin...
id int @register <Bad address: (Optimized Out)>
Processes & Th...
Lookup File or Fu...
Docu...
Description # P # T
srun (S3)
<local>
Running
<unknown ad...
b.out (S4)
Configure
Select process or thread attributes to group by:
Control Group
Share Group
Hostname
Process State
Data View
Command Line
Logger
GPU Status
Control Group
Variables Status
Processes
Process(r1)
Device(0)
Device(3)
SM($sm) for sm in [0 .. 26]:
Warp($warp) for warp in [0 .. 31]:
Lane($lane) for lane in [0 .. 31]:
Function = vecAdd
State = breakpoint
Action Points
Replay Bookmarks
ID Type Stop Location
1 Break Process ...perCXX.cu#10
1 Break Process ...perCXX.cu#10
Rank: 1 (3351@128.55.64.195:mrnet) @TEMP@CUDA@.b.out
Thread: 1.-2 (<<<(0,0,0),(0,0,0)>>>) - Breakpoint
Frame: vecAdd
File: ...al/u1/j/jdelsign/support-50919/vecAddWrapperCXX.cu
Line: 10

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

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

Supported tvscript events

Event Type	Event	Definition
	hoard_low_memory_threshold	Hoard low memory threshold crossed.
	realloc_not_allocated	Program attempted to reallocate an address that is not in the heap.
	rz_overrun	Program attempted to access memory beyond the end of an allocated block.
	rz_underrun	Program attempted to access memory before the start of an allocated block.
	rz_use_after_free	Program attempted to access a block of memory after it has been deallocated.
	rz_use_after_free_overrun	Program attempted to access memory beyond the end of a deallocated block.
	rz_use_after_free_underrun	Program attempted to access memory before the start of a deallocated block.
	termination_notification	The target is terminating.
Source code debugging event	**actionpoint**	A thread hit an action point.
	error	An error occurred.
Reverse debugging	**stopped_at_end**	The program is stopped at the end of execution and is about to exit.

Supported tvscript actions

Action Type	Action	Definition
Memory debugging actions	**check_guard_blocks**	Checks all guard blocks and write violations into the log file.
	list_allocations	Writes a list of all memory allocations into the log file.
	list_leaks	Writes a list of all memory leaks into the log file.
	save_html_heap_status_source_view	Generates and saves an HTML version of the Heap Status Source View Report.
	save_memory_debugging_file	Generates and saves a memory debugging file.
	save_text_heap_status_source_view	Generates and saves a text version of the Heap Status Source View Report.
Source code debugging actions	**display_backtrace** [**-level***level-num*] [*num_levels*] [*options*]	Writes the current stack backtrace into the log file. **-level** *level-num* sets the level at which information starts being logged. *num_levels* restricts output to this number of levels in the call stack. If you do not set a level, **tvscript** displays all levels in the call stack. *options* is one or more of the following: **-[no]show_arguments** **-[no]show_f**p **-[no]show_fp_registers** **-[no]show_image** **-[no]show_locals** **-[no]show_pc** **-[no]show_registers**

Supported tvscript actions

Action Type	Action	Definition
	print [**-slice** {*slice_exp*}] {*variable* | *exp*}	Writes the value of a variable or an expression into the log file. If the variable is an array, the **-slice** option limits the amount of data defined by slice_exp. A slice expression is a way to define the slice, such as **var[100:130]**in C and C++. (This displays all values from **var[100]**to **var[130]**.) To display every fourth value, add an additional argument; for example,**var[100:130:4]**. For additional information, see *"Examining Arrays"*in the *TotalView for HPC User Guide*.
Reverse debugging actions	**enable_reverse_debugging**	Turns on ReplayEngine reverse debugging and begins recording the execution of the program.
	save_replay_recording_file	Saves a ReplayEngine recording file. The filename is of the form `<ProcessName>-<PID>_<DATE>.<INDEX>.recording`.

tvscript examples

Simple example

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

MPI example

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

tvscript examples

Memory Debugging example

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

ReplayEngine example

```
tvscript \
-create_actionpoint "main=>enable_reverse_debugging" \
-event_action "stopped_at_end=>save_replay_recording_file" \
filterapp
```

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

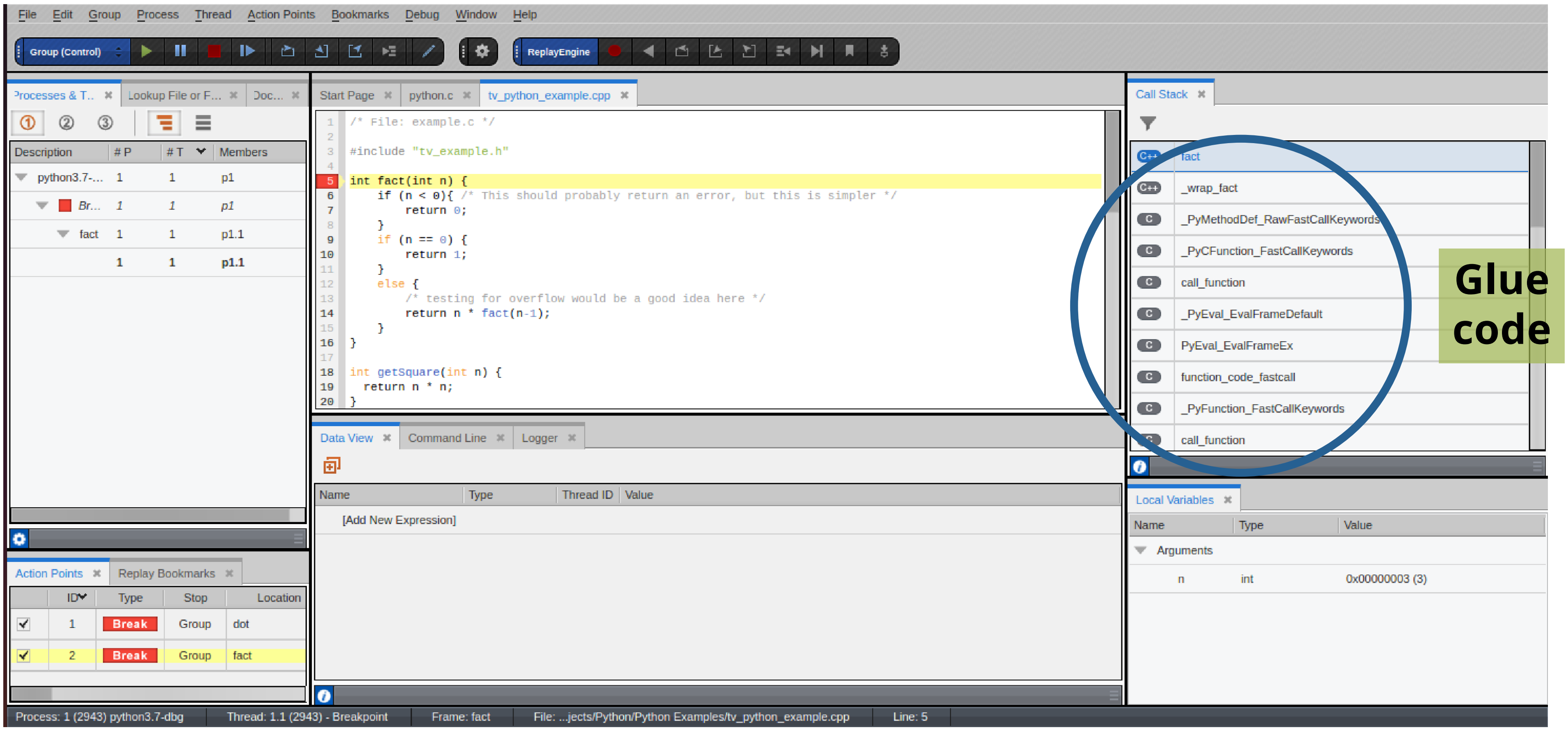
- What TotalView provides:
  - Easy python debugging session setup
  - Fully integrated Python and C/C++ call stack
  - ”Glue” layers between the languages removed
  - Easily examine and compare variables in Python and C++
  - Modest system requirements
  - Utilize reverse debugging and memory debugging
  - Support for Python 2.7 and Python 3.5 and above

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

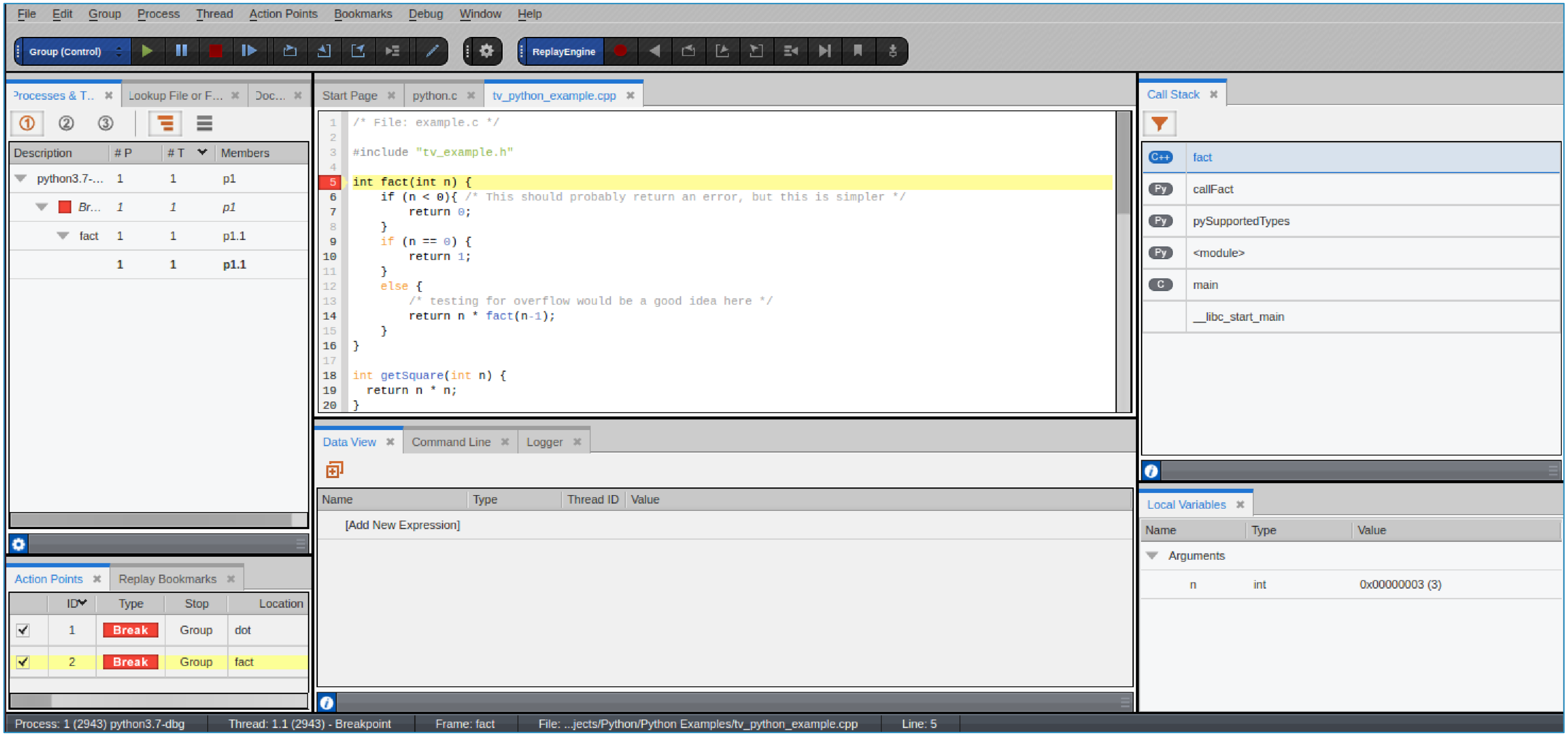
TotalView Power Tip

- 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



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

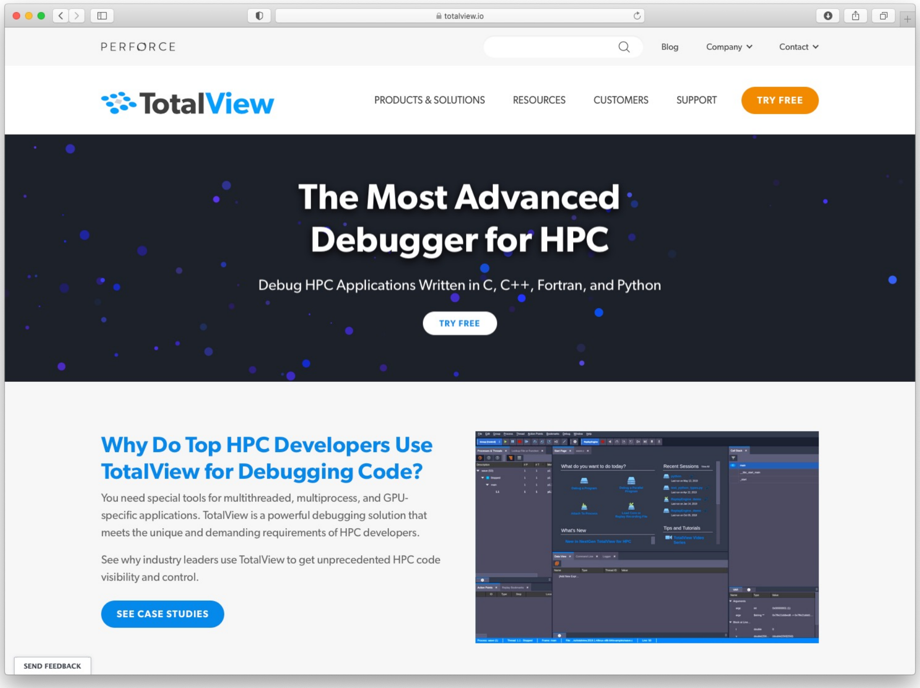
Common TotalView Usage Hints (cont.)

- 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
- 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

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