Techniques for Debugging HPC Applications

NIKOLAY PISKUN , TOTALVIEW   SOFTWARE ARCHITECT

AUGUST 11, 2021, ATPESC 2021

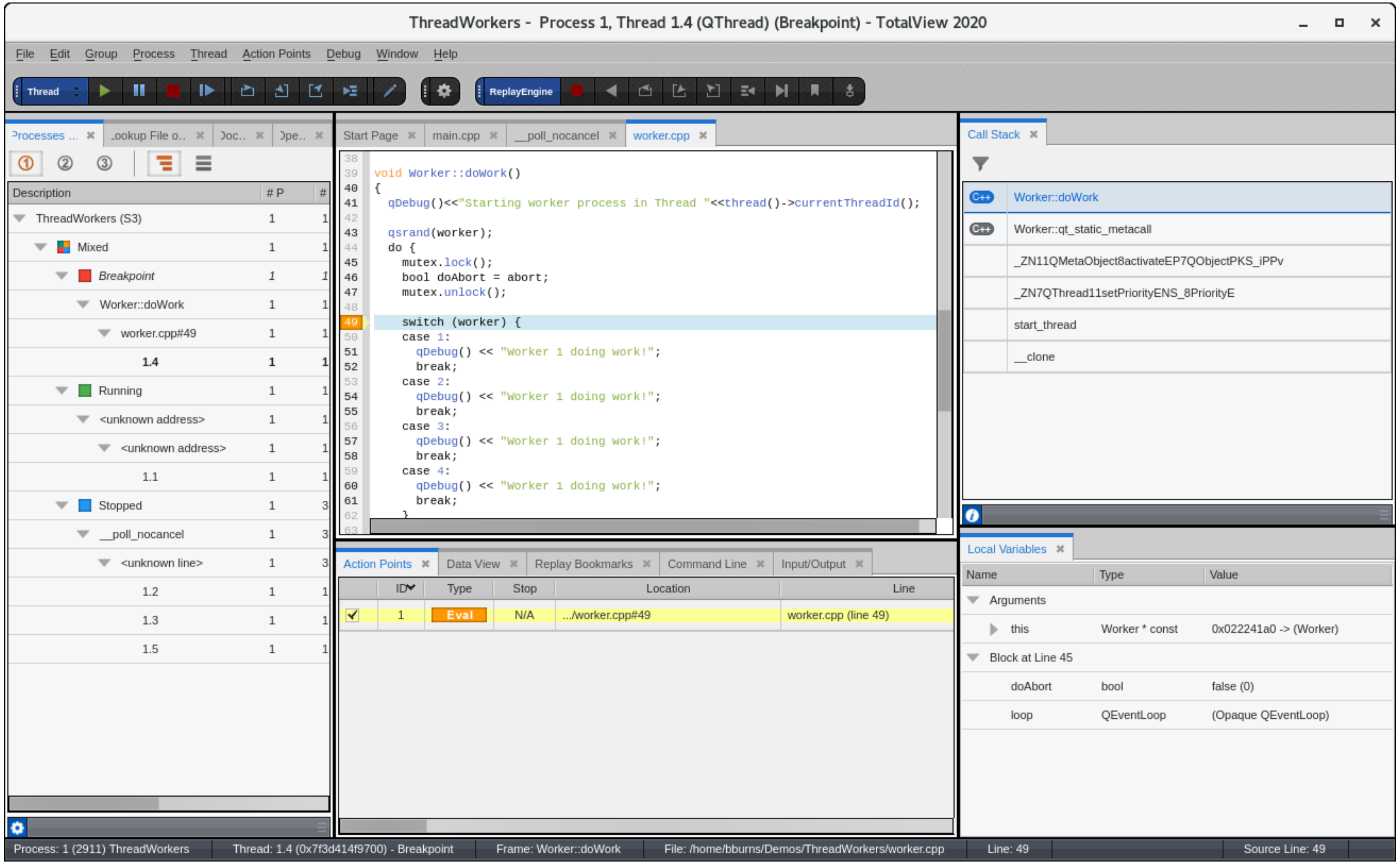
Agenda

- Introduction
- Overview of TotalView Features
- TotalView Debugging Solution
- General Debugging Features for C, C++, and Fortran
  - UI Navigation and Process Control
  - Action Points
  - Examining and Editing Data
  - Advanced C++ and Data Debugging
- Mixed Language C/C++/Fortran and Python Debugging
- Remote Debugging
- MPI, OpenMP, CUDA GPU, and Hybrid Debugging
- Reverse Connect/Attach
- Memory Debugging
- Reverse Debugging
- HPC Debugging Techniques
- TotalView Resources and Documentation
- Q&A

What is Debugging and Why do you need TotalView?

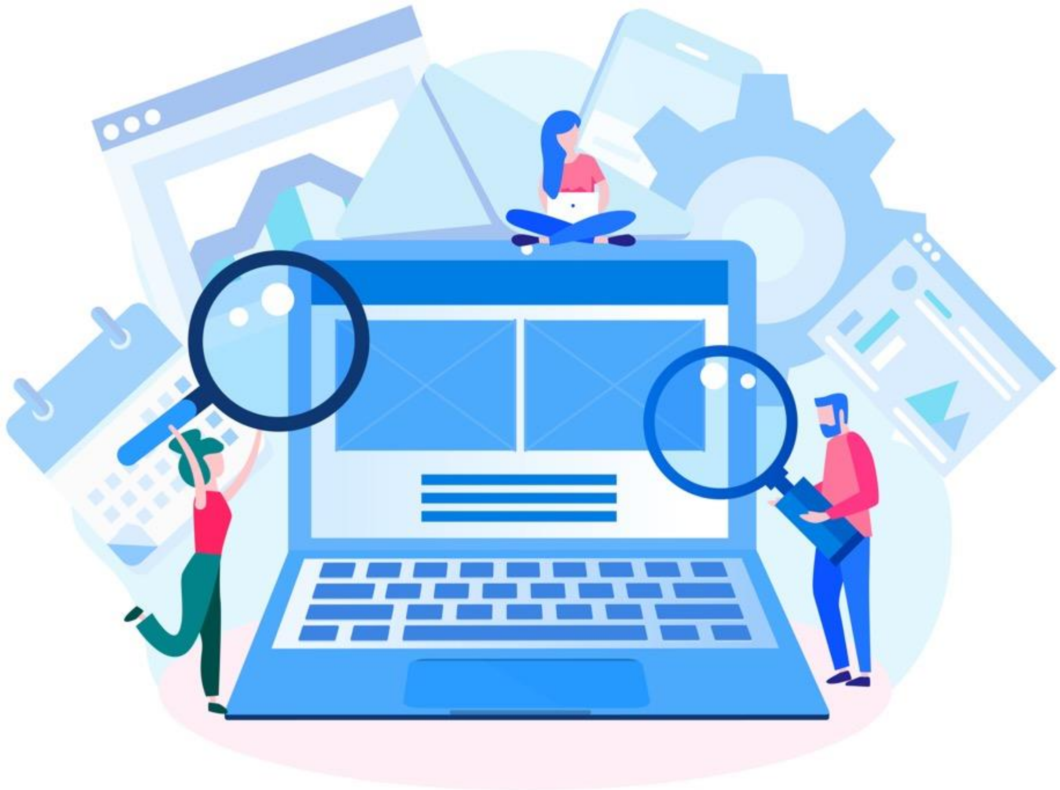
TotalView Features

- Comprehensive C, C++ and Fortran debugger
- Multi-process/multi-thread dynamic analysis
  - Thread specific breakpoints with individual thread control
  - View thread specific stack and data
  - View complex data types easily
- MPI, OpenMP, Hybrid and CUDA debugging
- Convenient remote debugging for HPC
- Integrated Reverse and Memory debugging
- Mixed Language - Python C/C++ debugging
- Script debugging
- Linux, macOS and UNIX



What is TotalView used for?

- **More than just a tool to find bugs**
  - 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
  - Applications in an automated test and batch environments



UI Navigation and Process Control

TotalView's Default Views

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

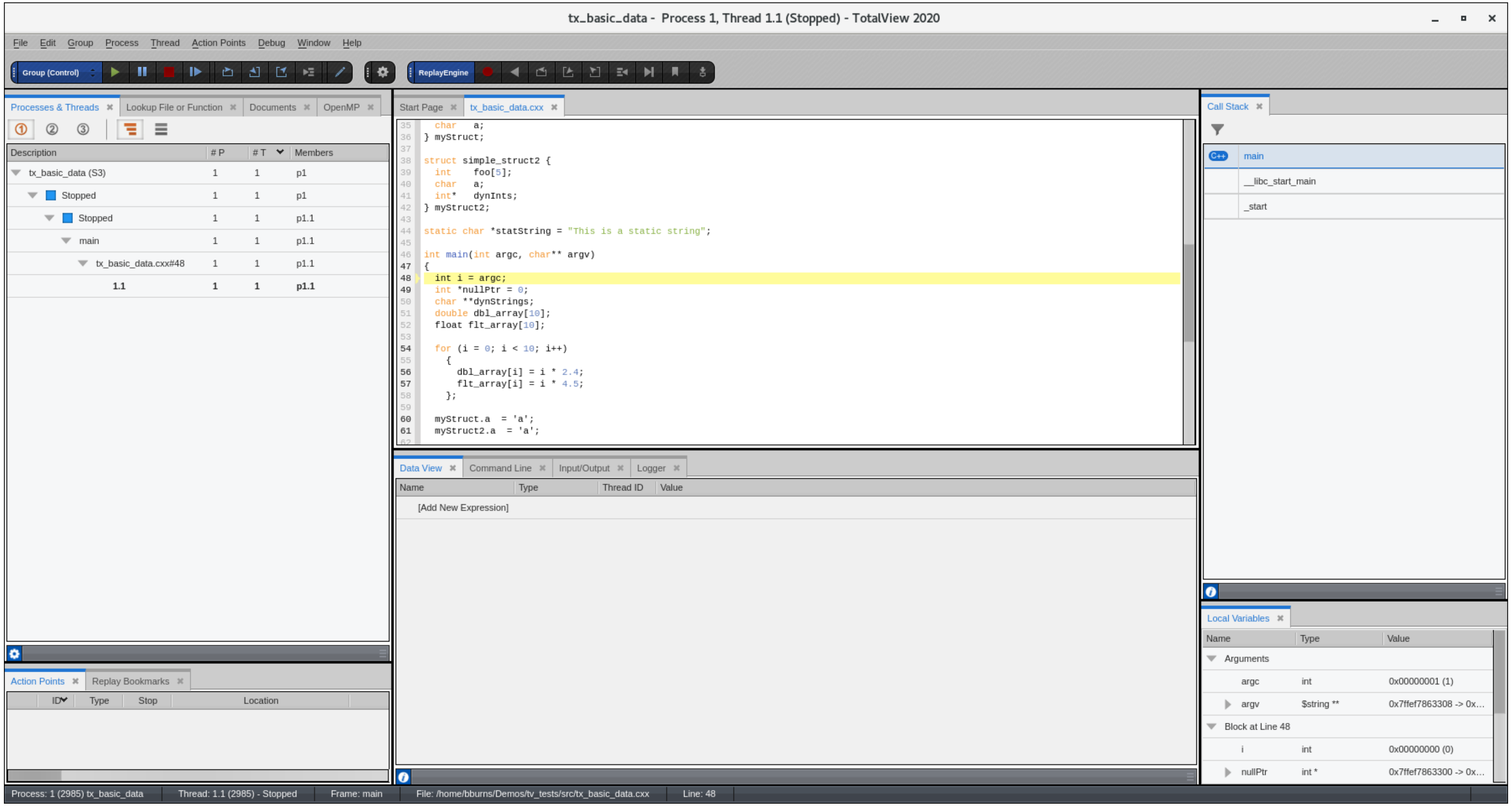
2. Source View

3. Call Stack View

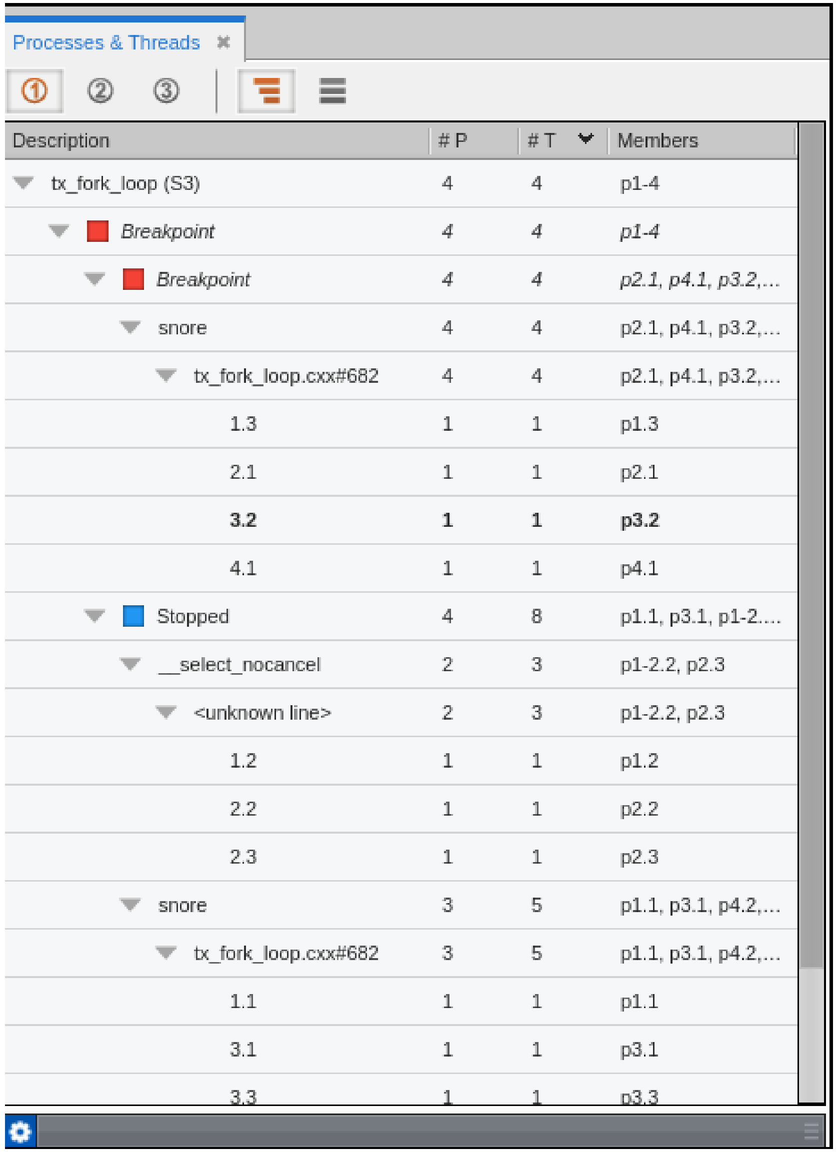
4. Local Variables View

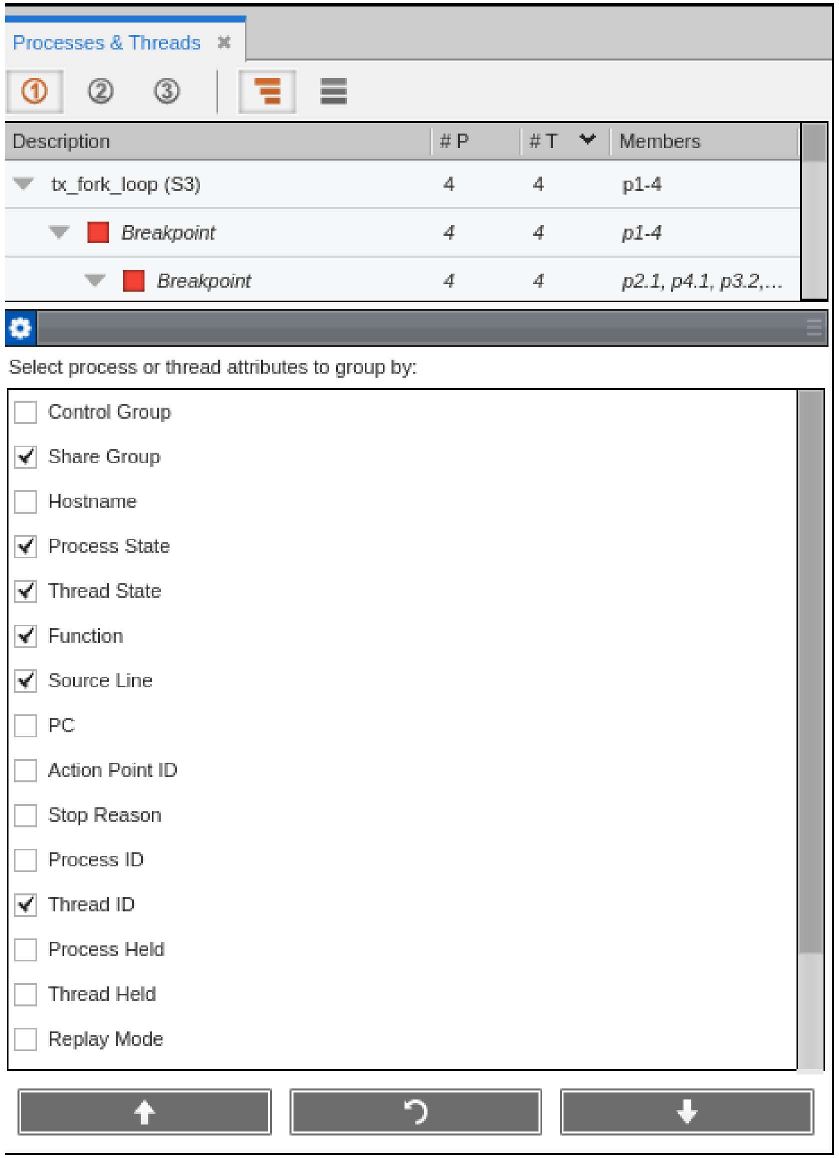
5. Data View, Command Line, Input/Output

6. Action Points, Replay Bookmarks



Process and Threads View

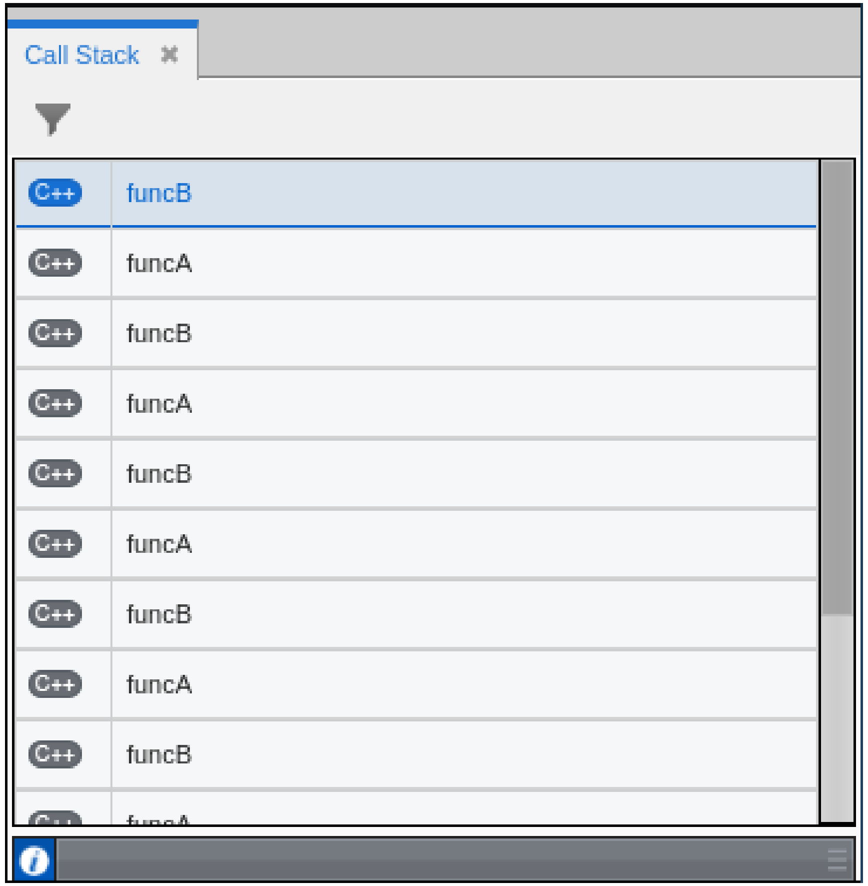


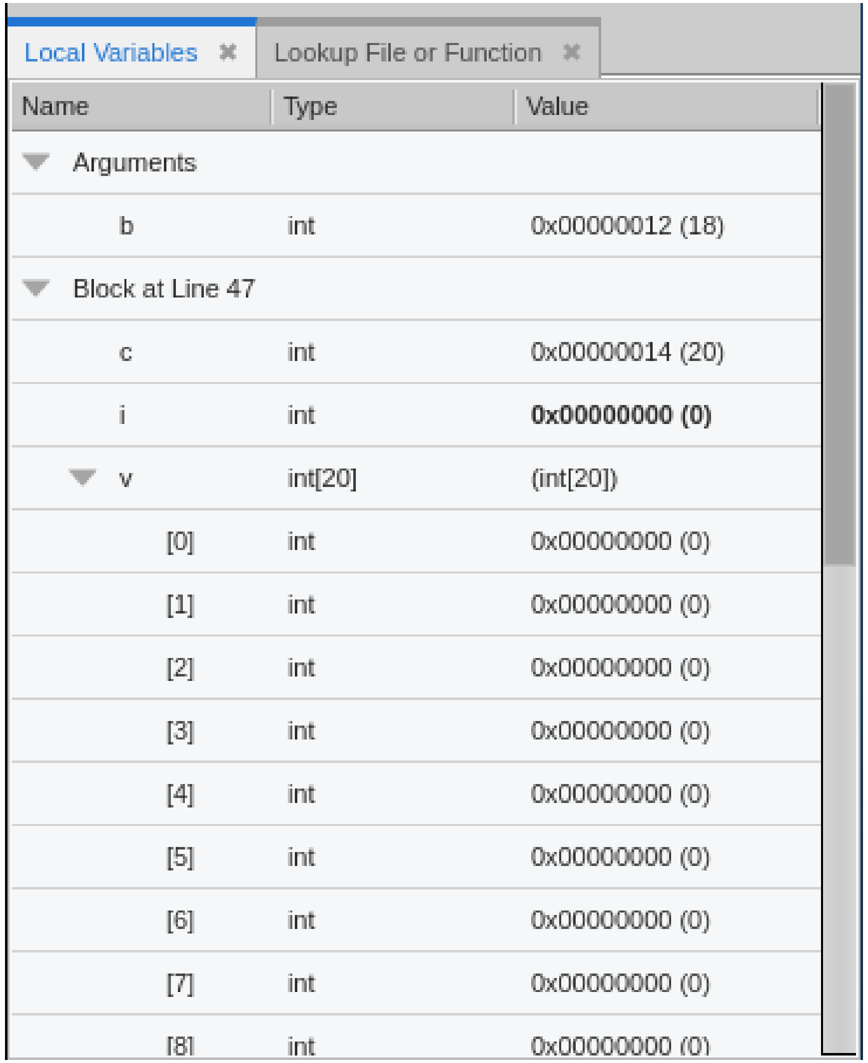


Source View

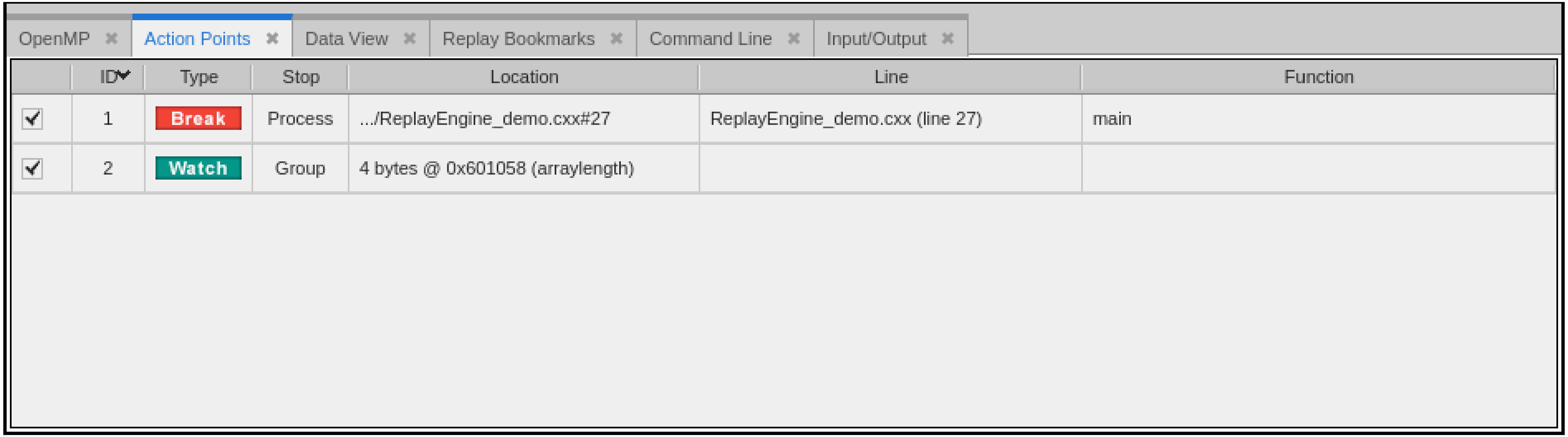

Start Page
tx_fork_loop.cxx
#else
whoops = pthread_create (&new_tid, &attr, (void*(*)(void*))forker, (void *)local_fork_count);
else
whoops = pthread_create (&new_tid, NULL, (void*(*)(void*))forker, (void *)local_fork_count);
#endif
if (whoops)
{
printf("pthread_create failed; result=%d, errno=%d\n", 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

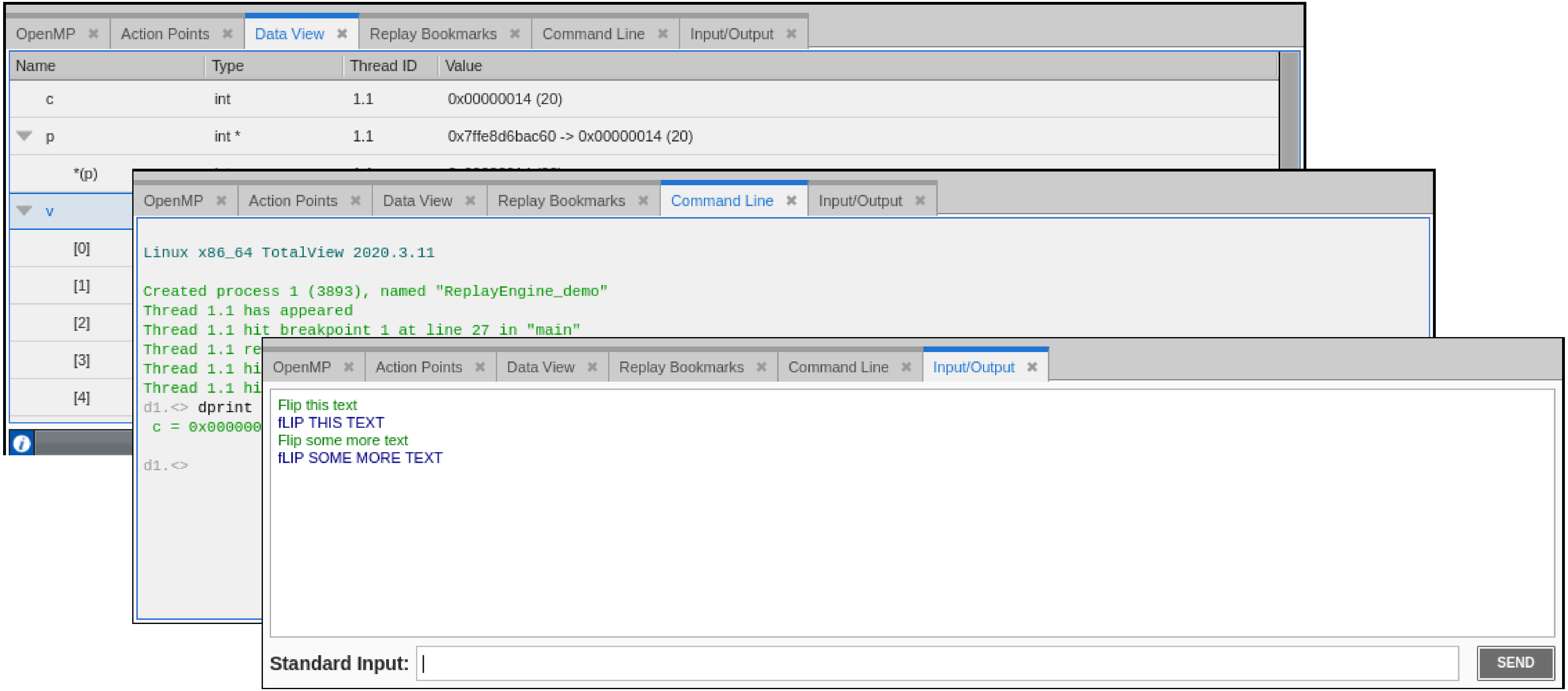




Action Points View

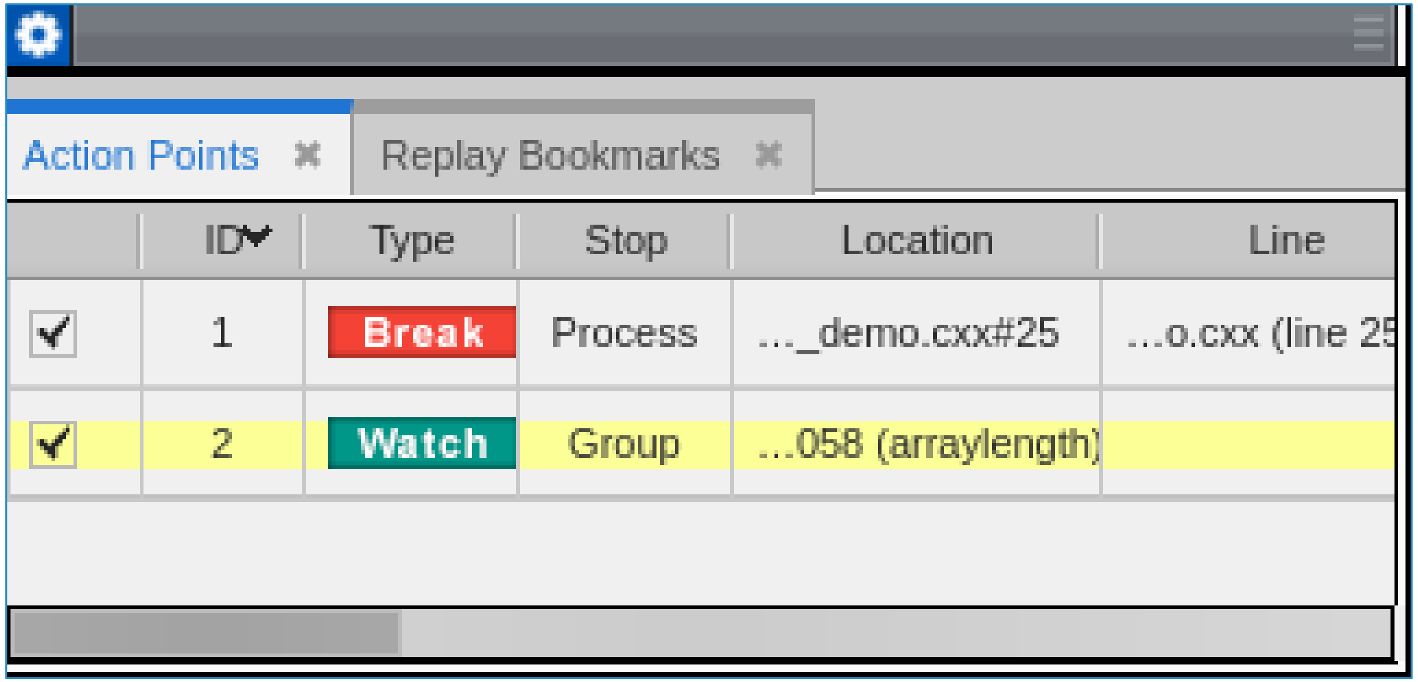


Data View, Command Line View and Input/Output View



Action Points

Action Points



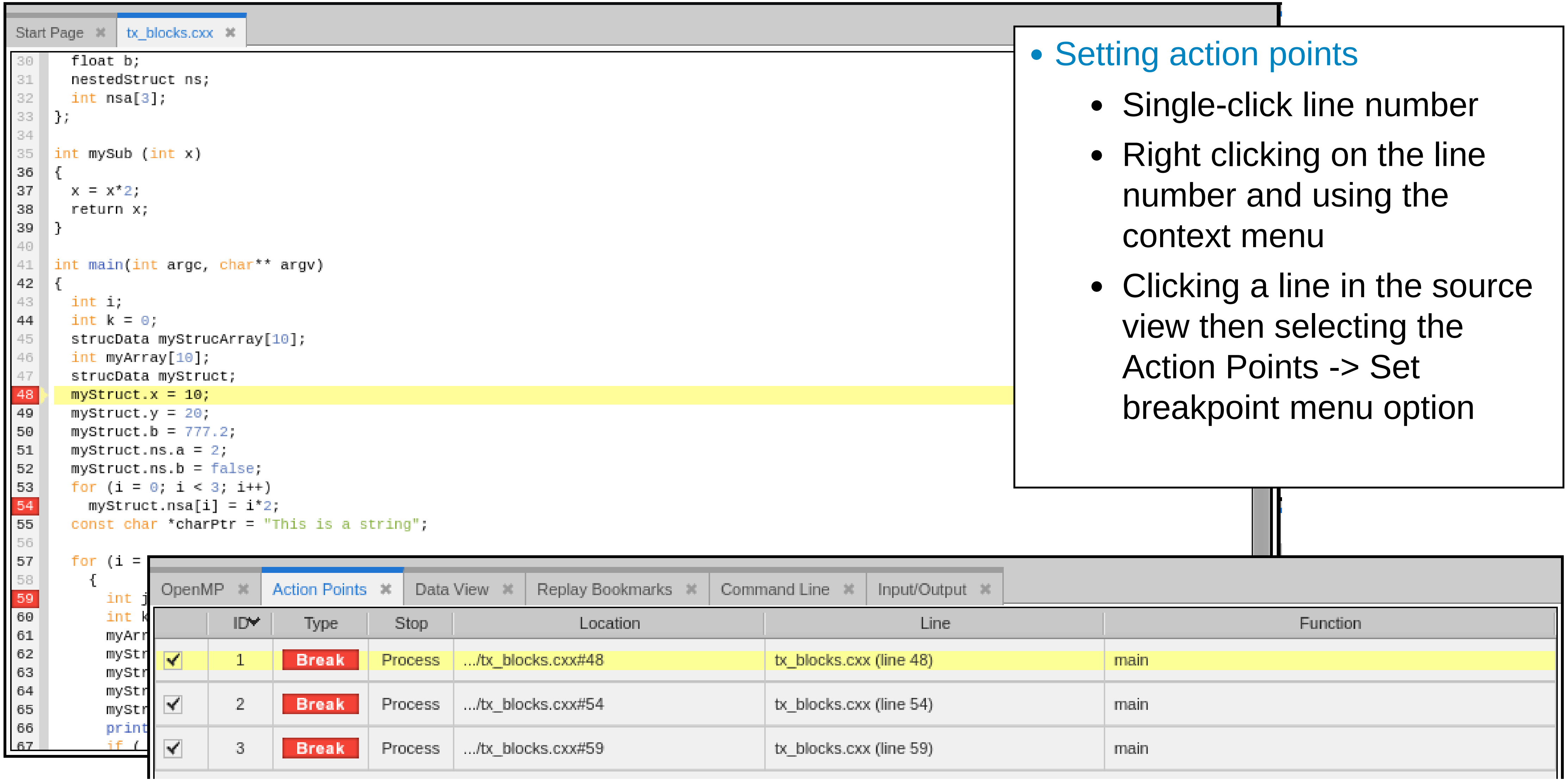
Breakpoint

Evaluation Point (Evalpoint)

Watchpoint

Barrier point

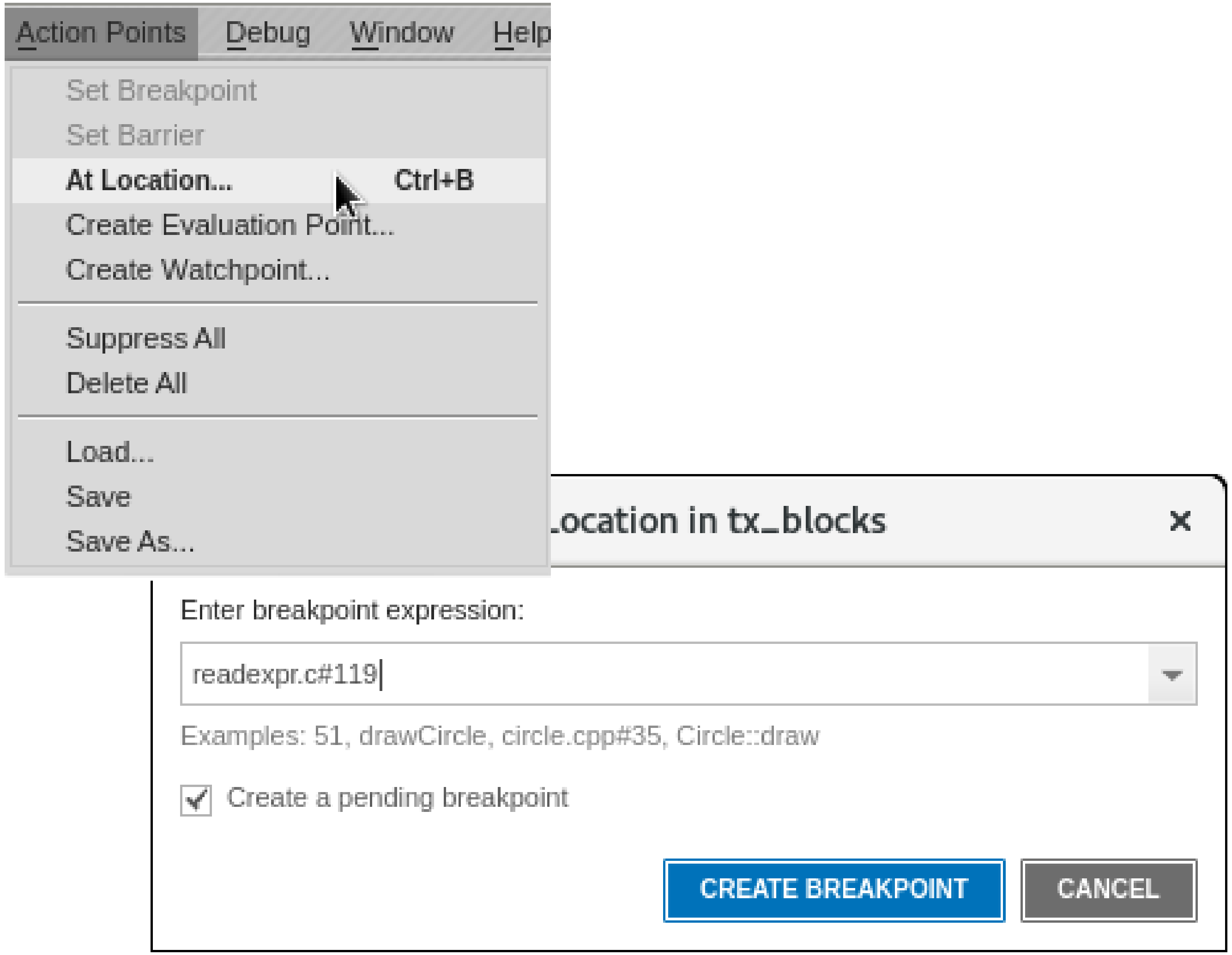
Setting Breakpoints



- 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

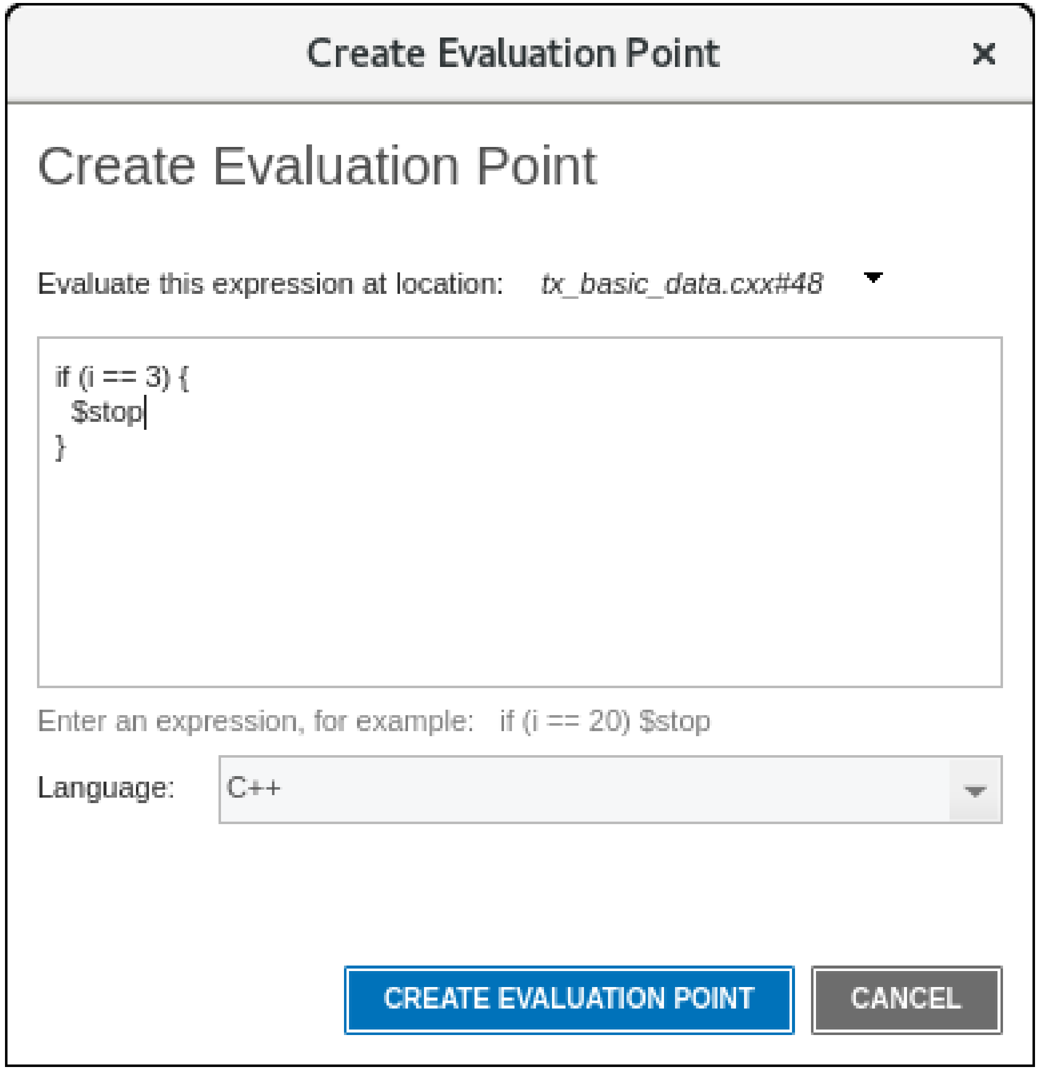
Setting Breakpoints

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



Evaluation points

- 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



Evaluation points 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;
```

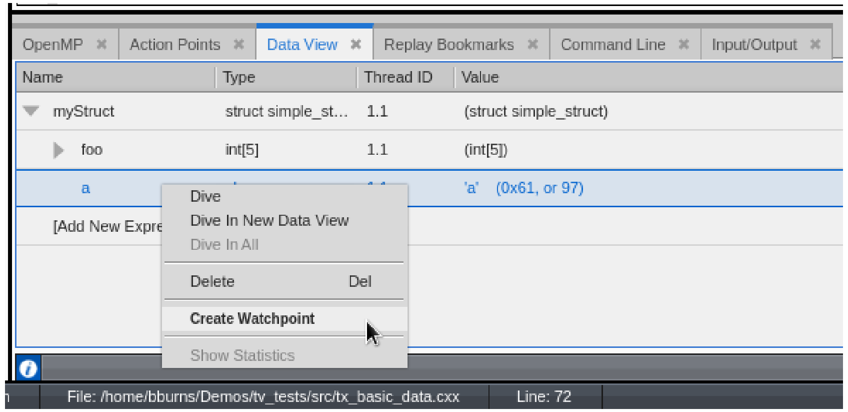
- Stop a loop after a certain number of iterations

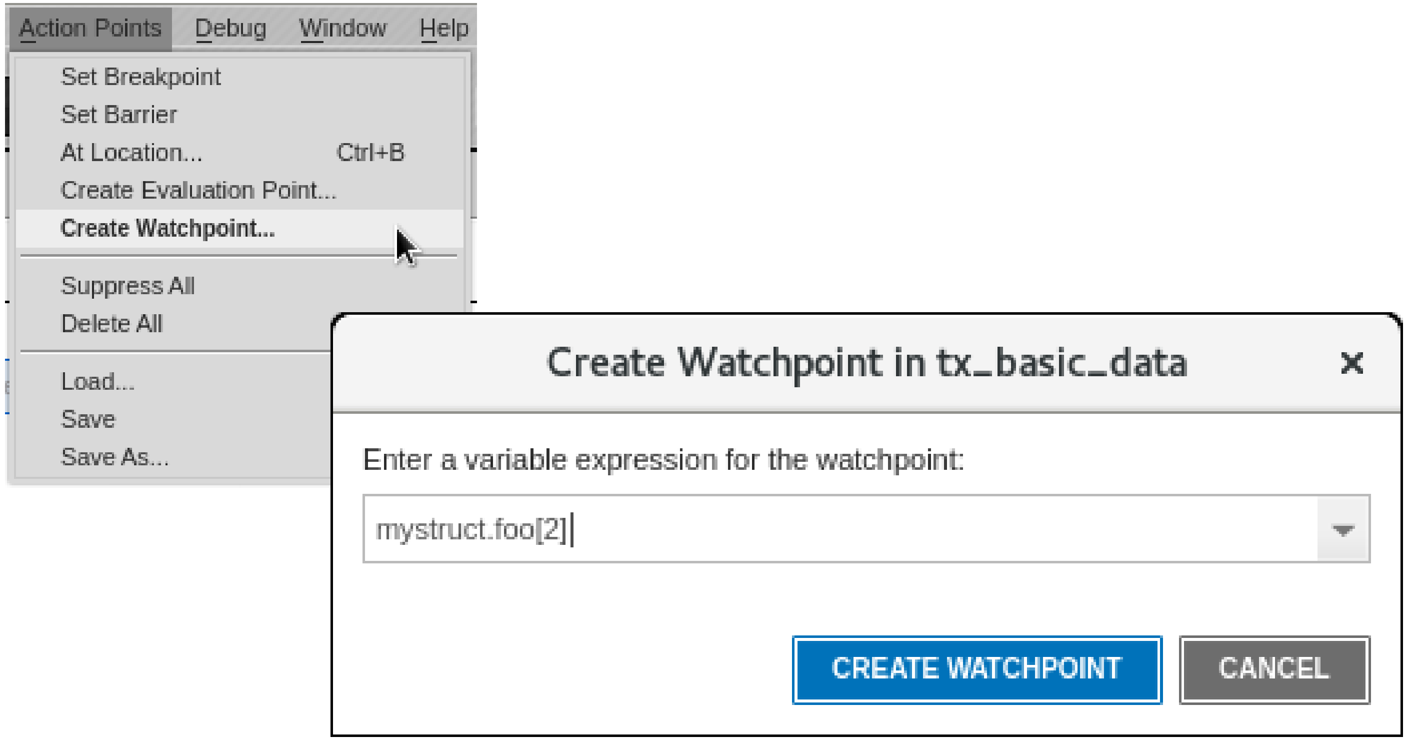
```
if ( (i % 100) == 0) {
   printf("The value of i is %d\n", i);
   $stop;
}
```

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

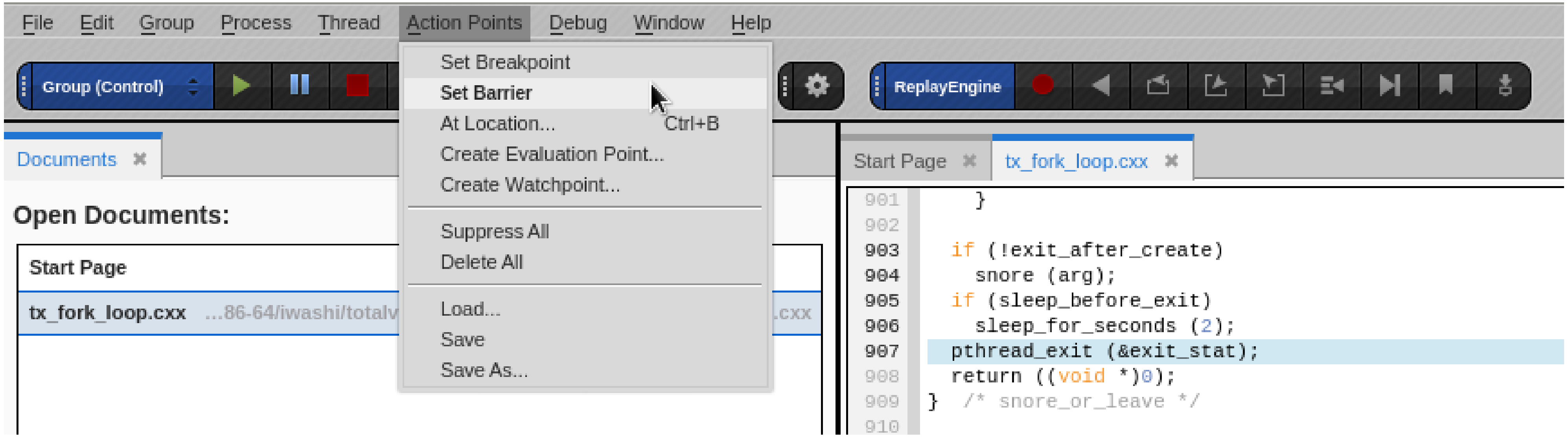
- Watchpoints are set on a specific memory location
- Execution is stopped when the value stored in that memory location changes
- A breakpoint stops ***before*** an instruction executes. A watchpoint stops ***after*** an instruction executes



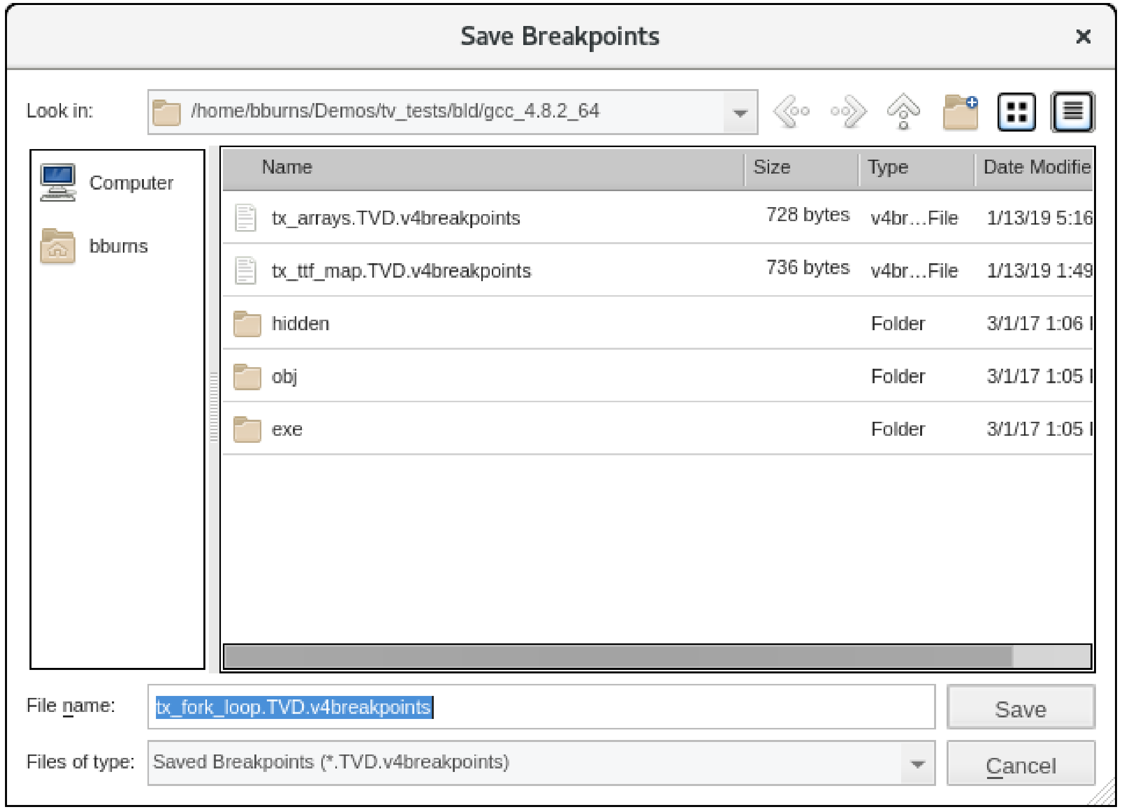


Barrier Breakpoints

- 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

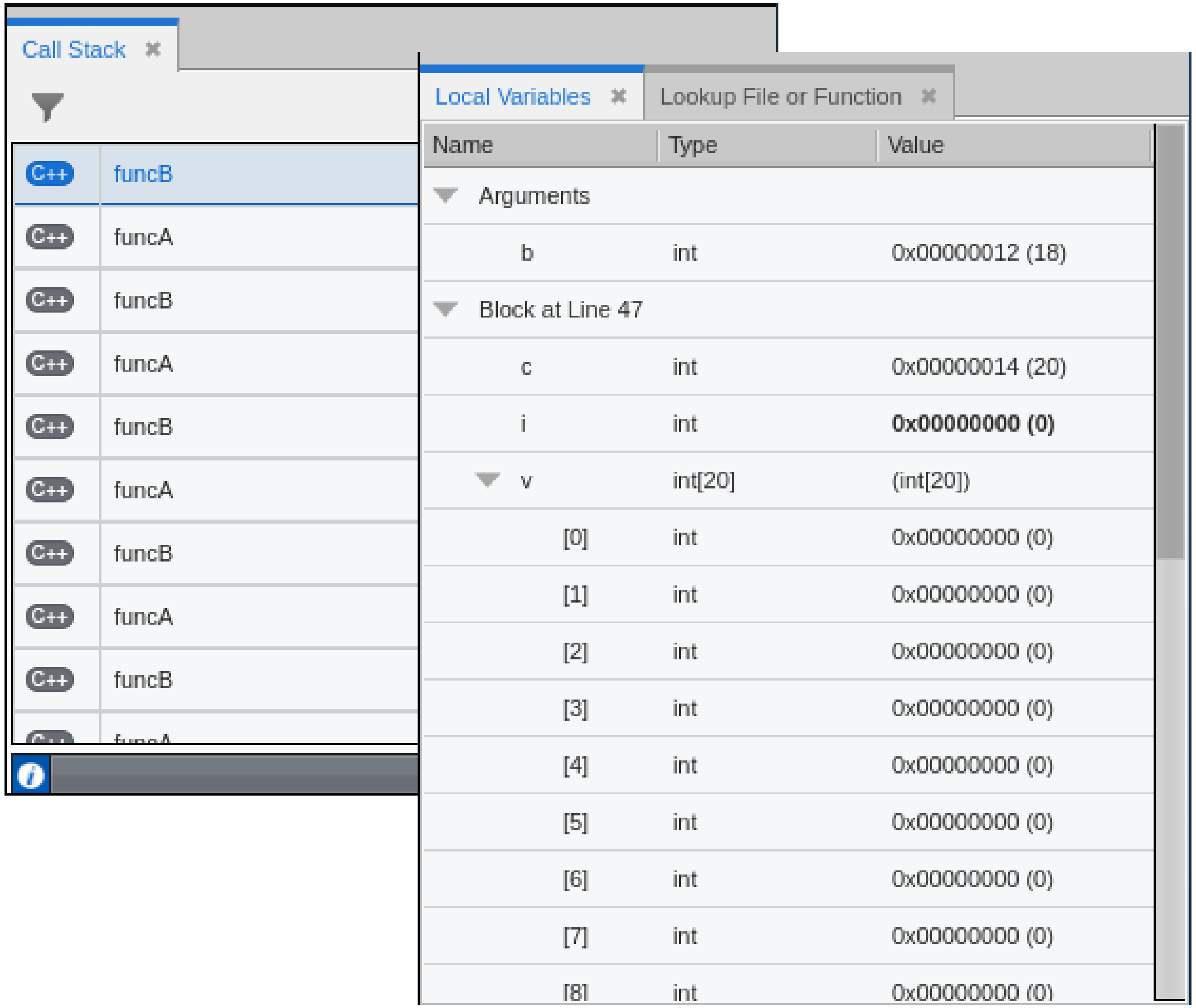


From the Action Points menu select Save or Save As to save breakpoints

Turn on option to save action points on exit

Examining and Editing Data

Call Stack and Local Variables



Call Stack View

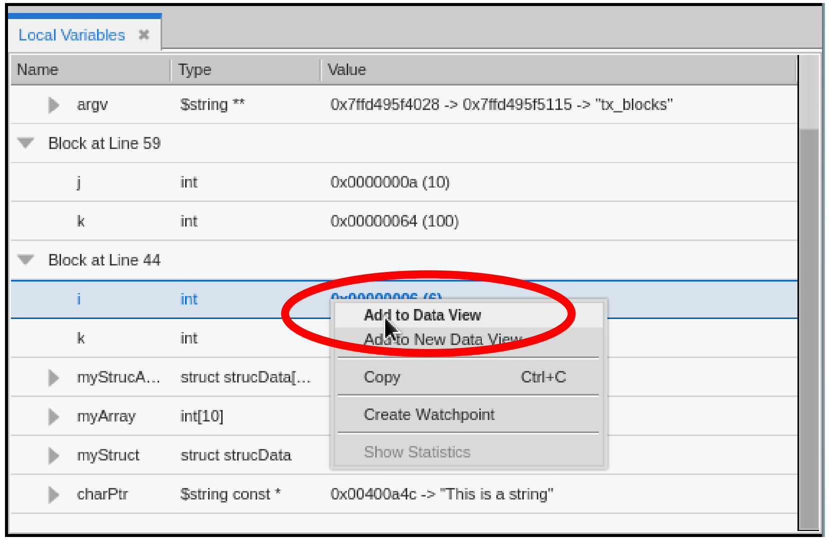
- Lists the set of call frames as the program calls from one function or method to another
- Filter button used to turn on or off filtering of frames.

Local Variables View

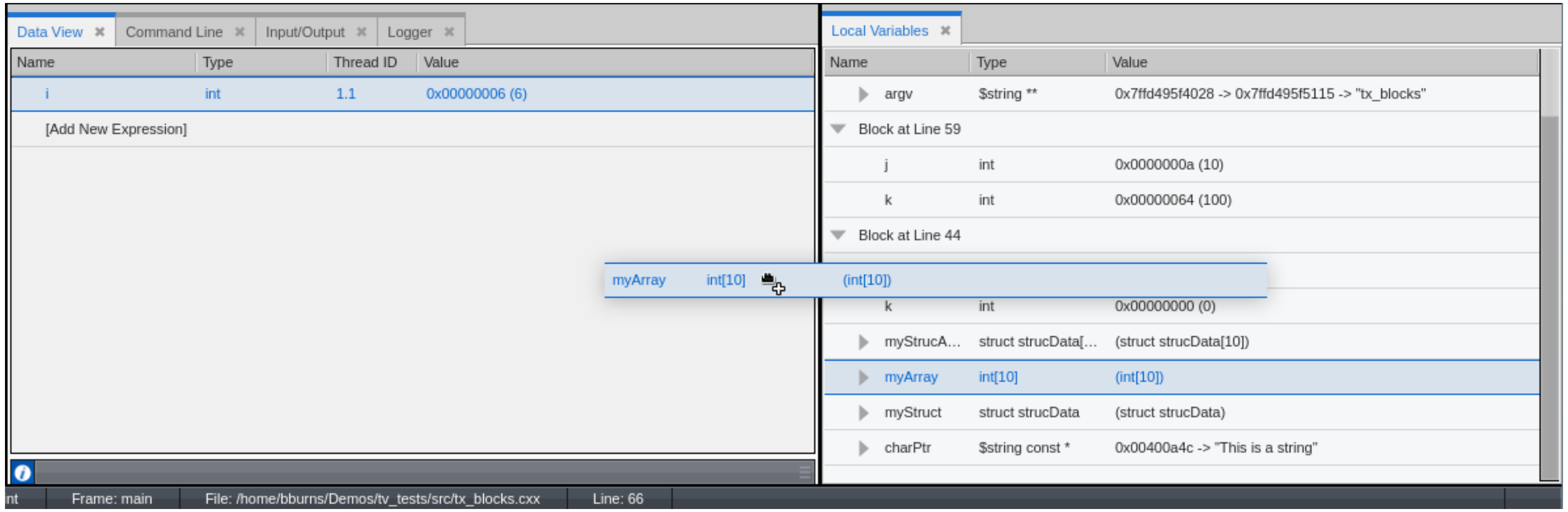
- Displays local variables relative to the current thread of interest and the selected stack frame
- Organized by arguments and blocks
- To edit values, add variable to the Data View

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

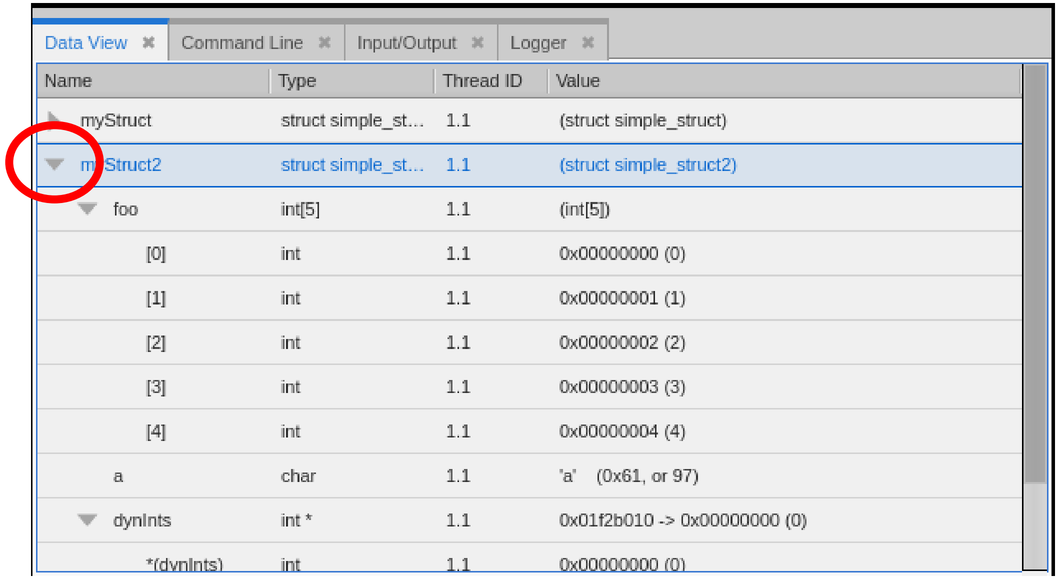
The Data View Panel – Expanding Arrays and Structures

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

Any nested structures are displayed in the data view


Data View | Command Line | Input/Output | Logger

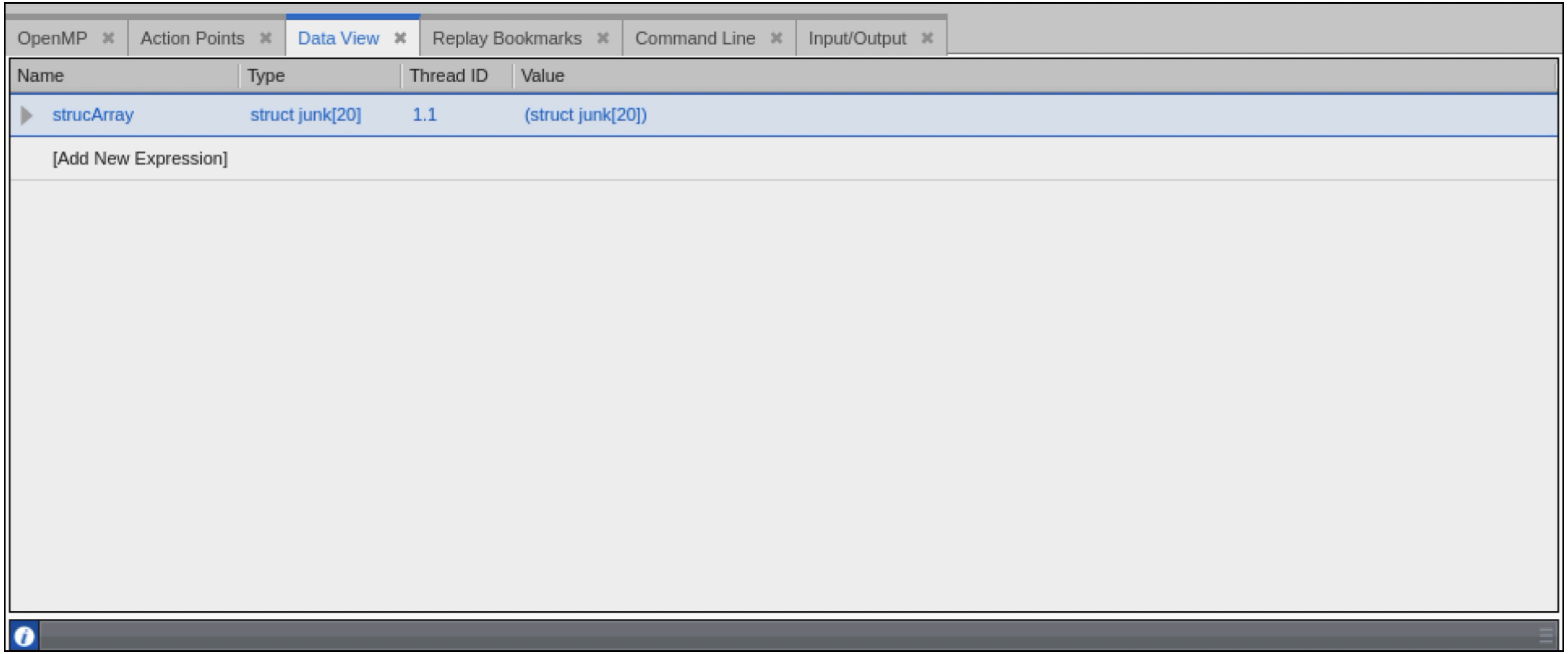
Name	Type	Thread ID	Value
flt_array	float[10]	1.1	(float[10])
[0]	float	1.1	0
[1]	float	1.1	4.5
[2]	float	1.1	9
[3]	float	1.1	13.5
[4]	float	1.1	18
[5]	float	1.1	22.5
[6]	float	1.1	27
[7]	float	1.1	31.5
[8]	float	1.1	36
[9]	float	1.1	40.5


Data View | Command Line | Input/Output | Logger

Name	Type	Thread ID	Value
myStruct	struct simple_st…	1.1	(struct simple_struct)
myStruct2	struct simple_st…	1.1	(struct simple_struct2)
foo	int[5]	1.1	(int[5])
[0]	int	1.1	0x00000000 (0)
[1]	int	1.1	0x00000001 (1)
[2]	int	1.1	0x00000002 (2)
[3]	int	1.1	0x00000003 (3)
[4]	int	1.1	0x00000004 (4)
a	char	1.1	'a' (0x61, or 97)
dynInts	int *	1.1	0x01f2b010 -> 0x00000000 (0)
*(dynInts)	int	1.1	0x00000000 (0)

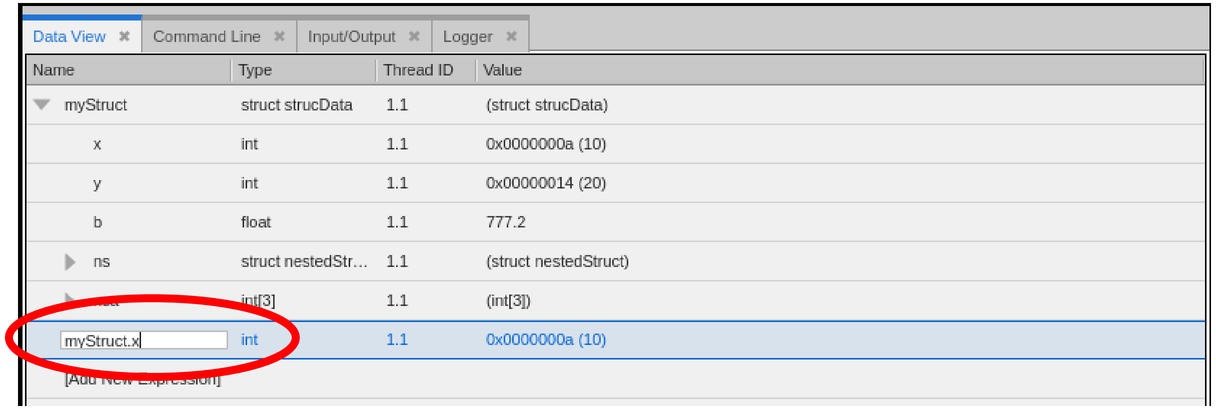
The Data View – Dive in All

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

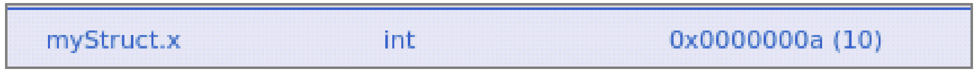


The Data View Panel – Entering Expressions

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



Type the expression in the [Add New expression] field

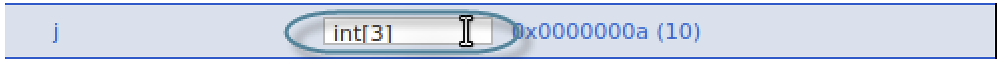


A new expression is added

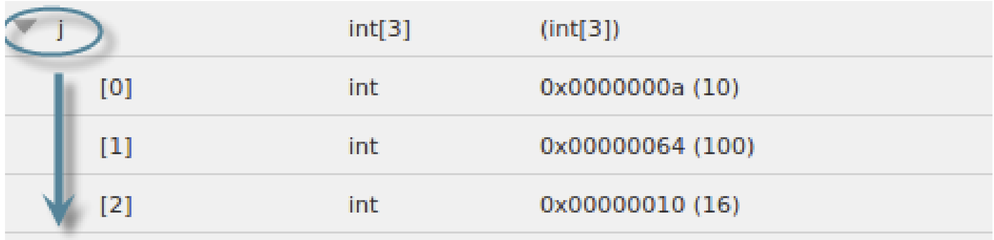
Increment a variable

The Data View Panel - Casting

Casting to another type



Cast a variable into an array by adding the array specifier



TotalView displays the array

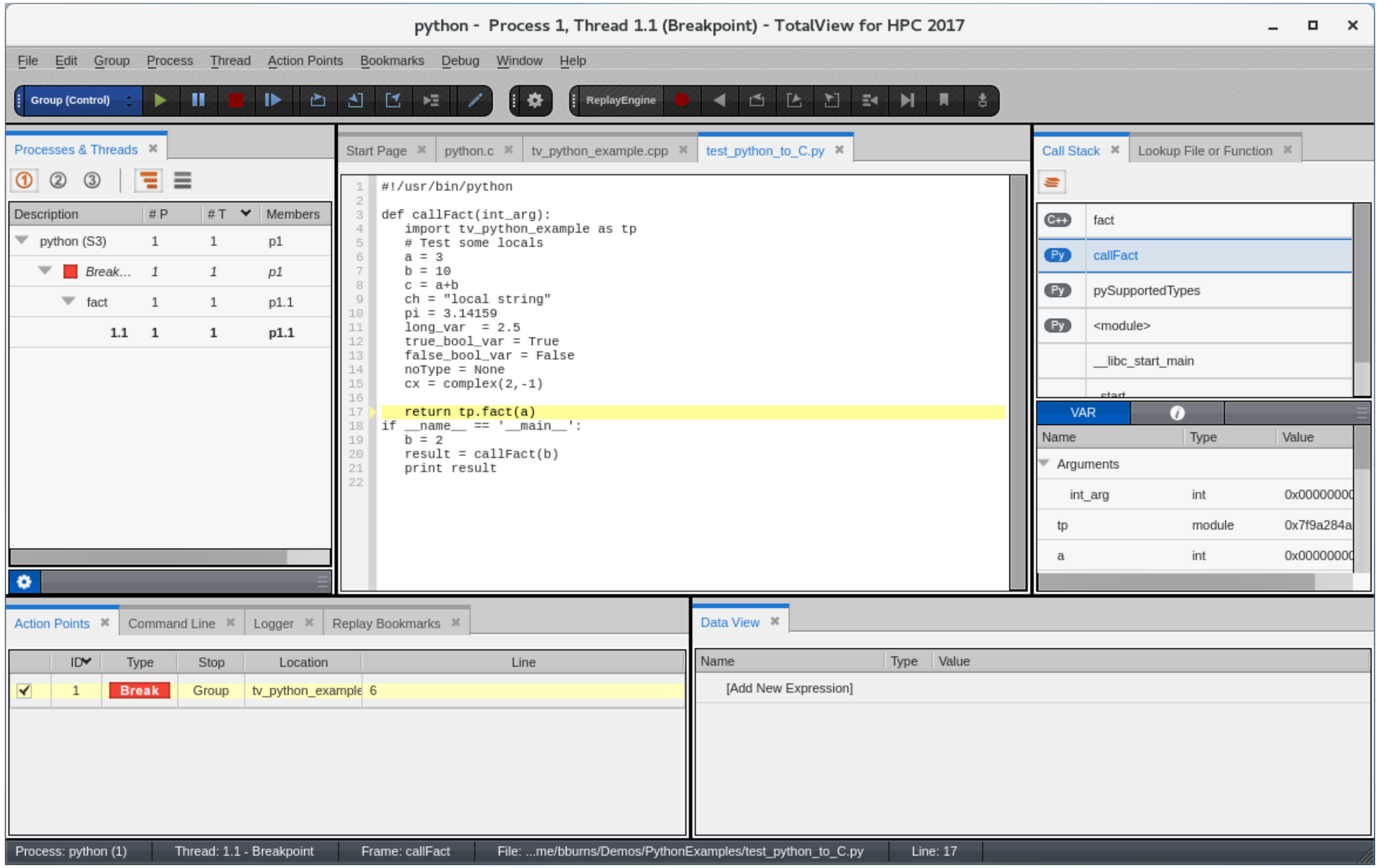
Extending Debugging Capabilities:
How to Debug (AI) Mixed Python/C++ Code

Mixed Language Python Debugging

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

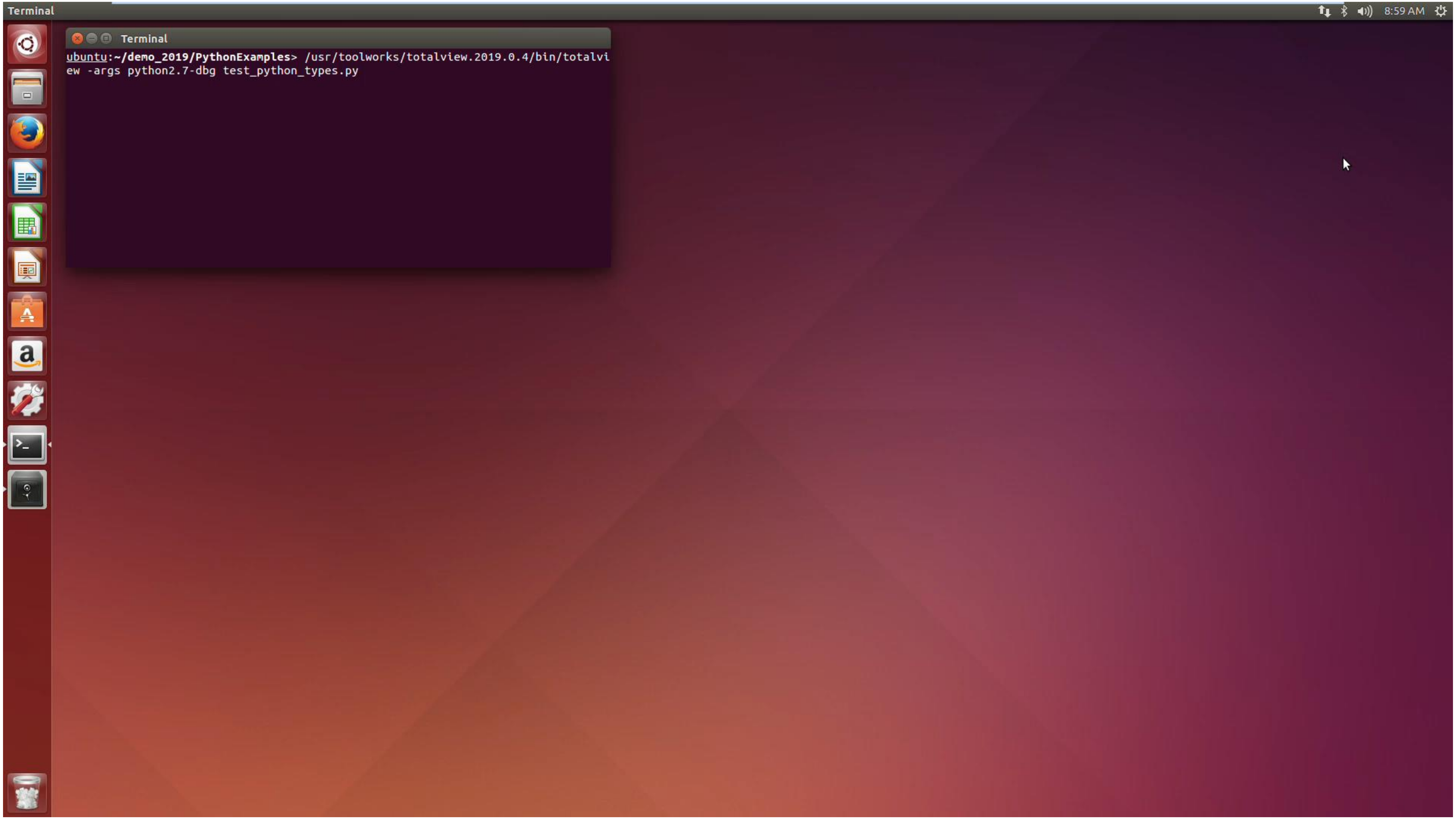
- What TotalView provides:
  - Easy python debugging session setup.
  - Fully integrated Python and C/C++ call stack.
  - ”Glue” layers between the languages removed.
  - Easily examine and compare variables in Python and C++.
  - Modest system requirements.
  - Utilize reverse debugging and memory debugging.

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



Python debugging with TotalView (demo)

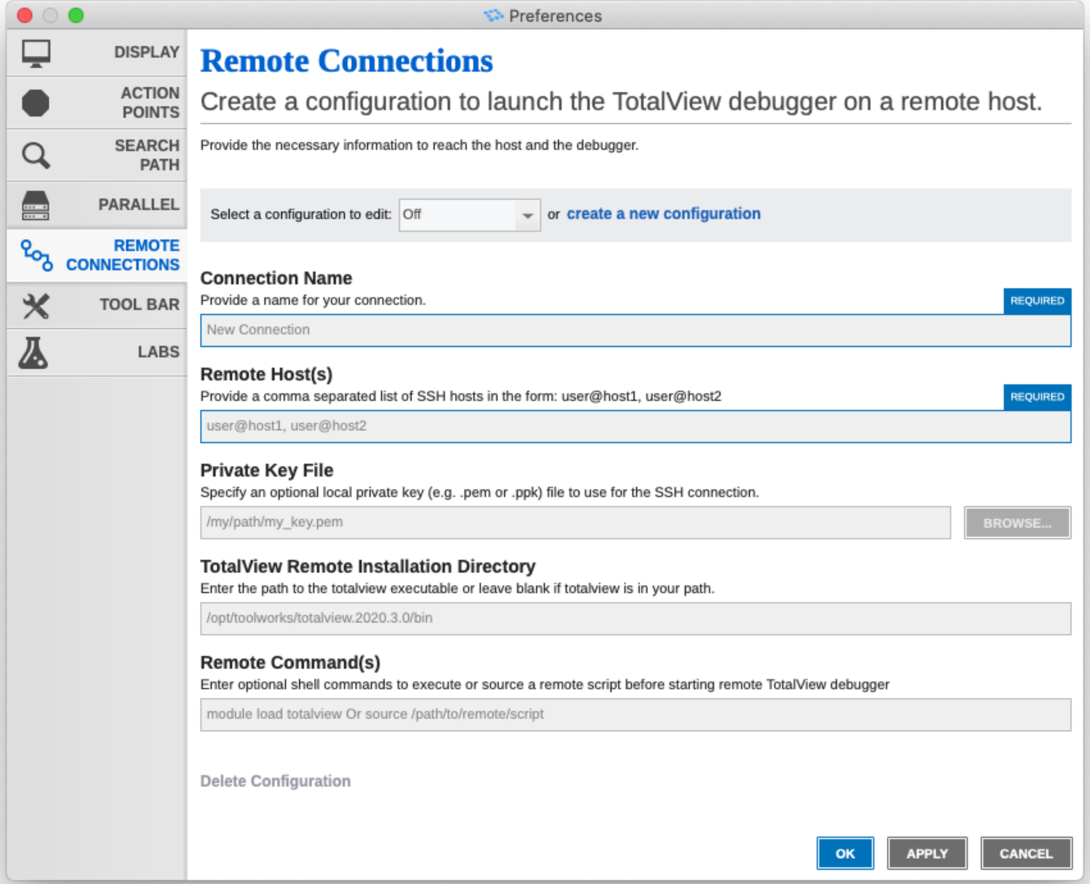
```
#!/usr/bin/python

def callFact():
   import tv_python_example as tp
   a = 3
   b = 10
   c = a+b
   ch = “local string”
    ……
   return tp.fact(a)
if __name__ == '__main__’:
   b = 2
   result = callFact()
   print result
```


Terminal
8:59 AM
Terminal
ubuntu:~/demo_2019/PythonExamples> /usr/toolworks/totalview.2019.0.4/bin/totalvi
ew -args python2.7-dbg test_python_types.py

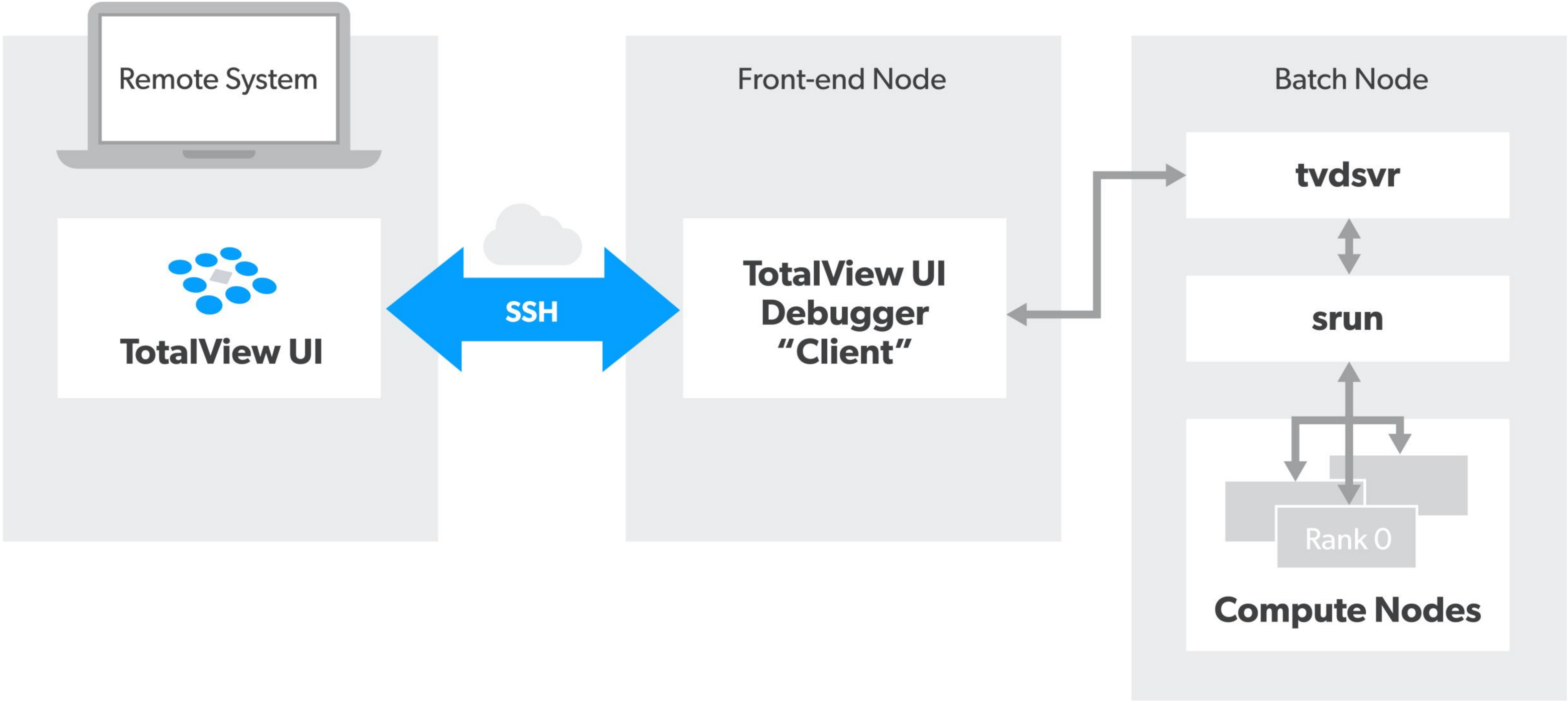
Remote Debugging - TotalView Remote UI

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.

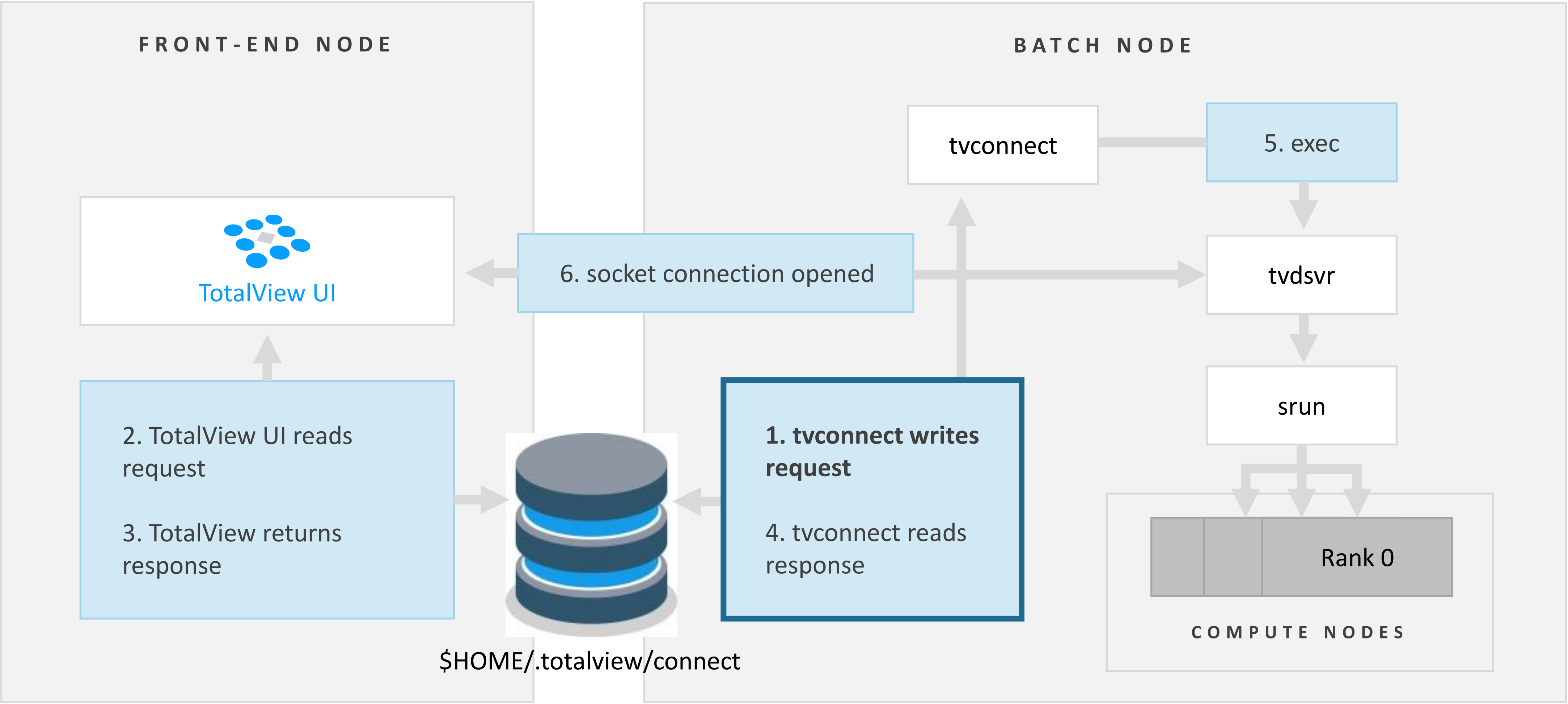


Remote UI Architecture

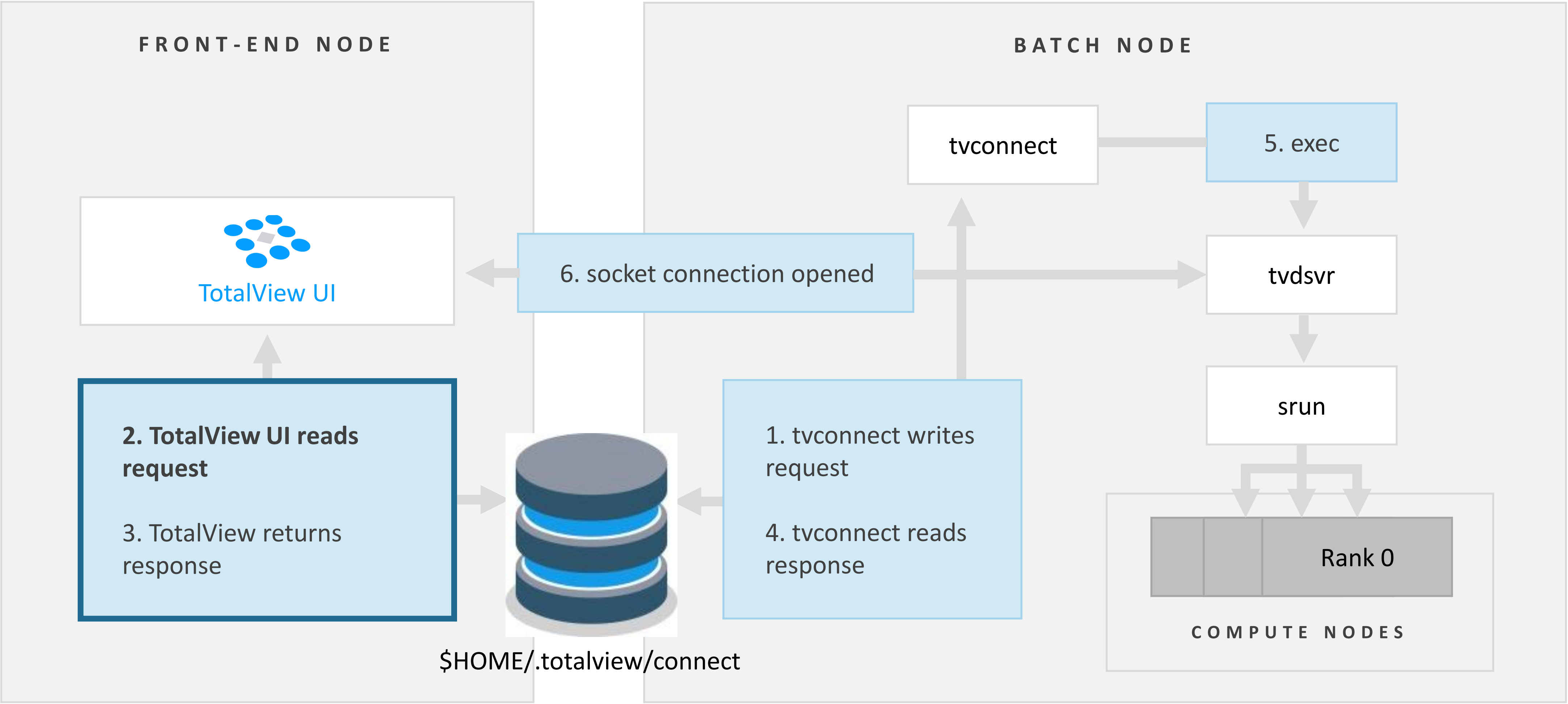


TotalView Reverse Connections

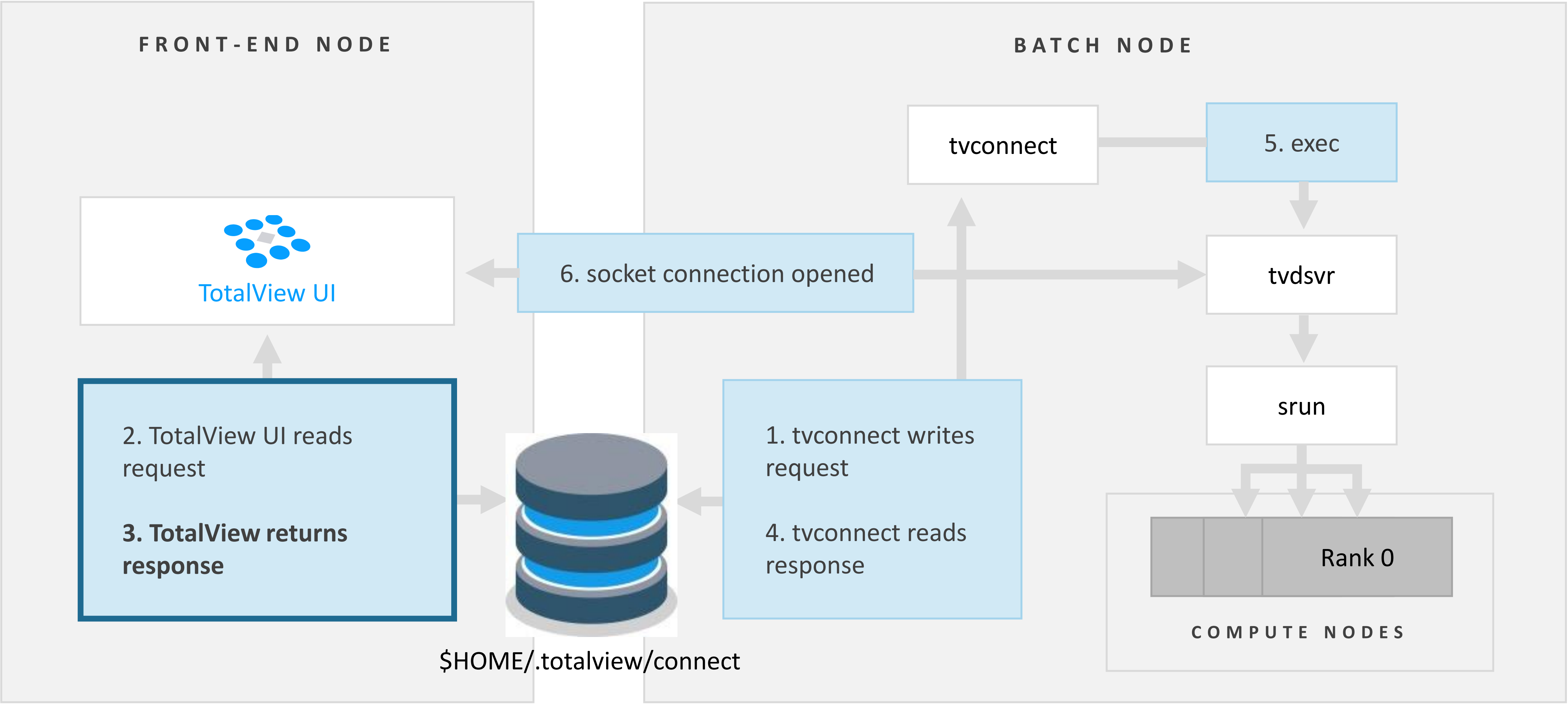
Reverse Connection Flow



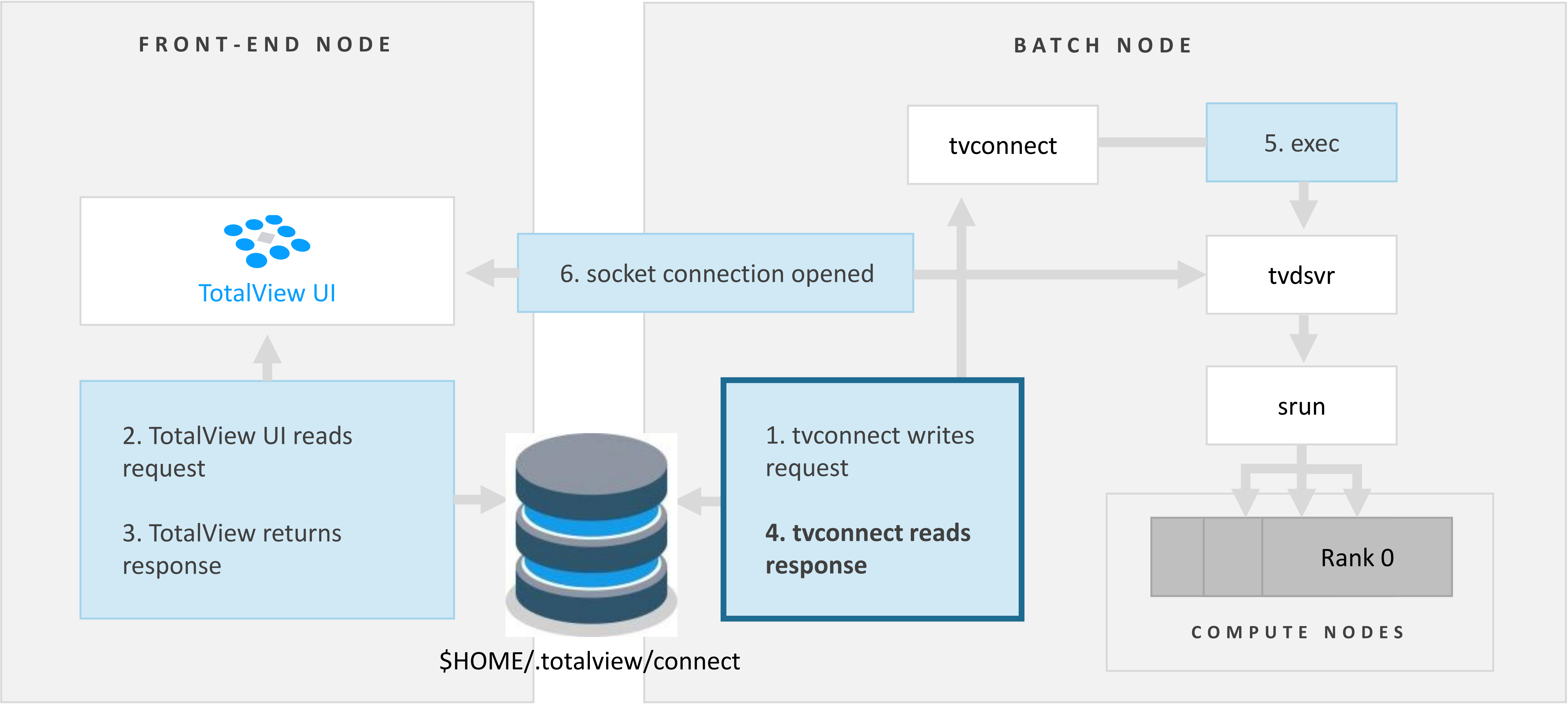
Reverse Connection Flow



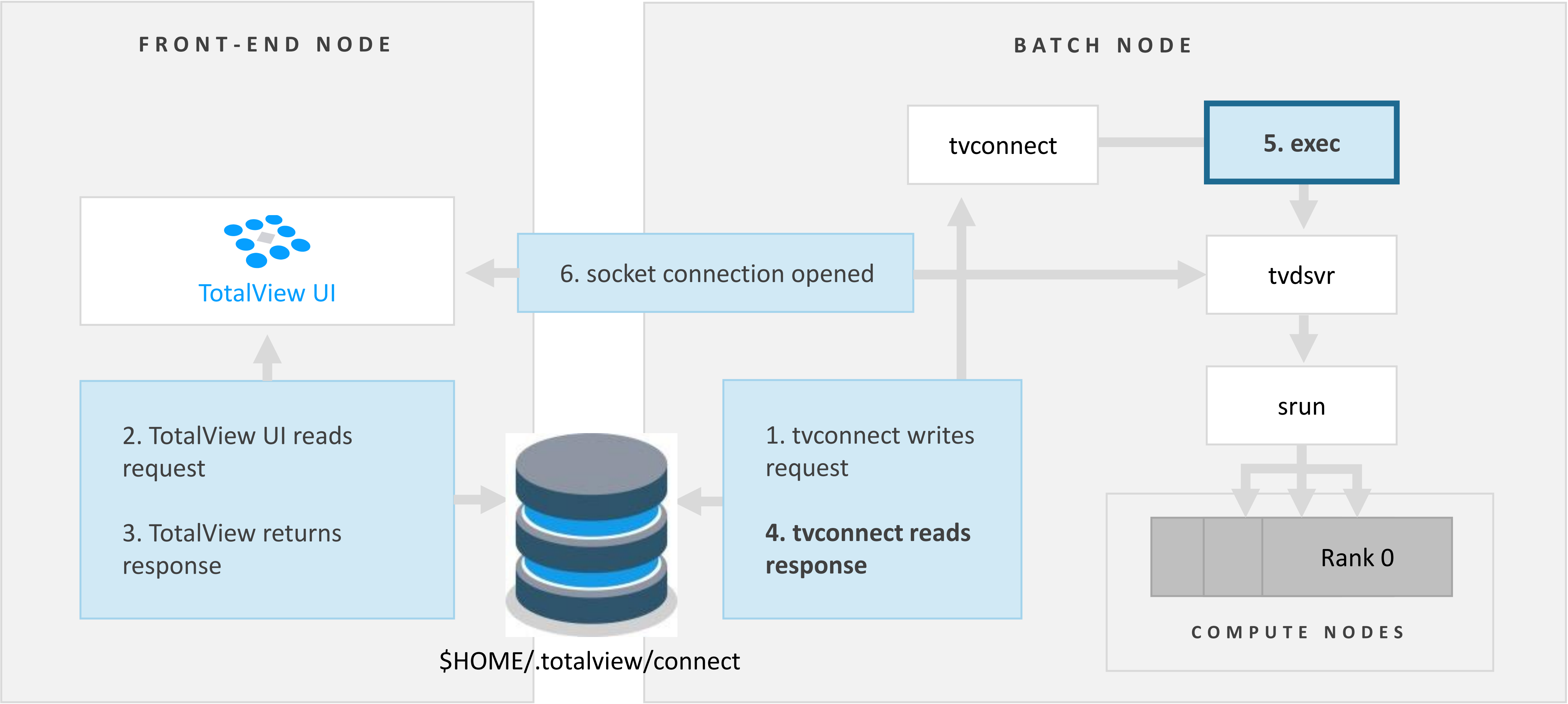
Reverse Connection Flow



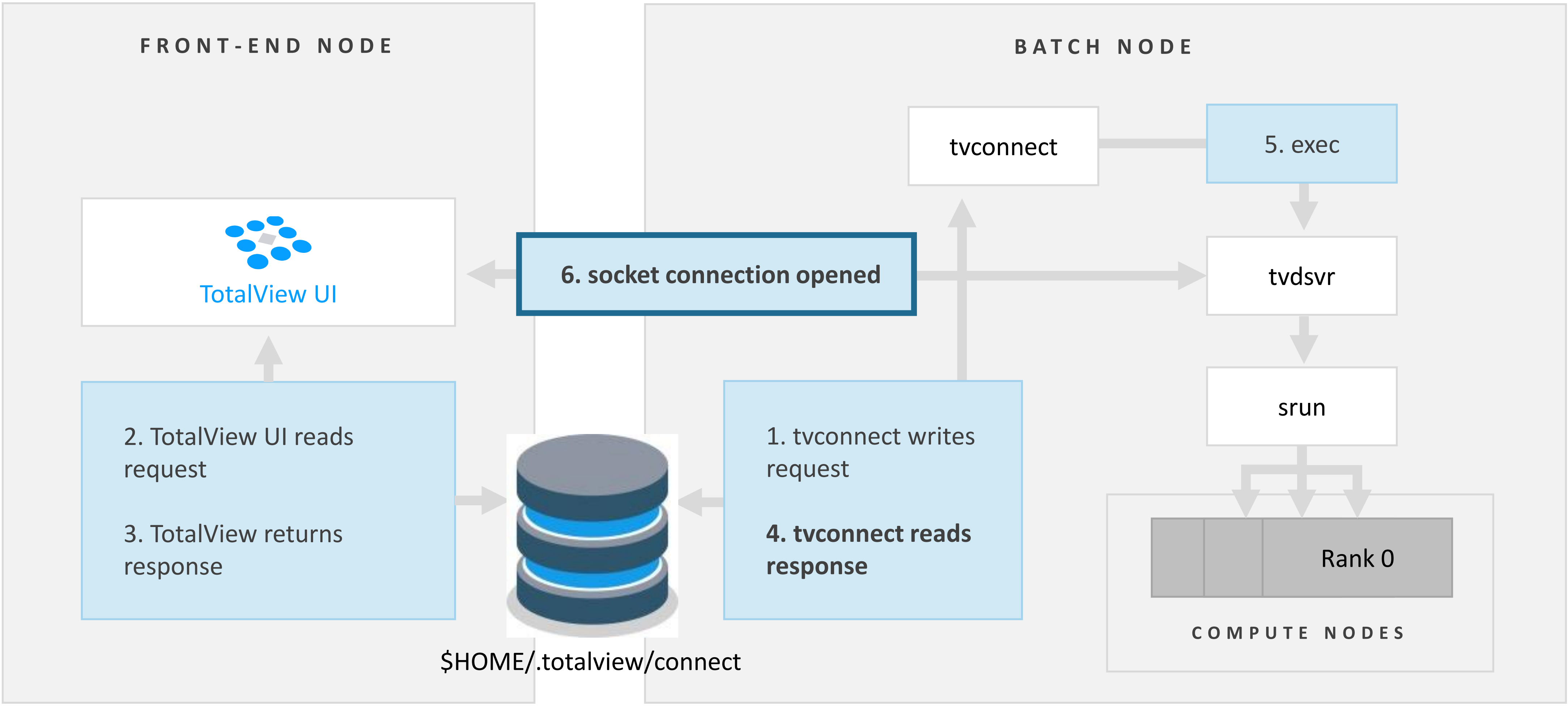
Reverse Connection Flow



Reverse Connection Flow



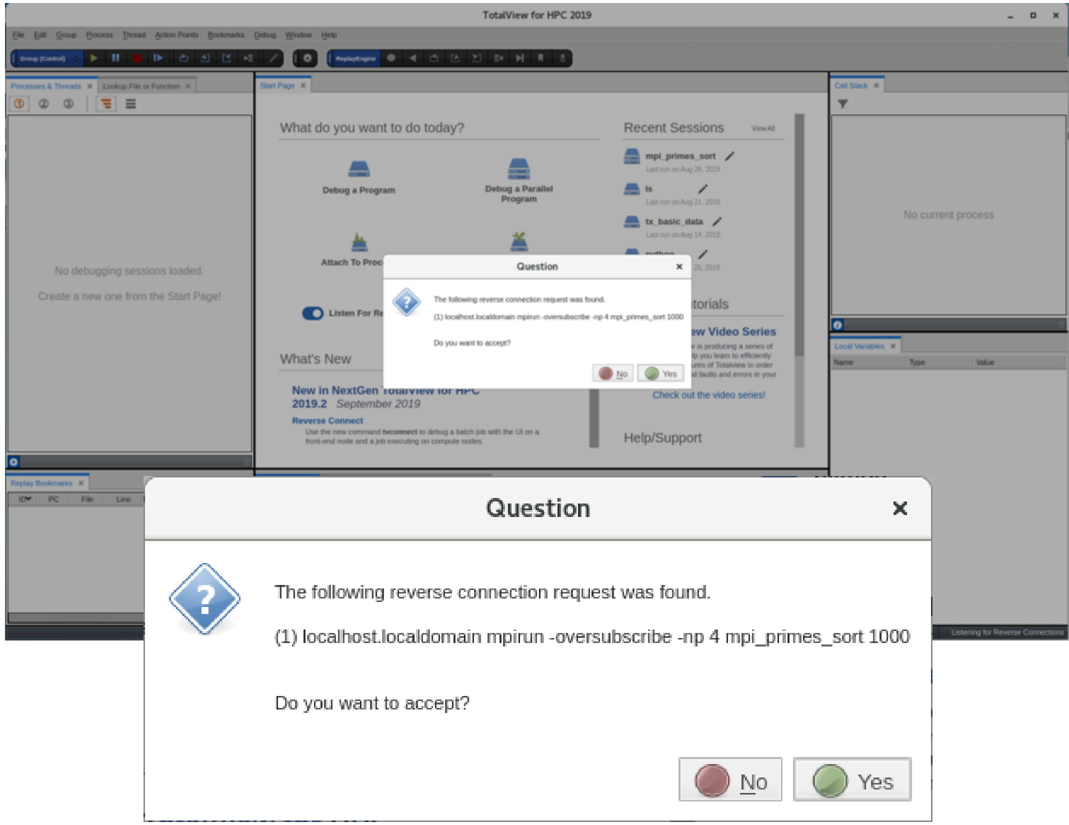
Reverse Connection Flow



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
```



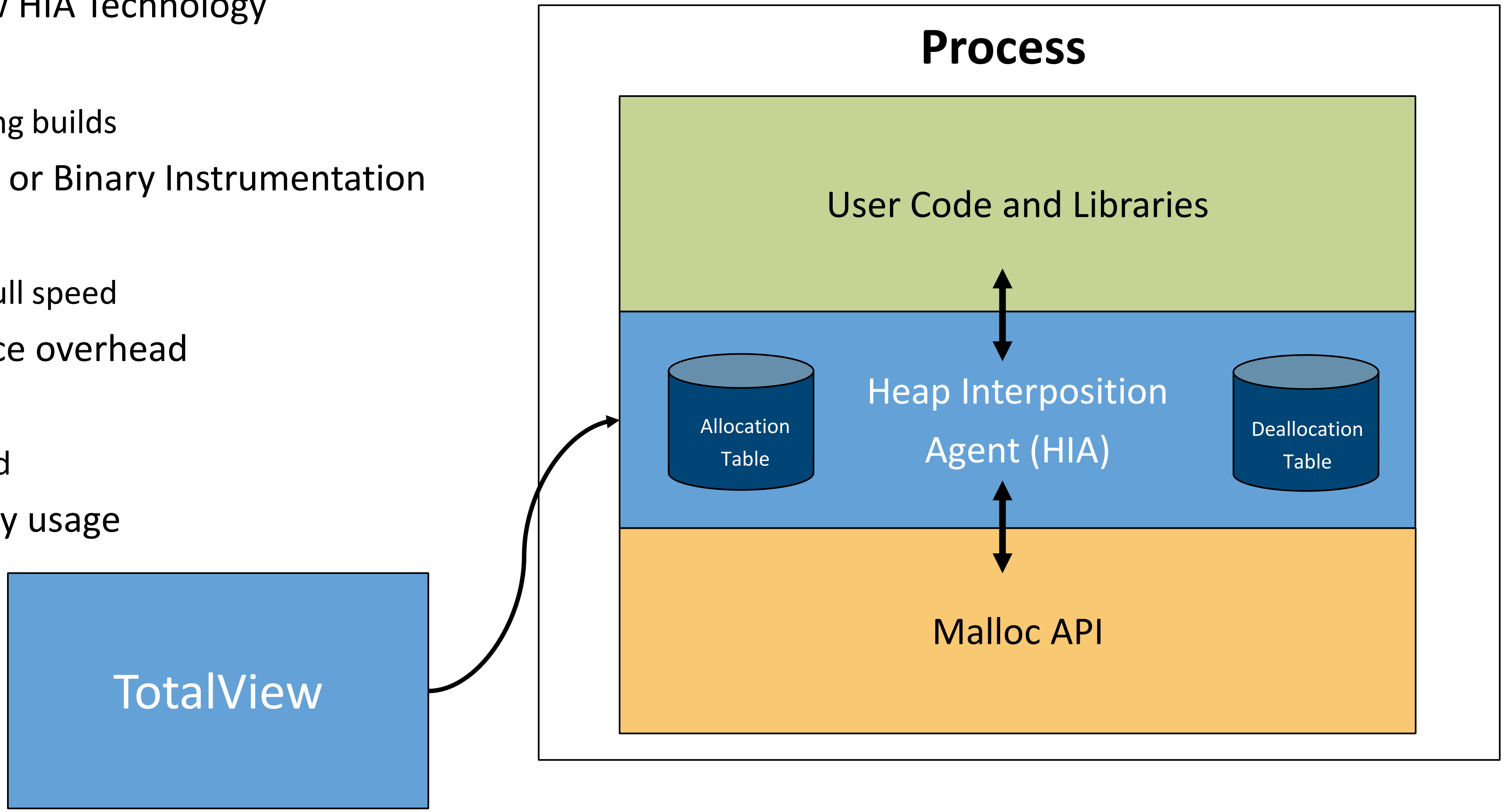
Memory Leaks, Heap Status, and Identifying Dangling Pointers

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



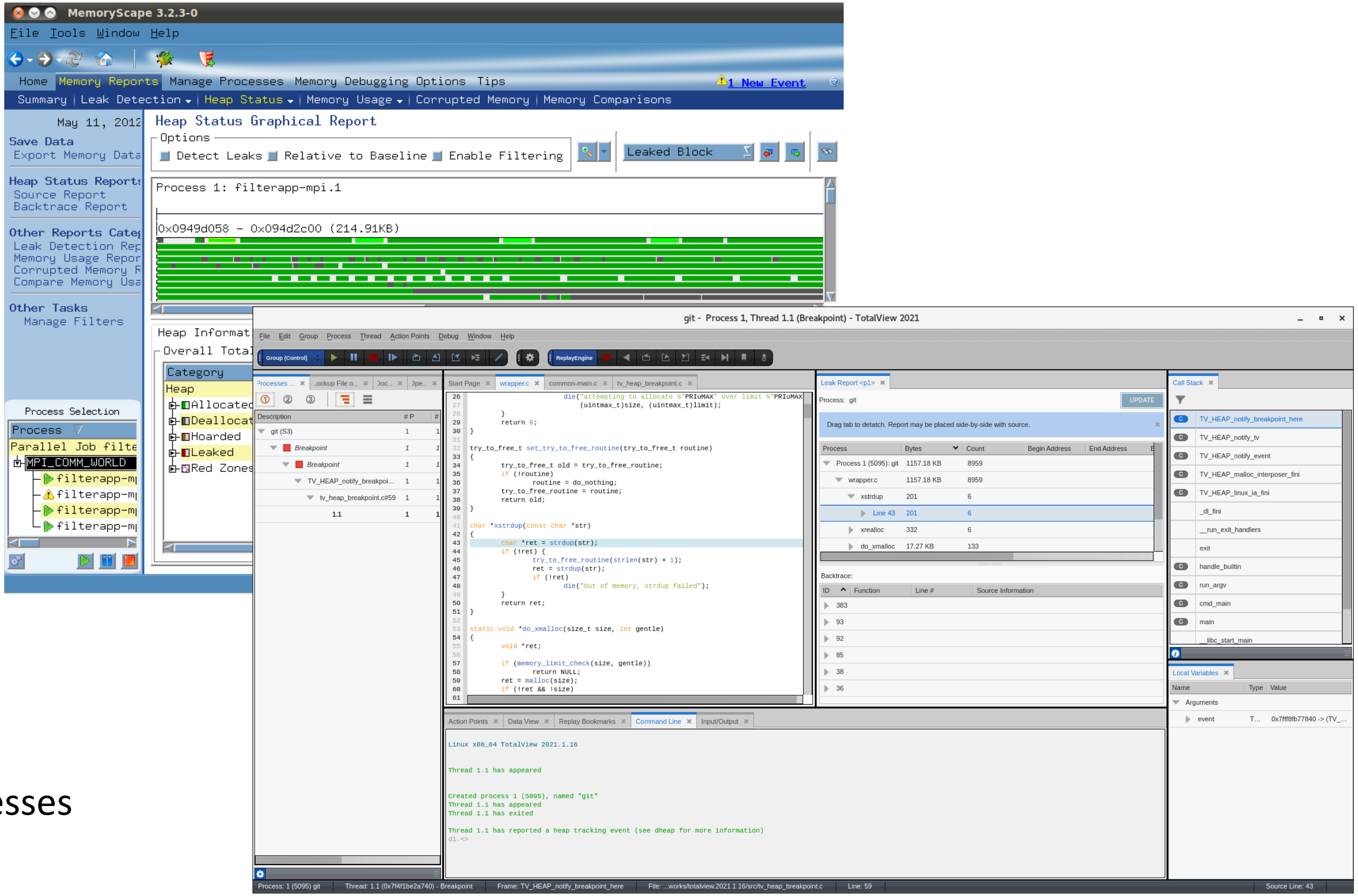
Memory Debugging Features – MemoryScape / TotalView

TotalView New UI Features

- Leak detection
- Heap Status
- Dangling pointer detection

Coming Features

- Memory Error Events
- Memory Corruption Detection
- Memory Block Painting
- Memory Hoarding
- Memory Comparisons between processes



TotalView Reverse Debugging

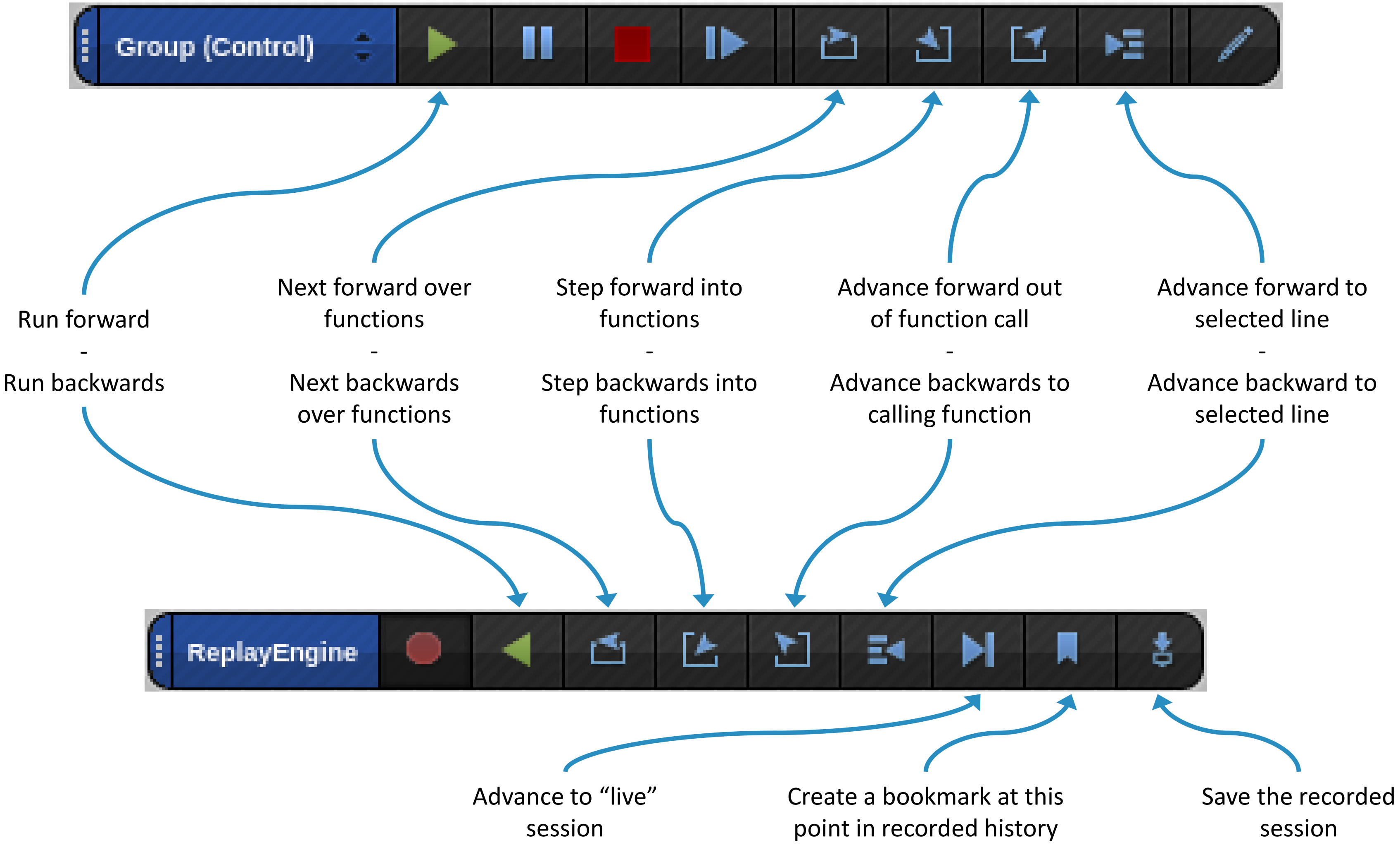
Reverse Debugging with TotalView

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

Start Page | common-main.c

```c
static void restore_sigpipe_to_default(void)
{
        sigset_t unblock;

        sigemptyset(&unblock);
        sigaddset(&unblock, SIGPIPE);
        sigprocmask(SIG_UNBLOCK, &unblock, NULL);
        signal(SIGPIPE, SIG_DFL);
}

int main(int argc, const char **argv)
{
        /*
         * Always open file descriptors 0/1/2 to avoid clobbering files
         * in die().  It also avoids messing up when the pipes are dup'ed
         * onto stdin/stdout/stderr in the child processes we spawn.
         */
        sanitize_stdfds();

        git_setup_gettext();

        git_extract_argv0_path(argv[0]);

        restore_sigpipe_to_default();

        return cmd_main(argc, argv);
}
```

Reverse Debugging Controls



Debugging CUDA Applications with TotalView

TotalView for the NVIDIA® GPU Accelerator

- NVIDIA Tesla, Fermi, Kepler, Pascal, Volta, Turing, Ampere
- NVIDIA CUDA 9.2, 10 and 11
  - With support for Unified Memory
- Debugging 64-bit CUDA programs
- 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

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
```

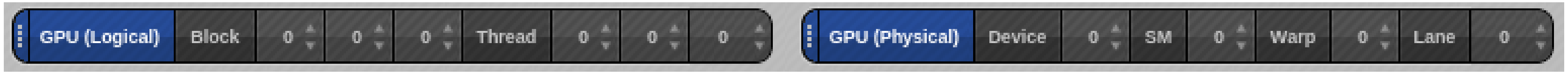
Breakpoint Set in CUDA Kernel Code Before Launch

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

Start Page | tx_cuda_matmul.cu

```

// Matrix multiplication kernel called by MatrixMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  // Block row and column
  int blockRow = blockIdx.y;
  int blockCol = blockIdx.x;
  // Each thread block computes one sub-matrix Csub of C
  Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
  // Each thread computes one element of Csub
  // by accumulating results into Cvalue
  float Cvalue = 0;
  // Thread row and column within Csub
  int row = threadIdx.y;
  int col = threadIdx.x;
  // Loop over all the sub-matrices of A and B that are
  // required to compute Csub
  // Multiply each pair of sub-matrices together
  // and accumulate the results
  for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
    // Get sub-matrix Asub of A
    Matrix Asub = GetSubMatrix(A, blockRow, m);
    // Get sub-matrix Bsub of B
    Matrix Bsub = GetSubMatrix(B, m, blockCol);
    // Shared memory used to store Asub and Bsub respectively
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // Load Asub and Bsub from device memory to shared memory
    // Each thread loads one element of each sub-matrix
    As[row][col] = GetElement(Asub, row, col);
    Bs[row][col] = GetElement(Bsub, row, col);
    // Synchronize to make sure the sub-matrices are loaded
    // before starting the computation
    __syncthreads();
    // Multiply Asub and Bsub together
    for (int e = 0; e < BLOCK_SIZE; ++e)
      Cvalue += As[row][e] * Bs[e][col];
    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();
  }
  // Write Csub to device memory
```

GPU Physical and Logical Toolbars



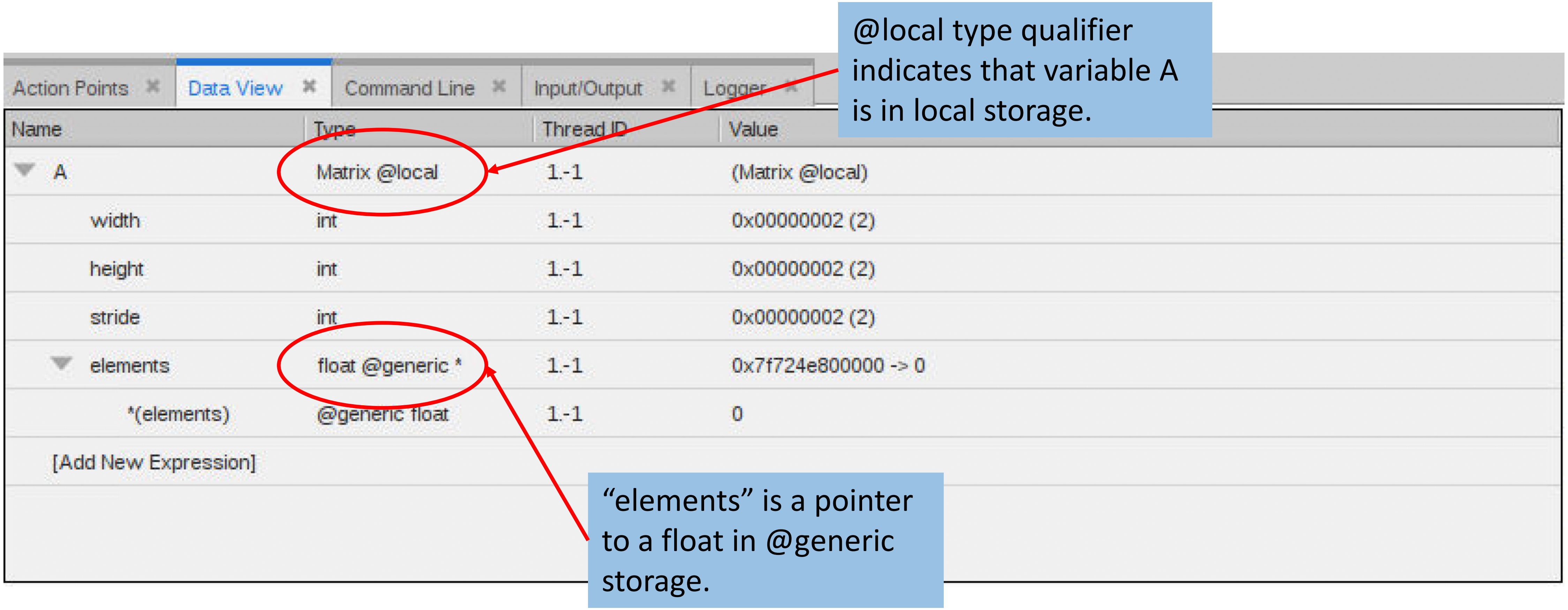
Logical toolbar displays the Block and Thread coordinates.

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 thread selector on the logical toolbar to focus on a specific GPU thread.

Displaying CUDA Program Elements

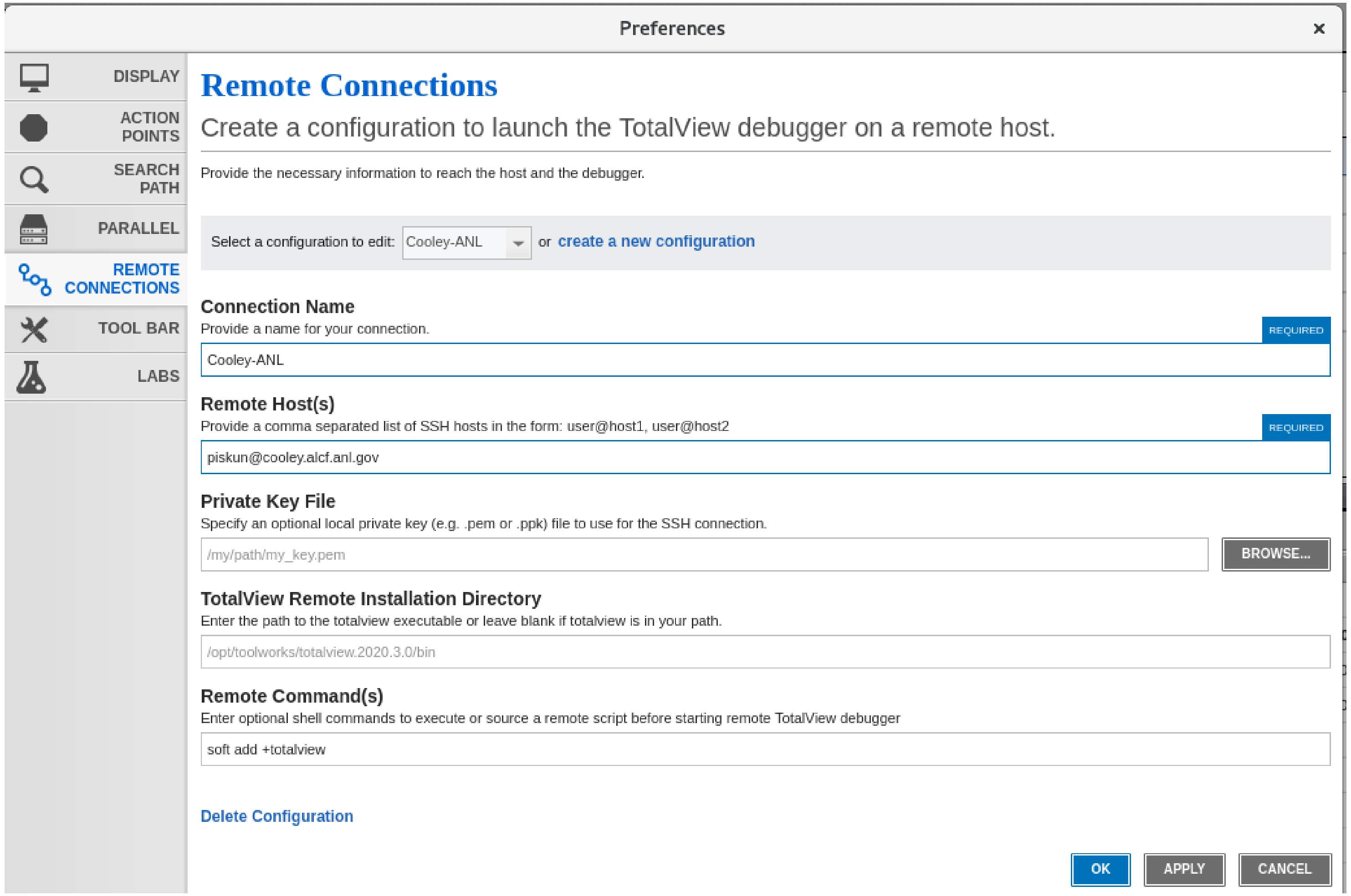


- The identifier @local is a TotalView built-in type storage qualifier that tells the debugger the storage kind of "A" is local storage.
- The debugger uses the storage qualifier to determine how to locate A in device memory

Using TotalView for Parallel Debugging on ANL

TotalView remote debugging on Linux and Mac OS

- Download and install TotalView on your linux or mac.
- Connect to remote front node.
- Run labs remotely



Hands-on labs

- Install TV from installers on Mac or Linux.
  - Ignore license code
- Star TotalView
- Remotely connect to cooley and enable Reverse Connection

Labs:

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

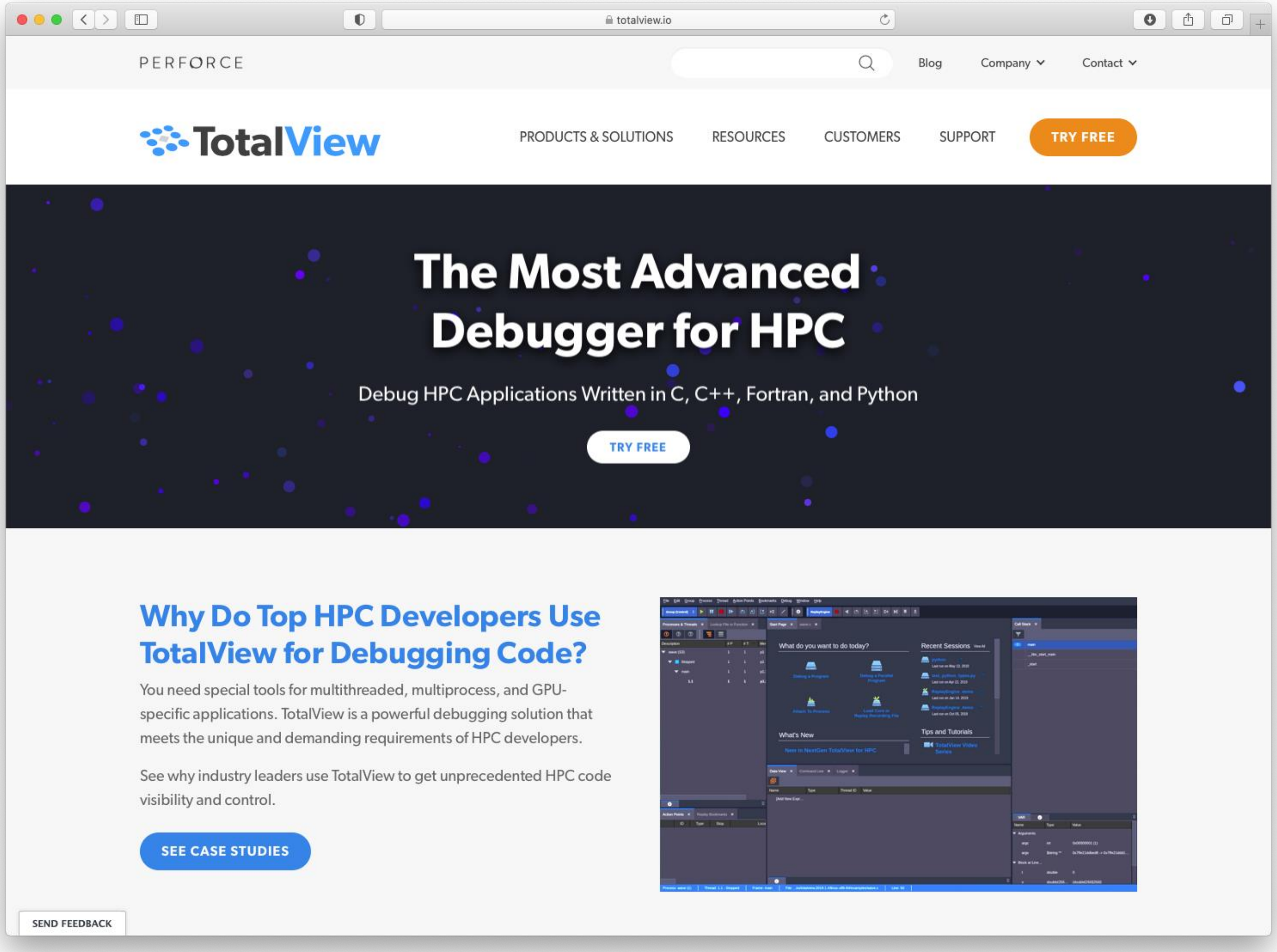
TotalView is available on Theta, Cooley

- Installed at: /soft/debuggers/totalview-2021-08-04/toolworks/totalview.2021X.3.756/bin/totalview

- Connect to Cooley/Theta
  - Get allocation first
    - qsub -A ATPESC2021 –n 4 –q debug-flat-quad –I (theta)
    - qsub -A ATPESC2021 –n 4 –q training –I (Cooley)
    - module load totalview (theta)
    - soft add +totalview (cooley)
  - totalview -args mpiexec –np <N>   ./demoMpi_v2
  - tvconnect mpiexec –np <N>   ./demoMpi_v2

TotalView Resources and Documentation

- TotalView website:
  - https://totalview.io
- TotalView documentation:
  - https://help.totalview.io
- TotalView Video Tutorials:
  - https://totalview.io/support/video-tutorials
- Other Resources:
  - Blog: https://totalview.io/blog



Summary

- Use of modern debugger saves you time.

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

THANK YOU