



ATPESC 2017

TotalView: Debugging from Desktop to Supercomputer

Peter Thompson

Principal Software Support Engineer

August 8, 2017



Some thoughts on debugging

- As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.
 - Maurice Wilkes
- Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.
 - Brian W. Kernigan
- Sometimes it pays to stay in bed on Monday, rather than spending the rest of the week debugging Monday's code.
 - Dan Saloman

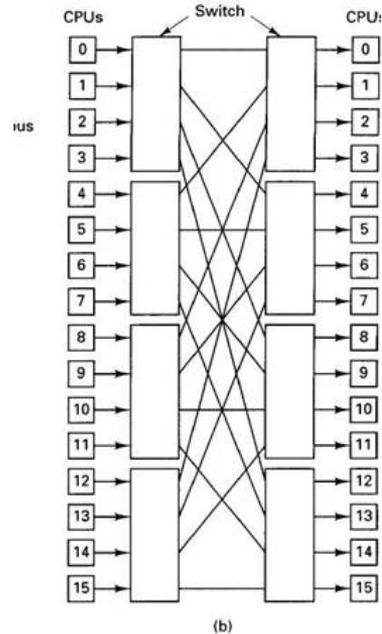
Rogue Wave's Debugging Tool

TotalView for HPC

- Source code debugger for C/C++/Fortran
 - Visibility into applications
 - Control over applications
- Scalability
- Usability
- Support for HPC platforms and languages

TotalView Overview

TotalView Origins



Mid-1980's Bolt, Berenak, and Newman (BBN) Butterfly Machine
An early 'Massively Parallel' computer

How do you debug a Butterfly?

- TotalView project was developed as a solution for this environment
 - Able to debug multiple processes and threads
 - Point and click interface
 - Multiple and Mixed Language Support
- Core development group has been there from the beginning and have been/are involved in defining MPI interfaces, DWARF, and lately OMPD (Open MP debugging interface)

Other capabilities added

- Support for most types of MPI
- Lightweight Memory Debugging
- Type transformations – STL and user containers
- Memscript and tvscript
- Reverse Debugging - only on Linux x86-64
- Remote Display Client
- GPU debugging
- Intel Xeon Phi – Including KNL
- Most popular platforms, Linux, Mac, Solaris, AIX... but not Windows
- ARM64
- Python Debugging support – currently in progress

Key TotalView Features

- Multi-process and Multi-thread debugging
- Interactive Memory Debugging
- Reverse Debugging
- Unattended Debugging
- Remote Display Client
- CUDA Debugging
- Xeon Phi Debugging

Serial, Parallel and Accelerated applications

Multi-process and Multi-thread Debugging

- Supports/Supported by most MPI flavors
 - Automatic process acquisition across nodes with lightweight debug servers in an MRNet tree configuration
 - Can attach to a running MPI job
- Support for OpenMP and pthreads
 - Ability to hold and control individual threads
- Mixed Multi-process and Multi-threaded programs
- Breakpoint control on the Group, process and thread level

TotalView's Memory Efficiency

- TotalView is lightweight in the back-end (server)
- Servers don't "steal" memory from the application
- Each server is a multi-process debugger agent
 - One server can debug thousands of processes
 - Not a conglomeration of single process debuggers
 - TotalView's architecture provides flexibility (e.g., P/SVR)
 - No artificial limits to accommodate the debugger (e.g., BG/Q 1P/CN)
- Symbols are read, stored, and shared in the front-end (client)
- Example: LLNL APP ADB, 920 shlibs, Linux, 64 P, 4 CN, 16 P/CN, 1 SVR/CN



Process	VSZ (largest, MB)	RSS (largest, MB)
TV Client	4,469	3,998
MRNet CP	497	4
TV Server	304	53

Memory Debugging

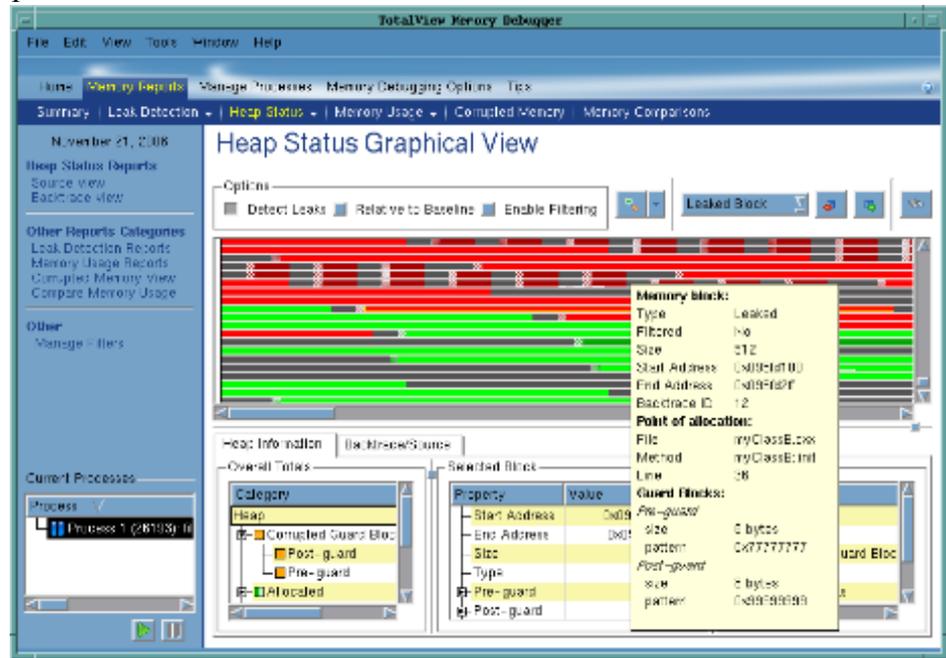
How do you find buffer overflows or memory leaks?

Runtime Memory Analysis : Eliminate Memory Errors

- Detects memory leaks *before* they are a problem
- Explore heap memory usage

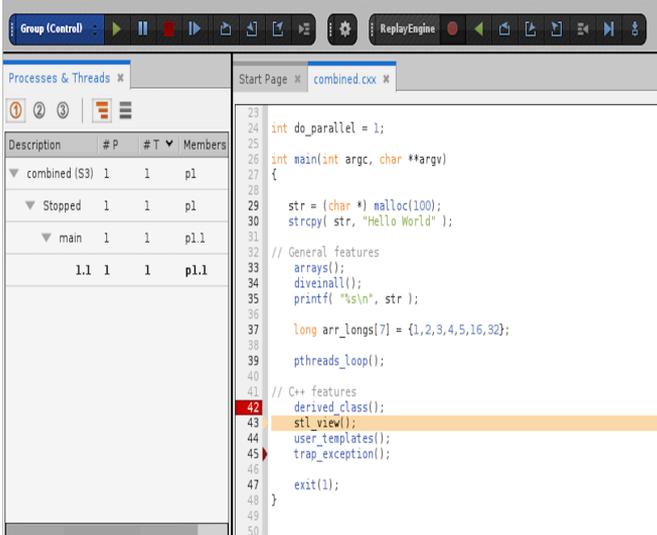
Features

- Detects
 - Malloc API misuse
 - Memory leaks
 - Buffer overflows
- Low runtime overhead
- Easy to use
 - Works with vendor libraries
 - No recompilation
 - No instrumentation
 - Link against HIA for MPI



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
 - Start recording
 - Set a breakpoint
 - 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



```
23
24
25
26 int do_parallel = 1;
27
28 int main(int argc, char **argv)
29 {
30     str = (char *) malloc(100);
31     strcpy( str, "Hello World" );
32
33 // General features
34 arrays();
35 diveinall();
36 printf( "%s\n", str );
37
38 long arr longs[7] = {1,2,3,4,5,16,32};
39 pthreads_loop();
40
41 // C++ features
42 derived class();
43 stl_view();
44 user_templates();
45 trap_exception();
46
47 exit(1);
48
49
50
```

Unattended Debugging

Memscript and Tvscrip

- Command line invocation to run TotalView and Memoryscape unattended
- tvscript can be used to set breakpoints, take actions at those breakpoints and have the results logged to a file. It can also do memory debugging
 - `tvscript -create_actionpoint "method1=>display_backtrace show_arguments" \ -create_actionpoint "method.c#342=>print x" myprog -a dataset 1`
- memscript can be used to run memory debugging on processes and display data when a memory event takes place. Exit is ALWAYS an event

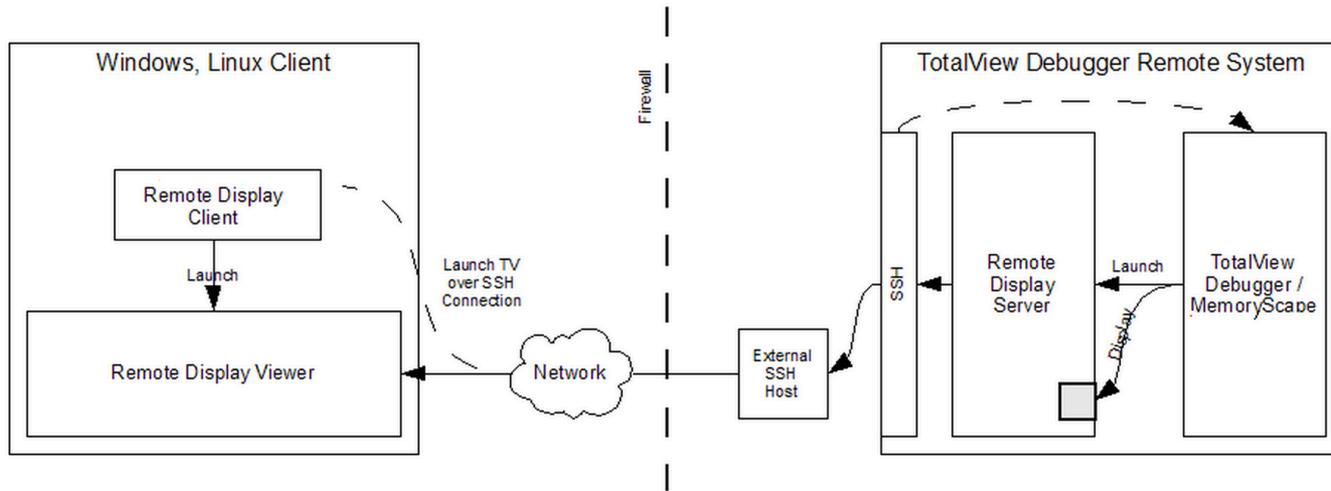
```
Memscrip -event_action \ "alloc_null=list_allocations,any_event=check_guard_blocks" \  
-guard_blocks -maxruntime "00:30:00" -display_specifiers \ "noshow_pc,noshow_block_address,show_image"\  
myProgram -a myProgramArg1
```

- Memscript data can be saved in html, memory debug file, text heap status file

Remote Display Client (RDC)

- Push X11 bits and events across wide networks can be painful. The RDC can help

Figure 17 – Remote Display Components



The RDC setup

The screenshot displays the Rogue Wave RDC setup interface. On the left, a 'Session Profiles' sidebar lists 'perseid' and 'vesta'. The main area is titled '1. Enter the Remote Host to run your debug session:' and contains a 'Remote Host' field with 'vesta.alcf.anl.gov', a 'User Name' dropdown with 'thompson', and an 'Advanced Options' button. Below this is section '2. As needed, enter hosts in access order to reach the Remote Host:', which includes a table with columns for Host, Access By, Access Value, and Commands. The table has two rows, both with 'User Name' in the 'Access By' column. Section '3. Enter settings for the debug session on the Remote Host:' features radio buttons for 'TotalView' (selected) and 'MemoryScape', followed by fields for 'Path to TotalView on Remote Host' (set to '/soft/debuggers/totalview/bin/totalview'), 'Arguments for TotalView', 'Your Executable (path & name):' (set to 'runjob'), 'Arguments for Your Executable' (set to '-p 1 --np 512 --block \${COBALT_PARTNAME} : ALLc2'), and 'Submit Job to Batch Queueing System' (set to 'Custom'). Section '4. Enter batch submission settings for the Remote Host:' includes fields for 'Submit Command' (set to 'qsub'), 'Script to execute via Submit Command' (set to './tv_PBS.csh'), and 'Additional Submit Command Options' (set to '-q ATPESC2015 -t 60 -n 512 --mode script -O LOG'). A 'Launch Debug Session' button is at the bottom.

Rogue Wave
SOFTWARE

Session Profiles:
perseid
vesta

1. Enter the Remote Host to run your debug session:
Remote Host: User Name

2. As needed, enter hosts in access order to reach the Remote Host:

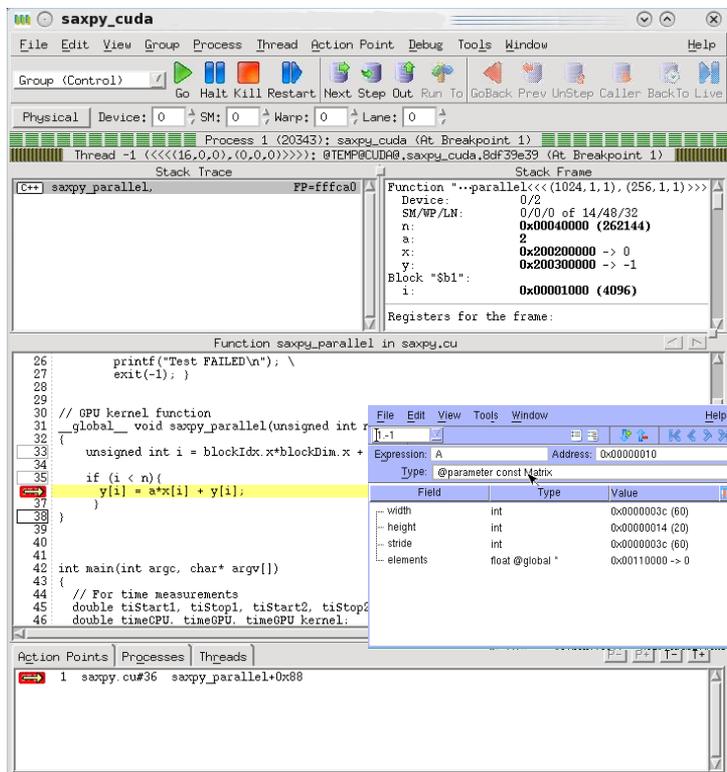
	Host	Access By	Access Value	Commands
1		User Name		
2		User Name		

3. Enter settings for the debug session on the Remote Host:
 TotalView MemoryScape

Path to TotalView on Remote Host:
Arguments for TotalView:
Your Executable (path & name):
Arguments for Your Executable:
Submit Job to Batch Queueing System:

4. Enter batch submission settings for the Remote Host:
Submit Command:
Script to execute via Submit Command:
Additional Submit Command Options:

TotalView for the NVIDIA® GPU Accelerator



- NVIDIA CUDA 6.5, 7.0, 7.5, 8.0 – (testing 9.0)
- 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 **CUDA Core** debugging
- Leverages CUDA memcheck
- Support for **OpenACC**

TotalView for the Intel® Xeon Phi™ coprocessor

Supports All Major Intel Xeon Phi Coprocessor Configurations

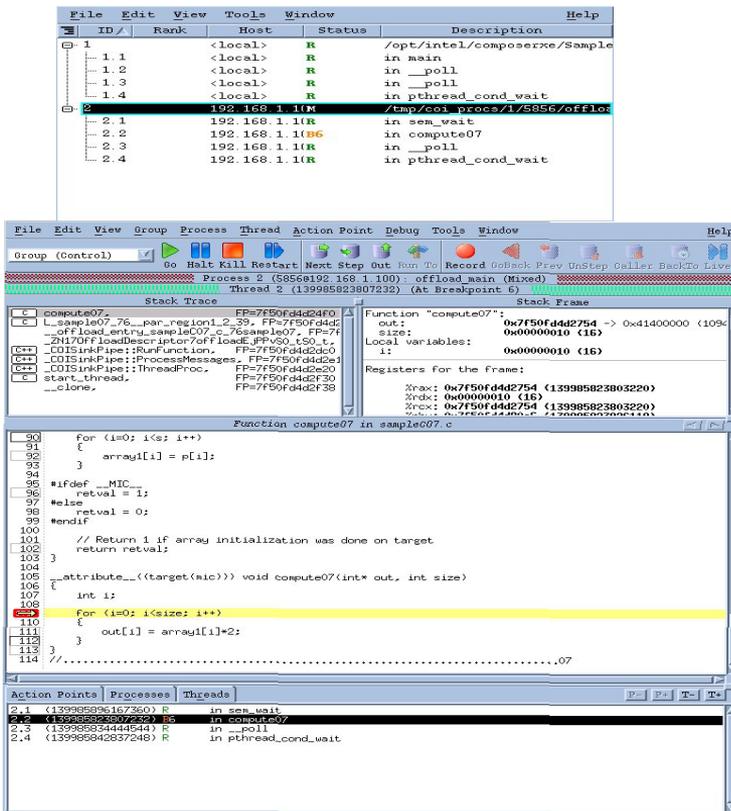
- Native Mode
 - With or without MPI
- Offload Directives
 - Incremental adoption, similar to GPU
- Symmetric Mode
 - Host and Coprocessor
- **Multi-device, Multi-node**
- Clusters
- KNL Support – Just works like a normal node
 - AVX2 support being added

User Interface

- MPI Debugging Features
 - Process Control, View Across, Shared Breakpoints
- Heterogeneous Debugging
 - Debug Both Xeon and Intel Xeon Phi Processes

Memory Debugging

- Both native and symmetric mode



Knights Landing Memory

- KNL has on-board high bandwidth memory (MCDRAM) which can be accessed much faster than going out to main memory.
 - Cache
 - Explicitly managed for placement of frequently accessed data
- MemoryScape will be able to track allocations made both the standard heap and the on-chip HBM
- Optimization may include making sure that the right data structures are available to the processor in HBM
 - MemoryScape can show you data structure usage and placement
- KNL machines online - right here! Let's test this...

TotalView – Next Generation

What's New?

Linux OpenPower (LE) support with GPU

- Support for OpenPower (Linux power LE)
 - All major functionality
 - Support for CUDA Debugging on GPU Accelerators
- Currently working with IBM and Lawrence Livermore to support the CORAL systems (Power 8 nodes with 4 Nvidia PASCAL cards)

New UI Framework – aka CodeDynamics

The screenshot displays the Rogue Wave IDE interface with the following components:

- Debugger:** Breakpoint at line 564 in tx_fork_loop.cxx.
- Process Tree:**

Description	#P	#T	Members
Breakpoint	4	4	p1-4
_select	4	4	p2-3.1, p1.2,
1.2	1	1	p1.2
2.1	1	1	p2.1
3.1	1	1	p3.1
4.3	1	1	p4.3
wait_a_while	4	8	p1.1, p4.1, p2
1.1	1	1	p1.1
1.3	1	1	p1.3
2.2	1	1	p2.2
2.3	1	1	p2.3
3.2	1	1	p3.2
3.3	1	1	p3.3
4.1	1	1	p4.1
- Source Code:**

```
564
565 if (do_segV && me == do_segV_index)
566 {
567     struct timeval timeout;
568     long bad_addr;
569     int *foo;
570     int bar;
571     wait_a_while (&timeout);
572     bad_addr = -3;
573     foo = (int *)bad_addr;
574     bar = *foo;
575     *foo = bar + 1;
576 }
577
578 for (;;)
579 {
580     struct timeval timeout;
581     wait_a_while (&timeout);
582     if (verbose)
583         printf ("Thread %ld woke up in Snore!\n", (long)pthread_self());
584     if (use_mut)
585     {
586         if (!please_shut_up)
587         {
588             #if defined __alpha && !defined __linux
589                 printf ("Thread %ld (posix = %ld): Trying for the lock\n",
590                     (long)pthread_t(pthread_self())->_sequence,
591                     (long)pthread_self());
592             #elif _AIX
593                 printf ("Thread (ktd) %d: Trying for the lock\n", (int)(pthread_self()));
594             #else
595                 printf ("Thread (posix) ID=%ld: Trying for the lock\n",
596                     (long)pthread_self());
597             #endif
598             /* if ??
599             fflush (stdout);
600             pthread_mutex_lock (&mutex);
601             */

```
- Call Stack:**

Name	Type	Value
_select		
wait_a_while		
snore		
forker		
fork_wrapper		
main		
__libc_start_main		
_start		
- Arguments:**

Name	Type	Value
arg	void *	0x00000000
Block		\$b1#52
timeout	struct timeval	{struct timeval}
Block		\$b1
me	int	0x00000000 (0)
old_ticket	int	0xffffffff (-1)
ticket	int	0x00000000 (0)
- Action Points:**

ID	Type	File	Line
1	com.roguewave.totalview.breakpoint	tx_fork_loop.cxx	564
- Command Line:**

```
Thread 1.2 hit breakpoint 1 at line 564 in "wait_a_while(time val)"
Thread 3.2 hit breakpoint 1 at line 564 in "wait_a_while(time val)"
Thread 4.1 hit breakpoint 1 at line 564 in "wait_a_while(time val)"
Thread 2.3 hit breakpoint 1 at line 564 in "wait_a_while(time val)"
Thread 1.1 hit breakpoint 1 at line 564 in "wait_a_while(time val)"
^!<=
```

Python Support

- Recently added to add in debugging mixed language programs
 - Still in development stages, but a good start

Calling C/C++ from Python

- Legacy libraries are written in C/C++ and Fortran
 - Run faster
 - Rewriting doesn't make sense
- Luckily there are many ways to call between the languages

Python C/C++ glue technology	Description
ctypes	A foreign function library for Python.
Cython	A superset of the Python language that additionally supports calling C functions and declaring C types on variables and class attributes.
SWIG	A software development tool that connects programs written in C and C++ with a variety of high-level programming languages including Python.
CFFI	Foreign Function Interface for Python calling C code.
PyQt/PySide and SIP	SIP is a tool that makes it easy to create Python bindings for C and C++ libraries.
Boost.Python	A C++ library which enables seamless interoperability between C++ and the Python programming language.

Python without Filtering

No viewing of Python data & code

The screenshot shows the CodeDynamics 2017X debugger interface. The main window displays a C program with a Python call. A green circle highlights the 'fact' function in the call stack, with a green box labeled 'Glue code' next to it. The call stack also shows other functions like '_wrap_fact', 'PyCFunction_Call', 'call_function', 'PyEval_EvalFrameEx', 'fast_function', 'call_function', 'PyEval_EvalFrameEx', 'PyEval_EvalCodeEx', 'PyEval_EvalCode', 'run_mod', 'PyRun_FileExFlags', and 'PyRun_SimpleFileExFlags'. The 'Arguments' section shows a variable 'n' of type 'int' with value '0x00000003 (3)'. The 'Data View' section is empty, showing '[Add New Expression]'. The 'Command Line' section shows the following output:

```
Thread 1.1 has appeared
Created process 1 (14759), named "python2.7-dbg"
Thread 1.1 has appeared
Thread 1.1 has exited
Thread 1.1 hit breakpoint 1 at line 6 in "fact(int)"
d1.> std
disabled
d1.<>
```

Showing C code with mixed data

Glue code filtered out - Python data available for viewing

The screenshot shows the CodeDynamics 2017X debugger interface. The main window displays C code from 'example.c' with a breakpoint at line 6. The code includes a 'fact' function and a 'square_array' function. The 'Process' pane on the left shows the current process and thread. The 'Call Stack' pane on the right shows the current function call stack. The 'Data View' pane at the bottom right shows the current state of variables 'n', 'a', and 'b'. A green circle highlights the 'fact' function in the call stack, and a green arrow points to the 'n' variable in the data view. Another green arrow points to the 'a' and 'b' variables in the data view.

```
1 /* File: example.c */
2
3 #include "tx_example.h"
4
5 int fact(int n) {
6     if (n < 0) { /* This should probably return an error, but this is simpler */
7         return 0;
8     }
9     if (n == 0) {
10        return 1;
11    }
12    else {
13        /* testing for overflow would be a good idea here */
14        return n * fact(n-1);
15    }
16 }
17
18 int getSquare(int n) {
19     return n * n;
20 }
21
22 void square_array(std::vector< std::vector < int > > myArray)
23 {
24     for (int i=0; i<3; i++)
25         for (int j=0; j<2; j++)
26             myArray[i][j] = getSquare(myArray[i][j]);
27
28     /* Print the results */
29     for (int i=0; i<3; i++)
30         for (int j=0; j<2; j++)
```

Shows Python & C++

Name	Type	Value
n	int	0x00000003 (3)
a	int	0x0000000000000003 (3)
b	int	0x000000000000000a (10)
[Add New Expression]		

C++ data

Py data

Python with filtering

Python code available - Program counter shows calling location

The screenshot displays the CodeDynamics 2017X debugger interface. The main window shows the source code of a Python script with a breakpoint set at line 6. The call stack on the right shows the current function call and its callers. The variables window shows the current state of variables, including arguments and local variables.

```
1 #!/usr/bin/python
2
3
4 def getFact(int_arg):
5     # Test some locals
6     a = 3
7     b = 10
8     c = a+b
9     ch = "local string"
10    pi = 3.14159
11    long_var = 2.5
12    true_bool_var = True
13    false_bool_var = False
14    noType = None
15    cx = complex(2,-1)
16
17    return _tx_python_example.fact(a)
18 if __name__ == '__main__':
19     b = 2
20     result = getFact(b)
21     print result
22
```

Call Stack

Frame	Function
004	fact
003	_wrap_fact
002	getFact
001	<module>
	__libc_start_main

VAR

Name	Type	Value
Arguments		
int_arg	int	0x0000000000000002 (2)
_tx_python_example	module	0x7f3eec118338 -> (PyModuleObject)
a	int	0x0000000000000003 (3)
b	int	0x000000000000000a (10)
c	int	0x0000000000000004 (13)

Action Points

ID	Type	Stop	Location	Line
1	Break	Process	tx_python_example.c	6

Data View

Name	Type	Value
a	int	0x0000000000000003 (3)

Process: python2.7-dbg (1) Thread: 1.1 - Breakpoint Frame: getFact File: ...gger/src/tests/bld/gcc_4.8_64/test_python_to_C.py Line: 17 Source Line: 22

Debug Fission – Split Dwarf Support

Debug Information takes up a lot of Space

- Line and symbol information generally represented in DWARF format
 - Allows us to show the source code and locate variables
 - The larger and more complex the code, the more data is needed to represent it. This can grow to GB's in size
- DebugFission SplitDwarf, gdb_index, dwz methods of dealing with this are now all supported.

Using TotalView

Using TotalView

For HPC we have two methods to start the debugger

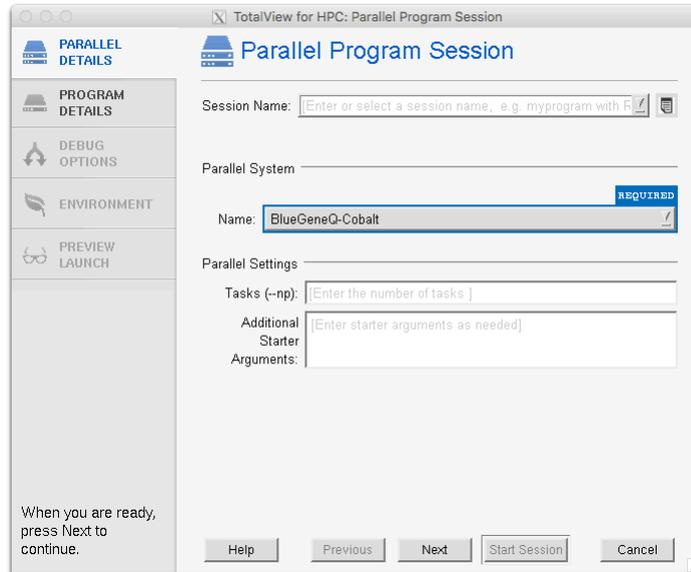
The 'classic' method

- `totalview -args mpiexec -np 512 ./myMPIprog myarg1 myarg2`
- This will start up TotalView on the parallel starter (mpiexec, srun, runjob, etc) and when you hit 'Go' the job will start up and the processes will be automatically attached. At that point you will see your source and can set breakpoints.
- Some points to consider...
 - You don't see your source at first, since we're 'debugging' the mpi starter
 - Some MPI's don't support the process acquisition method (most do, but might be stripped of symbols we need when packaging)
 - In general more scalable than the next method...

Starting TotalView

The 'indirect' method

- Simply 'totalview' or 'totalview myMPIprog' and then you can choose a parallel system, number of tasks, nodes, and arguments to the program.
- With this method the program source is available immediately
- Less dependent on MPI starter symbols
- May not be as scalable as some 'indirect' methods launch a debug server per process



The New UI for HPC

- MPI debugging with the new UI requires starting in 'classic' mode with the `-newUI` argument
 - `totalview -newUI -args mpiexec -np 4 ./cpi`
- Python debugging support stack transform only in newUI

The screenshot displays the TotalView for HPC 2016 interface. The main window shows the source code for 'cpi.c' with a red line at line 35. The right sidebar contains several panels: 'Processes & Threads' showing a table of process information, 'Call Stack' showing the main function, and 'Data View' showing the values of variables like 'myfield'. The bottom left shows the 'Command Line' and 'Logger' windows with a list of thread events.

Description	R/P	#T	Members
__pid_...	12	12	0.11.2
2.2	1	1	0.2
3.2	1	1	1.2
4.2	1	1	2.2
5.2	1	1	3.2
6.2	1	1	4.2

Name	Type	Value
myfield	str... (struct Field)	
x	float	1
y	float	2
value	ds...	3.14159265358979
myfield.value	ds...	3.14159265358979

Thread 12.2 has exited
Thread 13.2 has appeared
Thread 13.1 has appeared
Thread 13.3 has appeared
Thread 13.4 has appeared
Thread 13.1 hit breakpoint 1 at line 35 in "main"
Thread 12.1 hit breakpoint 1 at line 35 in "main"
Thread 11.1 hit breakpoint 1 at line 35 in "main"
Thread 10.1 hit breakpoint 1 at line 35 in "main"
Thread 9.1 hit breakpoint 1 at line 35 in "main"
Thread 8.1 hit breakpoint 1 at line 35 in "main"
Thread 7.1 hit breakpoint 1 at line 35 in "main"
Thread 6.1 hit breakpoint 1 at line 35 in "main"
Thread 5.1 hit breakpoint 1 at line 35 in "main"
Thread 4.1 hit breakpoint 1 at line 35 in "main"
Thread 3.1 hit breakpoint 1 at line 35 in "main"
Thread 2.1 hit breakpoint 1 at line 35 in "main"
Thread 1.1 hit breakpoint 1 at line 35 in "main"
Thread 1.2 stopped: Stop Signal
Thread 1.1 stopped: Stop Signal

Using TotalView at Argonne

- Modules available on Theta, Vesta, Mira
 - module load totalview
- Memory Debugging on BG/Q and Cray should link against the agent, either static or dynamically
 - BG/Q:
 - `-L<path> -Wl,@<path>/tvheap_bgqs.ld #static`
 - `-L<path> -ltvheap_64 -Wl,-rpath,<path> #dynamic`
 - Cray:
 - `-L<path> -ltvheap_cnl # static`
 - `-L<path> -ltvheap_cnl -Wl,-rpath,<path> #dynamic`
 - `<path>` = Path to platform specific TV lib
 - `export TVLIB=/soft/debuggers/totalview-2017-07-26/toolworks/totalview.2017.2.10/linux-x86-64/lib`
 - Substitute linux-power on BlueGene

Job Control at Argonne

- TotalView can be run on simple serial programs on login nodes (though maybe not the preferred method)
- MPI jobs require an allocation, either an interactive session (`qsub -I`) or through a batch script that creates an interactive session.
- Tvscrip and memscrip can be run totally in batch.
- Examples will be provided (After I confirm they work!)

And that's all...

- See me for demos of particular features or to try TotalView on your code

Our products and services



Tools

Klocwork On-the-fly static code analysis for app security

CodeDynamics Commercial dynamic analysis

OpenLogic Support Enterprise-grade SLA support

OpenLogic Audits Detailed open source license and security risk guidance

TotalView for HPC Scalable debugging

Zend Server Enterprise PHP app server

Zend Studio PHP IDE

Zend Guard PHP encoding and obfuscation



Libraries

SourcePro OS, database, network, and analysis abstraction for C++

Visualization Real-time data visualization at scale

PV-WAVE Visual data analysis

IMSL Numerical Libraries Scalable math and statistics algorithms

HydraExpress SOA/C++ modernization framework

HostAccess Terminal emulation for Windows

Stingray MFC GUI components



 **RogueWave**
S O F T W A R E

Innovate with Confidence