



# 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

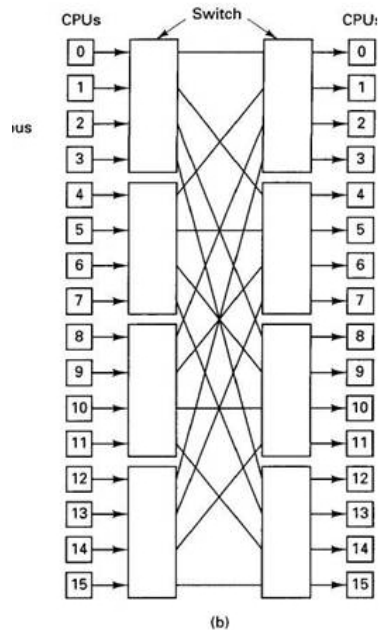
# Rogue Wave's Debugging Tool

## 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

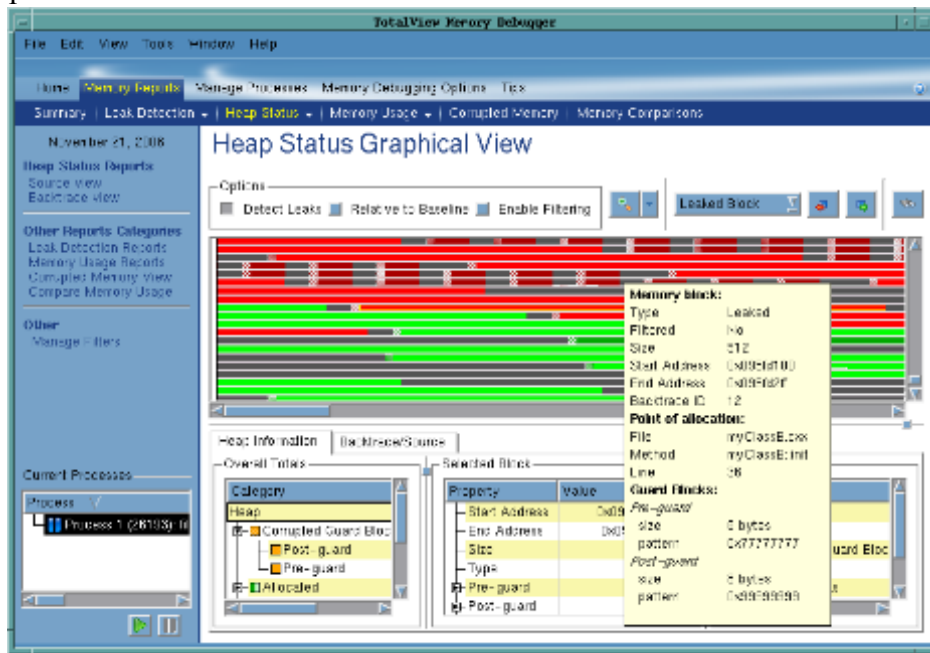
How do you find buffer overflows or memory leaks?

## Runtime Memory Analysis : Eliminate Memory Errors

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

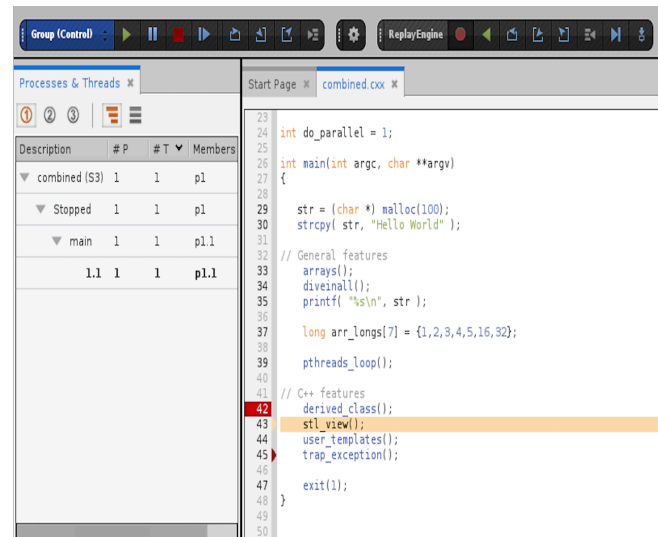
## Features

- 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



# 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

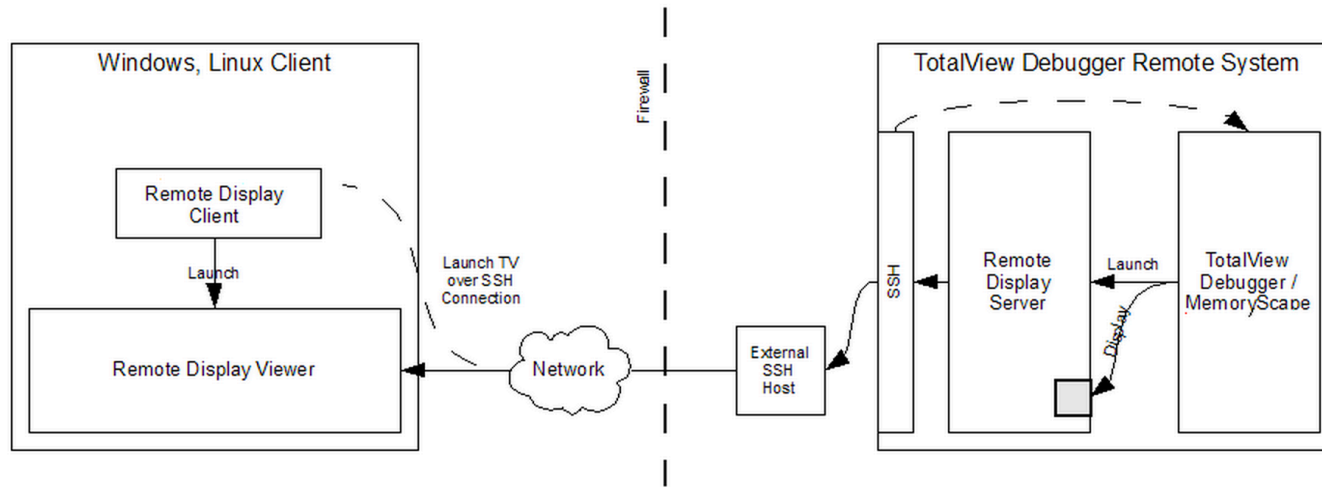
**Memscrip** `-event_action \ "alloc_null=list_allocations,any_event=check_guard_blocks" \`  
`-guard_blocks -maxruntime "00:30:00" -display_specifiers \ "noshow_pc,noshow_block_address,show_image" \`  
`myProgram -a myProgramArg1`

- 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



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:

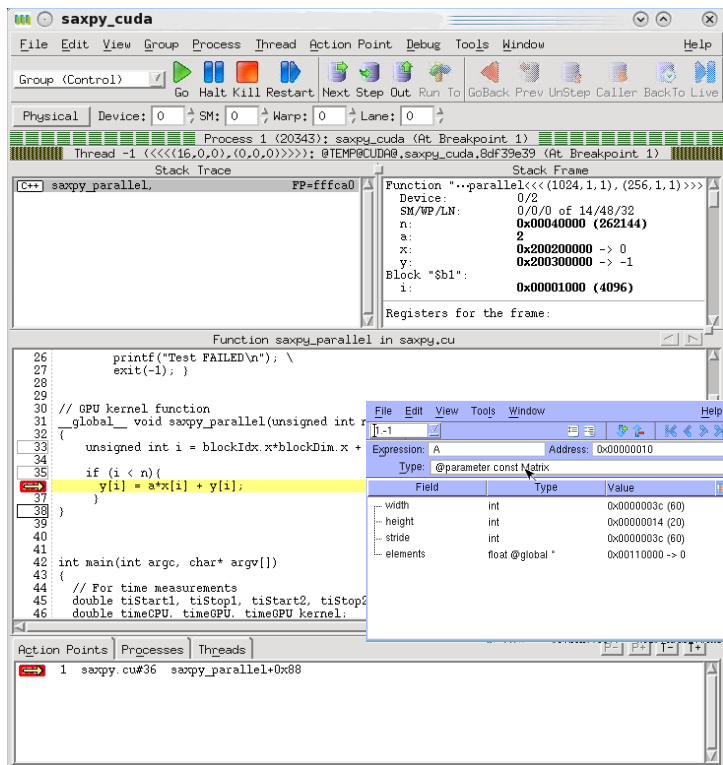
4. Enter batch submission settings for the Remote Host:

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

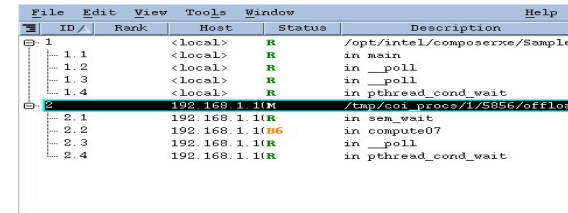
- Native Mode
  - With or without MPI
- Offload Directives
  - Incremental adoption, similar to GPU
- Symmetric Mode
  - Host and Coprocessor
- Multi-device, Multi-node**
- Clusters
  - AVX2 support being added

## User Interface

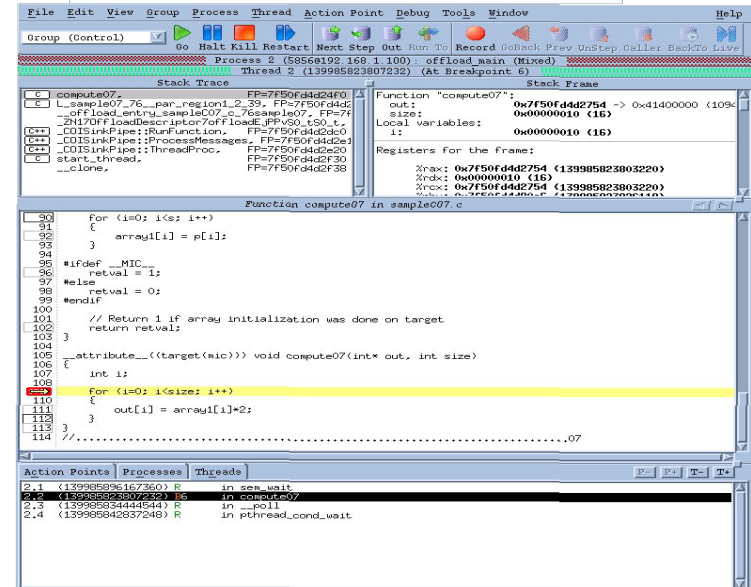
- MPI Debugging Features
  - Process Control, View Across, Shared Breakpoints
- Heterogeneous Debugging
  - Debug Both Xeon and Intel Xeon Phi Processes

## Memory Debugging

- Both native and symmetric mode



ID	Rank	Host	Status	Description
1		<local>	R	/opt/intel/composerxe/Sample
1.1		<local>	R	in main
1.2		<local>	R	in __poll
1.3		<local>	R	in __poll
1.4		<local>	R	in pthread_cond_wait
2		192.168.1.100	R	offload_main (Mixed)
2.1		192.168.1.100	R	in __poll
2.2		192.168.1.100	R	in compute07
2.3		192.168.1.100	R	in __poll
2.4		192.168.1.100	R	in pthread_cond_wait



The screenshot displays the TotalView debugger interface. The top window shows the process list with process 2 (offload\_main) selected. The bottom window shows the stack trace for process 2.2, which is at a breakpoint in the 'compute07' function. The 'Stack Frame' window shows the function signature and local variables. The 'Registers' window shows the values of registers for the frame. The 'Source Code' window shows the C code for the 'compute07' function, with the breakpoint set at line 110.

```
90 for (i=0; i<size; i++)
91 {
92     array[i] = p[i];
93 }
94
95 #ifdef __MIC__
96     retval = 1;
97 #else
98     retval = 0;
99 #endif
100
101 // Return 1 if array initialization was done on target
102 return retval;
103 }
104
105 #attribute__((target(mic))) void compute07(int* out, int size)
106 {
107     int i;
108
109     for (i=0; i<size; i++)
110     {
111         out[i] = array[i]*2;
112     }
113 }
114
```

# 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, showcasing the CodeDynamics UI framework. The main window is titled "Rogue Wave" and contains several panes:

- Processes and Threads:** A table showing the state of various threads. The table has columns for Description, #P, #T, and Members. The threads listed include Breakpoint, select, wait\_a\_while, and their respective members.
- Code Editor:** The central pane displays the source code of a C++ file named "tx\_fork\_loop.cxx". The code is a thread function that implements a loop with a timeout and a snore function. A breakpoint is set at line 564.
- Call Stack:** A pane on the right showing the current thread's call stack. The stack includes frames for \_select, wait\_a\_while, snore, forker, fork\_wrapper, main, \_libc\_start\_main, and \_start.
- Command Line:** A pane at the bottom right showing a log of thread events. The log includes messages such as "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\*)'", and "Thread 1.1 hit breakpoint 1 at line 564 in 'wait\_a\_while(time val\*)'".

The interface is designed for debugging and analyzing the execution of a program, providing a comprehensive view of the system's state and the flow of execution.

# Python Support

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

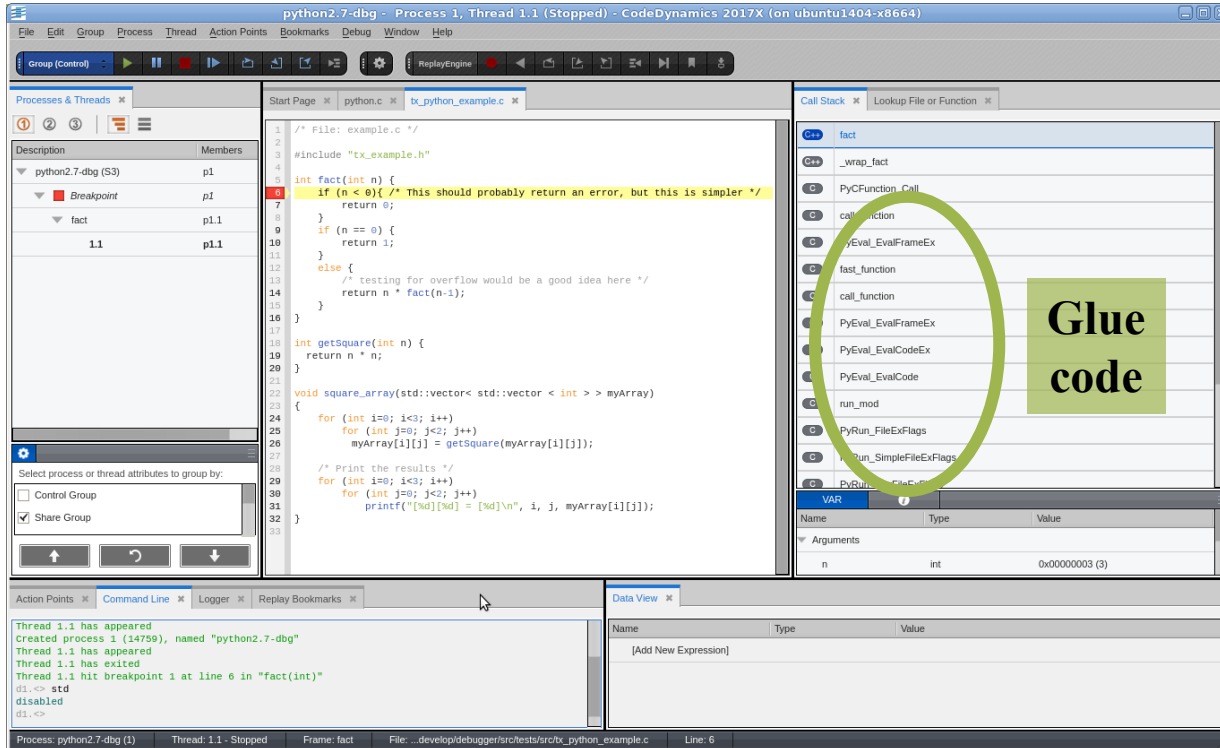
# Calling C/C++ from Python

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

# Python without Filtering

No viewing of Python data & code





## Showing C code with mixed data

Glue code filtered out - Python data available for viewing

The screenshot shows the Visual Studio Code interface with the following components:

- Top Bar:** Displays the current file path: `python2.7-dbg - Process 1.1 (Stopped) - CodeDynamics 2017X (on ubuntu1404-x8664)`.
- File Explorer:** Shows the project structure with files like `python2.7-dbg (S3)`, `Breakpoint`, `fact`, and `1.1`.
- Source Editor:** Displays the C++ code for `example.c`. The code includes a `fact` function and a `square_array` function. The `fact` function is currently selected, and its code is visible in the editor.
- Call Stack:** Shows the current call stack, with `fact` at the top. A green circle highlights the `fact` function, and a green callout box points to it with the text "Shows Python & C++".
- Data View:** Shows the current state of the program. It includes a table with columns `Name`, `Type`, and `Value`. The table contains the following data:
 

Name	Type	Value
<code>n</code>	<code>int</code>	<code>0x00000003 (3)</code>
<code>a</code>	<code>int</code>	<code>0x0000000000000003 (3)</code>
<code>b</code>	<code>int</code>	<code>0x000000000000000a (10)</code>

 A green callout box points to the `n` variable with the text "C++ data". Another green callout box points to the `a` and `b` arrays with the text "Py data".
- Bottom Bar:** Shows the current process: `Process: python2.7-dbg (1)`, `Thread: 1.1 - Stopped`, `Frame: fact`, `File: .../development/debugger/src/test/sr/fx_python_example.c`, and `Line: 6`.

# 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, `test_python_to_C.py`, with a breakpoint set at line 6. The left sidebar shows the 'Processes & Threads' pane with a tree view of the process `python2.7-dbg (S3)` and its threads. The right sidebar shows the 'Call Stack' pane, which lists the current call stack frames, including `fact`, `_wrap_fact`, and `getFact`. The bottom pane shows the 'Action Points' table, which lists the current action point (ID 1, Break, Process) and its location (Line 6). The status bar at the bottom indicates the current process, thread, frame, file, line, and source line.

python2.7-dbg - Process 1, Thread 1.1 (Breakpoint) - CodeDynamics 2017X (on ubuntu1404-x8664)

File Edit Group Process Thread Action Points Bookmarks Debug Window Help

Group (Control) [Run] [Pause] [Step Over] [Step Into] [Step Out] [ReplayEngine] [Previous] [Next] [First] [Last] [Breakpoints]

Processes & Threads

python2.7-dbg (S3) p1

Breakpoint p1

fact p1.1

1.1 p1.1

Select process or thread attributes to group by:

☐ Control Group

☒ Share Group

Start Page python.c tx\_python\_example.c test\_python\_to\_C.py tx\_python\_example\_wrap.cxx

```
1 #!/usr/bin/python
2
3 def getFact(int_arg):
4     import _tx_python_example
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
19 if __name__ == '__main__':
20     b = 2
21     result = getFact(b)
22     print result
```

Call Stack

fact

\_wrap\_fact

getFact

<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

Command Line Logger Replay Bookmarks

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

Data View

Name Type Value

a int 0x0000000000000003 (3)

[Add New Expression]

Process: python2.7-dbg (1) Thread: 1.1 - Breakpoint Frame: getFact File: ...ugger/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

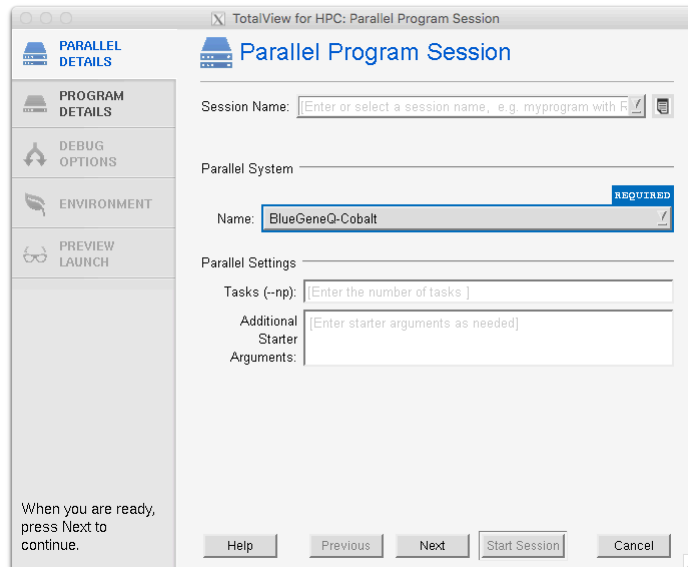
The ‘classic’ method

- `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 of a C program named `cpi.c`. The code includes MPI initialization, a loop for calculating pi, and a search for a prime number. The `main` function is highlighted. The `Processes & Threads` panel on the right shows a table of processes and threads. The `Call Stack` panel shows the `main` function. The `Command Line` panel shows the command `mpiexec -np 4 ./cpi`. The `Logger` panel shows a list of messages, including thread exits and breakpoints. The `What do you want to look for?` panel shows a search for the variable `h`. The `Data View` panel shows the values of variables `x`, `y`, and `myfield`.

Description	RP	#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	do...	3.14159265358979
myfield.value	do...	3.14159265358979

# 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.
- Tvscript and memscript 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

