Debugging Complex Programs with Etnus TotalView

Mary Kay Bunde, *Etnus*

ABSTRACT: *The Etnus TotalView debugger is a state-of-the-art, powerful debugger uniquely suited for debugging complex and parallel codes on major UNIX platforms and Linux. This papers describes some of the unique features of TotalView, and gives somes examples of how TotalView addresses some specific types of debugging problems in common parallel models of today.*

1. Introduction

Etnus TotalView is *the* debugger for complex and parallel code. Having roots in high performance computing, TotalView and its development and support team understand the problems in debugging codes typical of those being written at ECMWF, Los Alamos National Labs, Argonne National Lab, the National Institutes of Health, Chevron, and the majority of other classic HPC organizations. In recent years, Etnus has enjoyed increased visibility within commercial segments where threaded code is troublesome to debug. Customers using TotalView there include: Amazon.com, Nortel, Cisco, State Street Bank, British Telecom, Alcatel, and many more.

In this paper, I will describe some of the challenges of debugging complex code and how TotalView aids in the debugging process. I'll describe specific benefits and features of TotalView, and give a preview of our next release, TotalView 6.0.

2. TotalView Overview and Benefits

TotalView is a flexible and powerful debugger, with both graphical and command line interfaces, that supports debugging of codes written in C/C++, Fortran (77/90/95), and assembly code.

Serial programs will typically fail, when there is a bug, the same way each and every time. Not so with parallel code. Processes may finish in a different order than expected, or processors may not communicate as you intended. TotalView supports debugging of parallel programs, with features built-in specifically to address problems like these and others.

This debugger lets you control your program at all levels – program, process, and thread. Many styles of breakpoints give you the flexibility to start and stop your program in various ways, depending on the debugging situation. For example, a conditional breakpoint might be appropriate if you want to stop your program the 100th trip through a loop. And an evaluation point is perfect for testing a fix to your code without ever touching or recompiling the source.

In addition to this flexibility of control, TotalView has many data management features to help you debug problems with your data. These features are uncommon in other debuggers, and I'll describe them later in this paper

Along with TotalView's powerful features and flexibility comes a unique multiplatform capability. It runs on major UNIX platforms as well as Linux. Table 1 lists the platforms on which TotalView is currently available. For up to the minute information, please visit our Platforms page:
http://www.etnus.com/Products/TotalView/platforms/index.html

Table 1. Supported Platforms

Platform	*Operating System*
SGI	MIPS Irix
Compaq Alpha	Tru64 UNIX
HP	HP-UX
IBM RS/6000 and SP Power	Power AIX
Sun Sparc SunOS 5	Solaris
Compaq Alpha	Linux
Intel x86	X86 Linux

On SGI platforms, TotalView is available on IRIX 6.2 and later operating system levels. Additionally, TotalView supports the following compilers and parallel models on SGI:

- MIPSpro 7.2.1, 7.3, and 7.3.1.2 C/C++ and Fortran compilers
- KAI C++ 3.4
- SGI OpenMP Fortran and OpenMP C++
- KAI Guide 3.8 and 3.9
- SGI MPT 1.2, 1.3, 1.4
- MPICH 1.1.2, 1.2.0, 1.2.1
- ORNL PVM 3.4.1

3. General TotalView Features

TotalView has many features that simplify debugging, whether you're debugging a parallel code or a complex serial code. This section describes some of those features.

3.1 Breakpoints

TotalView has several types of breakpoints:

- Simple breakpoints, which every debugger has
- Conditional breakpoints let you specify a simple expression to evaluate before TotalView decides whether to stop the program or not. For example, you may wish to stop the program at the 100th trip through a loop. This expression would be written

  ```
  if (i==100);$stop
  ```

  $stop is one of many TotalView built-in intrinsics you can use to instruct TotalView within evaluation statements.
- Evaluation points let you write code fragments that are compiled and executed on the fly at the designated line number. The most common use for eval points is to test a simple fix to your program without ever touching the source and recompiling.
- Barrier points, which will be discussed in section 4.1
- Data watchpoints tell TotalView to stop the program if the value of a local variable changes. TotalView supports both conditional and unconditional watchpoints. A powerful use of conditional watchpoints is to test thresholds to ensure they haven't been violated.

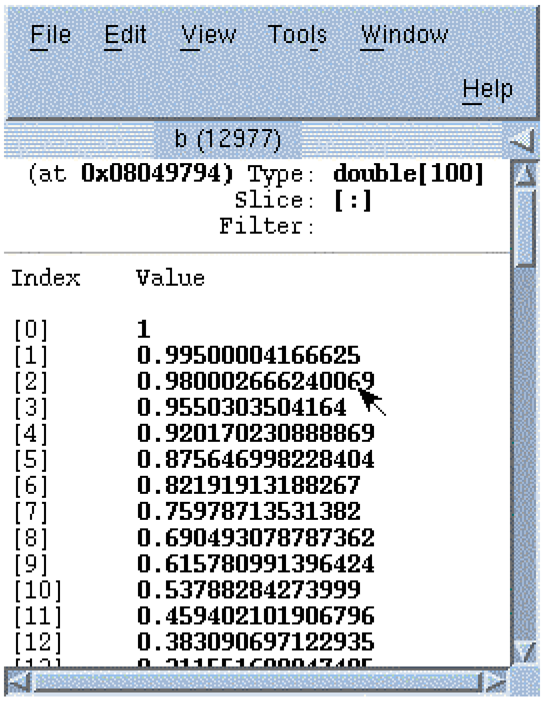
3.2 Dive

Perhaps the best-loved feature of TotalView, *dive* is a powerful tool for obtaining more information about any object. Simply double-left-click on anything, and TotalView will display more information about it if it can. For example, if you dive on a variable, TotalView will show you the value of that variable. If you dive on a function, TotalView will show you the source code for that function.

3.3 Data Analysis Features

TotalView has features to analyse data within your code. Many of these features cannot be found in other debuggers.

- Dive – already described
- Slicing – lets you view a subsection of your array
- Filtering – lets you view only the elements satisfied by an expression you write. For example ">100" will show all the array elements whose values are greater than 100. Figures 1 - 3 shows a progression of views of a 100-element array. This array, as you see, has been sliced and filtered.

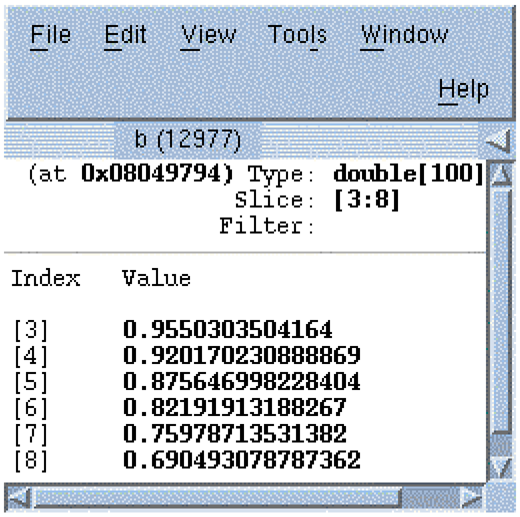


```
b (12977)
(at 0x08049794) Type:  double[100]
                Slice: [:]
                Filter:

Index     Value

[0]       1
[1]       0.99500004166625
[2]       0.980002666240069
[3]       0.9550303504164
[4]       0.920170230888869
[5]       0.875646998228404
[6]       0.82191913188267
[7]       0.75978713531382
[8]       0.690493078787362
[9]       0.615780991396424
[10]      0.53788284273999
[11]      0.459402101906796
[12]      0.383090697122935
```

Figure 1. A 100-element array

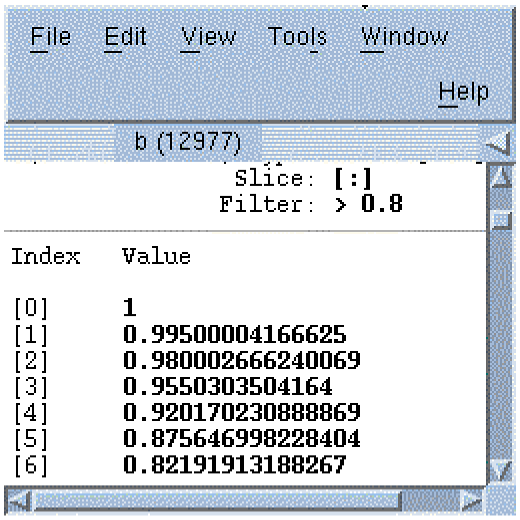


```
(at 0x08049794) Type: double[100]
                Slice: [3:8]
               Filter:

Index   Value

[3]     0.9550303504164
[4]     0.920170230888869
[5]     0.875646998228404
[6]     0.82191913188267
[7]     0.75978713531382
[8]     0.690493078787362
```

Figure 2. 100-element array, sliced



```
                Slice: [:]
               Filter: > 0.8

Index   Value

[0]     1
[1]     0.99500004166625
[2]     0.980002666240069
[3]     0.9550303504164
[4]     0.920170230888869
[5]     0.875646998228404
[6]     0.82191913188267
```

Figure 3. A 100-element array, filtered

- ♦ Statistics – provides important information about your array. For example, min, max, mean values, checksum, the number of NANs and the number of INFs (in the case of floating point numbers).
- ♦ Visualization – TotalView lets you visualize data. You can either set a breakpoint to view a snapshot in time, or you can write an evaluation point and use the built-in intrinsic $visualize. This is handy for watching your data change. For example you could add this evaluation point to a loop that is operating on an array. Then each trip through the loop the visualization would be updated, thereby creating an animation of your data as the program executes. It's a great way to see anomalies in your data. Figure 4 shows a 2-dimmensional array that's been visualized.

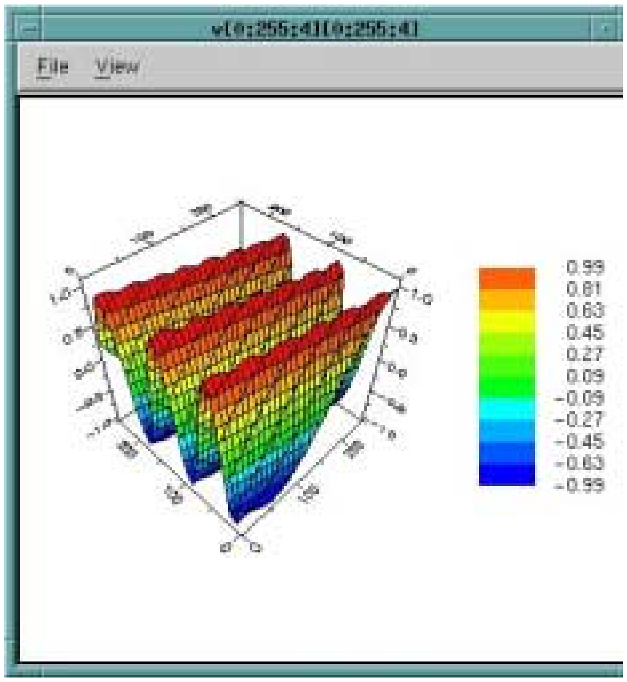


Figure 4. Visualization of 2-dimmensional array

- ♦ Data watchpoints – described in the previous section.
- ♦ Sorting – lets you sort the data you are viewing in either an ascending or descending fashion. It does not change the contents of memory.

All the data analysis features can be combined to give you powerful tools for identifying problems in your data. For programs with gigabytes of data, these analysis features can be powerful time savers.

3.4 Type Mapping

Oftentimes, compilers do not expose enough information about data types for any debugger to display them properly. Type mapping lets you tell TotalView how to display types of data that otherwise generally confuse debuggers. User defined and non-native types are examples of such objects.

Within the command line interface (CLI) you can write a prototype definition that describes how you want data of the specific type to appear. Then every time TotalView displays data of that type (in either GUI or CLI interface), it will display as you've defined. It is rather like turning trash into treasure!

4. Debugging Parallel Codes

While it is impossible to describe all the problems that can go wrong with parallel code, there exist some general issues with each common model for which TotalView is especially suited to help.

4.1 MPI Programs and TotalView

Common issues with MPI programs include

- ♦ Deadlocks (or worse, wrong answers) due to inter-process communication problems
- ♦ Processors completing out of order
- ♦ Misusing local and global variables

Inter-process communication problems are especially difficult to debug because tracing the path of communication can be difficult to do by hand, or even with the help of print statements. TotalView offers two tools to provide insight into how your processes are communicating.

The Message Queue Window is a textual dump of the state of your MPI communicators. Here you can assemble the puzzle of pending sends, receives, and unexpected messages in your program. Further, you can dive on many of the fields in this window to obtain further information or to open an appropriate Process Window.

TotalView can also visualize the Message Queues of your program at a particular instant (for example when you're deadlocked), perhaps providing insight even faster than examining the textual queues. Figure 5 shows the visualization of the program's inter-process communication, and Figure 6 depicts the contents of the message queues via the Message State Window.

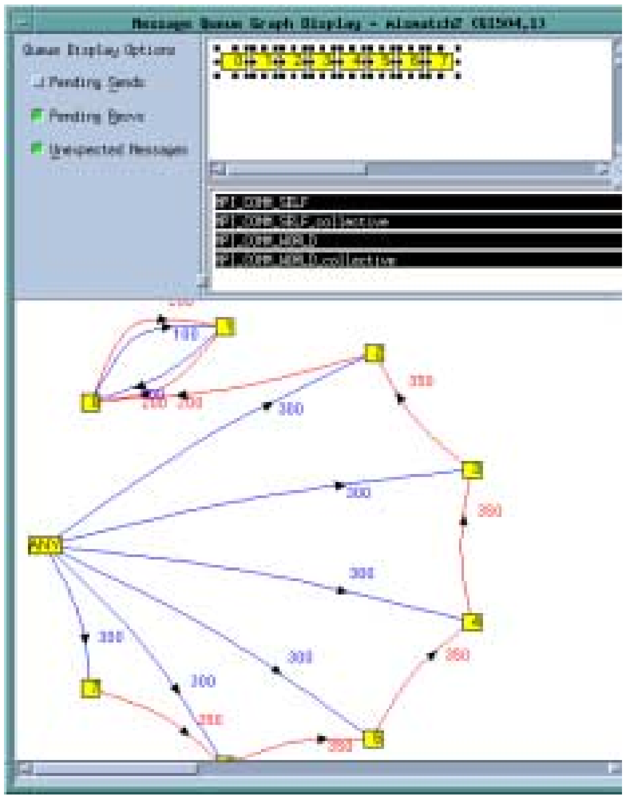
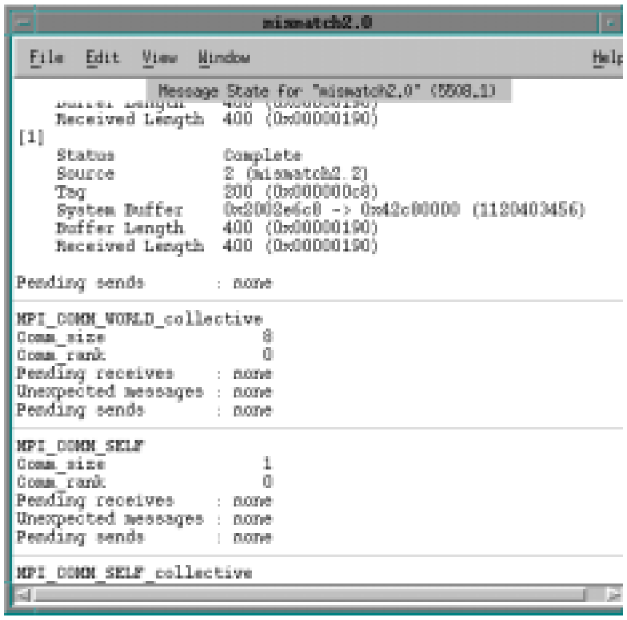


Figure 5. Message Queue Graph



```
    Received Length  400 (0x00000190)
[1]
    Status           Complete
    Source           2 (mismatch2.2)
    Tag              200 (0x000000c8)
    System Buffer    0x2002e6c8 -> 0x42c00000 (1120403456)
    Buffer Length    400 (0x00000190)
    Received Length  400 (0x00000190)

Pending sends        : none

MPI_COMM_WORLD_collective
Comm_size                8
Comm_rank                0
Pending receives     : none
Unexpected messages  : none
Pending sends        : none

MPI_COMM_SELF
Comm_size                1
Comm_rank                0
Pending receives     : none
Unexpected messages  : none
Pending sends        : none

MPI_COMM_SELF_collective
```

Figure 6. Message State Window

In the visualization in Figure 5, the numbers within the boxes indicate a process's rank. The numbers next to the arrows indicate the number of messages when TotalView created the graph. Diving on an arrow displays the Message State Window to provide detailed information about those messages. Diving on a box opens a Process Window for that process. Here are some hints as you peruse a Message Queue Graph:

- ♦ Pending messages often indicate that a process cannot keep pace with the amount of work it is expected to perform. These messages indicate places where you may be able to improve your program's efficiency.
- ♦ Unexpected messages can indicate that something is wrong with your program because the receiving process does not know how to process the message.
- ♦ After a while, the shape of the graph tends to tell you something about how your program is executing. If something does not look right, you might want to determine why it looks different than expected.

When debugging MPI programs, along with any other parallel program, it is often useful to start and stop individual processes or threads, or to carefully observe how they execute with respect to each other. TotalView's barrier breakpoint, and the ability to control the offending program at the process level become invaluable tools for problems in MPI or other parallel codes.

The barrier breakpoint lets you set a barrier at a particular place in your program. When the first process reaches the barrier, all processes will stop (this is configurable). At this point, you can examine the state of your program, and perhaps choose to continue. If you choose to continue, those processes having arrived at the barrier will remain there, while the others execute. Of course, the program will stop again when the next process reaches the barrier.

There may be times when you wish to leave a subset, or perhaps all but one process, in their current state and continue execution with just one process (or a subset of processes). TotalView easily lets you change your control group to be that of a subset of processes, or a single process. Then, any command you give to TotalView will only operate on that process (or subset). This allows you to drill down or fine-tune your debugging session as you learn more about why your program is failing.

Misuse of global and local variables in parallel programs is common. Using the drill down approach to isolating bugs may help you isolate misuse of local and global variables as well. You will find it also useful to see the value taken by every instance of an MPI variable – that is, the value across each and every process in the program. TotalView offers a *laminated* view of data for this purpose. Simply dive on an object, and choose **Laminate** > **Process** from the **View** menu. You may see problems with data as you view their contents side by side or watch them change as your program executes.

4.2 OpenMP Programs and TotalView

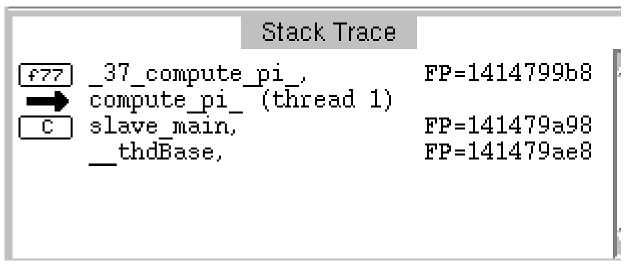
OpenMP programs pose unique problems in that the compiler does the work to parallelize the code. In order to exploit parallelism in parallel sections of OpenMP code, the OpenMP compiler must generate code for the parallel region that can run concurrently in many threads, while still accessing the shared variables. Most OpenMP compilers accomplish this by creating what is commonly called an *outlined* routine. The outlined routine is called by the OpenMP runtime, which assigns the outlined routines to specific threads for execution.

Since the OpenMP compiler performs the parallelization of the program, and the OpenMP runtime assigns each outlined routine to a thread for execution, the user need not know what is exactly going on with the program. However the debugger must understand it in order to help the user find bugs.

Common issues with OpenMP programs include
- ♦ Debugging within parallel regions
- ♦ Understanding how the program arrived at its current execution point
- ♦ Accessing private, shared, and thread-private data

TotalView lets you debug code within parallel regions. To arrive in a parallel region, however, you must set a breakpoint within the region, and run to that point. TotalView does not allow you to step into the outlined routine. Setting the breakpoint is the only safe way to ensure you arrive at the place you are expecting.

Upon arriving in a parallel region, you may be able to control your program at the thread group (all the like threads) level or at the single thread level. Once in a parallel region, you might also like to know how, exactly, execution arrived at this point. TotalView's *stack parent token* is an arrow in the stack frame of the Process Window (the stack frame is in the upper left corner of the Process Window) that points to the parent routine that spawned this worker thread. Figure 7 shows a close up of the stack frame and stack parent token. Diving on the stack parent token will change TotalView's context, and your view, to that parent.



```
                  Stack Trace
f77   _37_compute_pi_,            FP=1414799b8
 ➡    compute_pi_ (thread 1)
 C    slave_main,                 FP=141479a98
      __thdBase,                  FP=141479ae8
```

Figure 7. OpenMP stack parent token

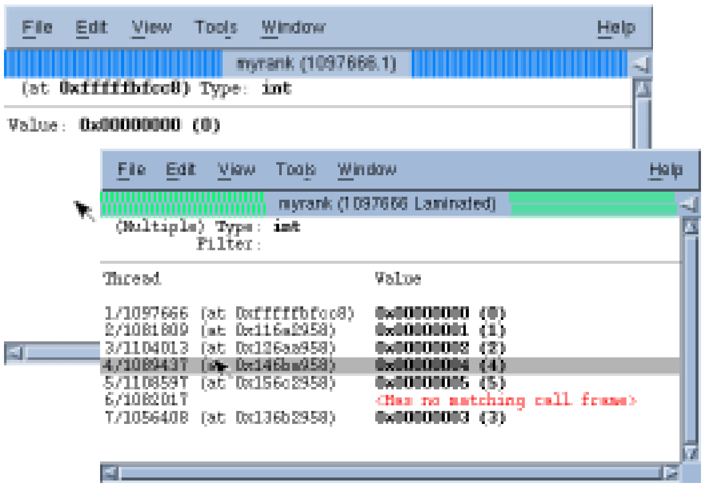
Other interesting things to note when viewing the context of a worker thread include:
- ♦ In the lower left corner of the Process Window, the *threads pane*, you can see the number of threads in the program. On some operating systems, you will

be able to distinguish between operating system threads and user threads.
- ♦ You can see the name(s) the compiler has generated for each of the threads and on which thread you are currently focussed.

Displaying the values of OpenMP local private variables in parallel constructs poses no problems for TotalView. This capability is already built-in. When you dive on any data object, TotalView displays the value(s) of that object.

When debugging OpenMP code, it is often useful to see the value taken by every instance of a private variable – that is, the value across each and every thread in the program. TotalView offers the *laminated* view of data for this purpose. Simply dive on an object, and choose **Laminate > Thread** from the **View** menu. Figure 8 shows an example of a dive (background picture) and its laminated counterpart (foreground).

Accessing thread-private data is a bit more complex, and is compiler and operating system dependent. Even so, TotalView does allow you to access thread-private data, in a similar way to showing other data.



```
(at 0xffffbfcc8) Type: int

Value: 0x00000000 (0)
```

```
(Multiple) Type: int
         Filter:

Thread                              Value

1/1097666 (at 0xffffbfcc8)          0x00000000 (0)
2/1081809 (at 0x116a2958)           0x00000001 (1)
3/1104013 (at 0x126aa958)           0x00000002 (2)
4/1089437 (at 0x146ba958)           0x00000004 (4)
5/1108597 (at 0x156c2958)           0x00000005 (5)
6/1082017                           <Has no matching call frame>
7/1056408 (at 0x136b2958)           0x00000003 (3)
```

Figure 8. OpenMP thread object, laminated view

4.3 Threaded Programs and TotalView

Common issues with threaded programs include
- ♦ Understanding the execution order of the program
- ♦ Synchronization, locks, and condition variables

Multithreaded codes can be a nightmare to debug. Not only must you rely on the scheduler for scheduling threads, but each operating system has a different thread-scheduling model, so your program could execute differently on different platforms. It is difficult to know how your program is actually executing.

The first aid TotalView has is to automatically group your threads according to the task they are doing. Like threads will be in a group. If you have a hint that a thread designed to do a very specific piece of work is in error, you can look at all the threads designed to do that piece of work, and ignore others.

TotalView lets you regroup threads into groups of your own choosing, through the Process Thread Setting (P/T sets) mechanism. You can then issue commands, such as step and next, or set breakpoints to just a single group, the whole program, or (on some operating systems) just a single thread.

TotalView also does its best to distinguish between threads that you care about and all other threads – even those spawned by the operating system on behalf of the program you are debugging. In doing so, TotalView can hide those extraneous threads from your view, such that you never have to worry about them while debugging.

Synchronization and locking contention can pose problems in threaded code. TotalView exposes the operating system locks, mutexes, and condition variables so that as you step through your code you can get a better understanding of how your program is using, asking for, and waiting on, resources.

As discussed for the other parallel models, barrier points are also useful tools for controlling threaded code.

5.0 TotalView 6.0 – A Preview

TotalView 6.0 will be released later this year. Here is a brief look at the contents of the release:

New hardware and operating system support:
- ♦ Sun 64-bit, Solaris 7, 8, 9
- ♦ IBM Power 4 and Regatta systems, AIX 5.1
- ♦ Scyld systems

New Compilers:
- ♦ Sun Forte 6, update 2. Forte 7
- ♦ Intel Linux compilers
- ♦ Lahey F90/F95 version 6.1
- ♦ GCC 3.X
- ♦ KAI 4.0
- ♦ UPC (on Compaq first)

TotalView Features
- ♦ Improved C++ support: proper scoping, C++ namespaces, ability to write expressions in C++, improved handling of C++ templates

- Memory utilization statistics, including heap, data, and text sizes, stack size of the main thread, overall memory footprint size (including mappings), and a sort by column feature.
- Improved performance on SGI and IBM for large codes running under TotalView's control
- The addition of a P/T-Set browser in the GUI (formerly P/T sets were managed in the CLI only)
- Ability to evaluate expressions across processes, called *parallel evaluation of expressions.*
- More control over the way in which data is displayed. You choose the type, the presentation (scientific, fixed, decimal, etc), and the format (width, precision, justification, etc).
- *Dive in all* – transforms one field member from an array of structures to an array of the field member's type. OR transforms an array of pointers to an array of objects. The result can then be sliced, sorted, filtered, etc…
- Display C++ pointer variables as arrays, which can then be sorted, sliced, filtered, etc…
- Attach to a subset of processes

Conclusion

Etnus TotalView is a robust, stable, and powerful debugger. Its advanced features ease the debugging process of complex and parallel code. Etnus offers free two-week, fully functional, evaluations of TotalView from the website at www.etnus.com/Download/demo-tv.html.

About the Author

Mary Kay Bunde is the Director of Market Development at Etnus. She previously worked at SGI and Cray, as Engineering Manager of the Tools group. She can be reached at Etnus, 24 Prime Parkway, Natick, MA 01760. Email: mkay@etnus.com.