Automation Assisted Debugging on the Cray with TotalView

Chris Gottbrath, *Rogue Wave Software*

ABSTRACT: *A little bit of automation can go a long way towards streamlining and simplifying the process of debugging scientific applications. This talk will demonstrate using a new TotalView feature, C++View, to transform complex data structures and automatically perform validity checks within them. C++View is an element of TotalView's extensive scripting framework, which also includes a type transformation facility, a fully programmable TCL-based CLI, a C and Fortran expression evaluation system, and the scripting tools MemScript and TVScript.*

KEYWORDS: XE6, Parallelism, Development Tools, Debugging, Automation

1. Introduction

Cray provides a uniquely powerful and flexible set of environments for developing high performance applications with the XT5, XT5m, XE6 and XE6m. These machines provide scientists and developers with a unique combination of familiarity and performance. Each node uses an x86 processor and runs a version of Linux. Individual nodes are connected with a low-latency interconnect. The entire package has been tuned first to provide high performance on scientific applications; both the hardware and software provided are designed to allow the system to be managed relatively easily. Cray systems hold out the promise of being able to accomplish modeling that is staggeringly computationally intensive. Cray systems can compute physical processes that require fine detail and the large scale processes that form the context for those details. Cray systems can support models that begin to introduce not just one but numerous physical processes with different time scales and numerical behaviors.

1.1 Developing Parallel Software

Nonetheless developing parallel software continues to be a daunting task, one that needs to be taken in stages. Scientists and developers typically start with an algorithm that is initially developed in a serial context and then analyze it to find places where the data can be distributed and the calculation done simultaneously across multiple separate processes that can be hosted on different nodes of the cluster. While conceptually simple this process of distributing the work adds significantly to the complexity of the application. The data structures, and sometimes the algorithm, have to be re-factored in ways that often obscure the physical meaning of the data. Distributing the data and computations invariably introduces numerous new concepts such as domain decomposition, halo regions and message passing to the application; all of which may have little to do with the fundamental science and more to do with simply keeping track of the data. Completely new issues arise that are non-existent in serial programming, such as load balancing and optimizing inter-node communication patterns.

This process will typically only get an application to the point where the data can be spread across dozens or hundreds of separate processes before some factor will limit the scaling, so that further dividing the work either no longer increases or actually degrades the overall application performance. If the science goals call for a higher level of performance the developer frequently faces a choice: either introduce multi-thread parallelism, find another way to divide the data, or overcome the

factor that is currently limiting the scaling of their application.

This decision is hardly a simple one. Threading introduces a paradigm with very different rules and behavior, but since recent Cray systems can have 24 cores per node, there is quite a bit to be gained by threading. Scaling up an application requires sophisticated analysis to identify the factor cutting off the scalability. Overcoming the limitation may require significant additional data and code re-factoring.

The end result of this iterative process is an application in which the original domain-specific code is dwarfed by complex and arcane "work-load management" and "parallel plumbing" routines that efficiently distribute the work and transfer data so that each component task is able to spend the majority of its time computing without having to wait for data to arrive. Some estimates place the ratio of "framework" code to "model specific" code at 4:1.

These applications are understandably hard to develop and to maintain.

1.2 Improving Developer Productivity

What can we do to help make scientists and developers engaged in such work more productive? What can we do that might allow development of higher quality applications that generate results that are more reliable?

TotalView provides a graphical environment in which developers can examine an application, focusing on a single process or a single thread, or synchronizing groups of processes and threads and looking at the behavior of the parallel application as a whole. Developers can look at variables in a single thread or process and easily ask the debugger to show them the value of corresponding variables on other threads and other processes. More detail on TotalView is available on the web[i] and in previous CUG papers[ii,iii,iv].

Still, even with this graphical interface, parallel programming and parallel debugging can be daunting and overwhelming. In particular, the data structure refactoring involved in building "parallel plumbing" can obfuscate concepts and values that would otherwise be familiar to a scientist troubleshooting the application. Several things can be done to simplify this.

At the language level some of the frequently used constructs can be made part of the language. This has been done with thread parallelism in the form of OpenMP and with process parallelism with UPC. Both of these serve to reduce the volume of overhead code and represent data in a way that more closely matches the way the user thinks of an algorithm. The TotalView debugger works with both of these extensions and in the case of UPC provides a unified view of the shared array objects, gathering individual elements from across the clusters and showing both their global indices and originating process.

While inventing a new parallel language is always temping the reality is that most scientific applications are written in traditional languages such as Fortran and C or C++. What can be done to simplify programming and debugging for scientists working with more traditional technologies like C++ and MPI? One possibility is to reduce the time spent sifting through complex data structures to find the data relevant for troubleshooting their problems. This can improve scientific productivity because it allows scientists to focus their effort.

1.3 Automation-Assisted Debugging

Automation can make debugging more flexible and powerful, with a tool that can be customized to handle tedious operations. Two new TotalView automation features, C++View and TVScript, are outlined in this paper.

C++View enables better interactive debugging. It automates steps for examining and validating data structures during interactive graphical debugging. By eliminating tedious and perhaps error-prone operations it allows scientists to focus on the part of the program that is misbehaving. C++View makes it trivial to work with the dynamic and extensible data structures used in modern C++ programming. While the actual implementation of the data structure may involve multiple layers of indirection, users can instruct the debugger as to how they want to see the data contained within.

Even better, scientists frequently work with systems in which there are aggregate characteristics, such as the energy and momentum in some part of the system, which should evolve in predictable ways as the model runs. Deviations in these aggregate characteristics can be critical indicators of what is happening. C++View can be used to automatically trigger diagnostic calculations and present the results. Cray systems are generally resource-managed through a batch queuing system. In such a context debugging time is a critical factor. Automation provided by C++View enables scientists to progress much more quickly and confidently through test runs before an

interactive session times out and is terminated by the batch queue environment.

The second feature, TVScript, enables scientists to do unattended debugging within batch queue environments. Cray systems are frequently configured with multiple queues. Batch queues generally work most efficiently with jobs that don't require direct user interaction. It is not uncommon for sites to establish policies that limit interactive access to just a small fraction of the system resources. Such policies can become a challenge for scientists who need to troubleshoot errors that occur at scales larger than what can run within these interactive queues. In such cases non-interactive batch debugging can be an effective alternative.

Large scale jobs also manifest a variety of other characteristics that recommend an automated approach to debugging. First, they may be very large; the sheer volume of data to be explored may be prohibitive, warranting an offline analysis strategy. Scientists frequently separate the recording of data from its analysis, and data gathered in such sessions can be used, for example, to narrow and refine a debugging hypothesis that can then guide a more focused interactive debugging session. Large scale jobs may run into problems after an unknown, or long runtime. Scripted debugging frees the scientist from tediously watching jobs that may or may not fail. When the program does fail the script can gather data that either points to the error or serves to narrow down the parameters to set up for a more focused interactive debugging session.

These two features provide scientists with extremely powerful interactive and non-interactive debugging options that simplify debugging moderate and large scale applications on Cray systems.

2. C++View

2.1 C++View Overview

C++View is based on the premise that a user may want to specify what is to be displayed by the debugger for a program data object of a specific type. TotalView provides an optional function that the user can write to define what to print when the object is examined by the debugger.

2.2 C++View Interface

This function has the declaration

```
int TV_ttf_display_type (const T *)
```

where T is the type of the object that the user wants to have transformed. If such a function is present in the target program TotalView will call it whenever it would normally display an object of that type. The object itself is passed into the function through the pointer argument and the function uses a return value to indicate either success or failure. Debuggers are generally used on programs that are broken and a display function might return something other than success if, for example, the pointer that gets passed in doesn't appear to point to valid data.

Within this function the scientist can examine the object pointed to, checking its validity, and ask the debugger to display elements of the object, other variables and values drawn from the program, or data derived on the fly either from the object or from the program more generally. Each element that the scientist wishes to display is presented to the debugger by calling the function provided by the debugger:

```
int TV_ttf_add_row(
   const char * field_name,
   const char * type_name,
   const char * address )
```

This function tells the debugger about one element that will be part of the set of information displayed for the object in question. The function takes three simple arguments. The first argument, `field_name,` is just a string that TotalView will display as a label for this element. The second argument defines the element type, critical to the function, since TotalView's powerful display capabilities are based on object type. The third field establishes the address of the element. The return value of this function is used to communicate status information. TotalView will return `TV_ttf_ec_ok` if everything is successful, otherwise it will return a code indicating an error.

These two simple functions give scientists the ability to streamline what might otherwise be tedious parts of the debugging process. Let's look at several transformations.

2.3 Simplest C++View Example

The simplest possible transformation looks like this

```
struct S
{
  int X;
};

int TV_ttf_display_type(
        const struct S * item)
{
  return TV_ttf_add_row(
           "X from struct S",
           "int",
           &item->X)
}

main()
{
struct S myS;
}
```

If you then examine myS in TotalView you will see that it labels the field "X from struct S", reports that it has type int and shows you the value of the X field of struct S.

Please note that for readability this code sample as well as those below are pseudo-code. Please consult the product documentation and the examples that come with TotalView for compileable examples. Contact the author or Rogue Wave Tech Support, tvsupport@roguewave.com, for help using the product.

2.4 Using C++View to Simplify Structures

Things get more interesting if you choose to highlight only the most interesting or relevant information. C++View can be used to show some of the fields while omitting others.

```
struct person
{
            int age;
            char * name;
            char * title;
            float salary;
};



int TV_ttf_display_type(
      const struct person * item)
{
   TV_ttf_add_row(
          "Name",
          "$string",
          *(&item->name));
   TV_ttf_add_row(
          "Age",
          "int",
          &item->age);
   /*error handling logic omitted*/
}

main()
{
struct person myPerson;
}
```

When you dive on myPerson you will see only the two fields that you had selected through C++View. In the CLI if you **dprint** myPerson it might look like:

```
> dprint myPerson
myPerson = {
   Name = "sample name"
   Age = 23
}
```

C++View can be toggled on and off with a debugger preference. Use it, for example, ir you usually want to see just this abbreviated view but occasionally need to see the full list of elements that make up a specific person structure.

2.5 Using C++View with Dynamic Objects

C++View can also be used to simplify the display of dynamic structures like linked lists. Walking a null terminated linked list might be done with something like:

```
int TV_ttf_display_type(
       const struct ll * item)
{
   ll * current=item;
   while (current->next != 0 )
   {
      TV_ttf_add_row(
        "member",
        "int",
        &current->value);
      current = current->next;
   }
   TV_ttf_add_row(
      "final member",
      "int",
      &current->value);
   /* error handling logic omitted */
}
```

2.6 Presenting Supplemental Data Using C++View

So far each of these transformations has pointed to data that is present in the underlying data structure. C++View can also present data that isn't present there, which is useful for presenting the results of validation as well as various kinds of diagnostic data. For example, in working with vectors if you generally first compute a vector's length, automate that with a C++View transform along the lines of the following:

```
int TV_ttf_display_type (
      const struct vector * item)
{
   float length=sqrt(
           item->x*item->x
               +
           item->y*item->y);
   TV_ttf_add_row(
         "X component",
         "float",
         &item->x);
   TV_ttf_add_row(
         "Y component",
         "float",
         &item->y);
   TV_ttf_add_row (
         "Length",
         "float",
         &length);
   /* error handling logic omitted */
}
```

2.7 Templates and C++View

I've shown simple structures but C++View also works with templated C++ code; transformations can be polymorphic along with the classes they transform.

2.8 Fortran, C, and C++View

C++View can be used with Fortran or C but due to the limitations of the way function overloading works you are limited to just a single TV_ttf_display_type function per object file. See the TotalView documentation or contact support for help on using C++View with C or Fortran.

2.9 Composition and Elision with C++View

C++View supports type composition, so if you have objects and types that are built up of collections of other objects you can easily write transformations for just those objects you want to transform. Each object will be transformed if there is a corresponding function to do the transformation. In some cases this might result in cumbersome data displays so C++View includes a streamlined display mechanism called "elision". Your `TV_ttf_display_type` function indicates via a return type whether the data it is providing can be elided. If it can be elided then if it is part of an array or other aggregate then TotalView will display just the data and not the structure type.

For example, if type1 doesn't support elision and type2 does, then dprinting arrays of type1 and type2 will look like:

```
> dprint type1_array_instance
   type1_array_instance = {
      [0] = {
            X = 0x0001 (1)
        }
        [1] = {
            X = 0x0002 (2)
        }
   }

> dprint type2_instance
   type2_array_instance = {
      [0] = 0x0001 (1)
      [1] = 0x0002 (2)
   }
```

Elision provides a way to work more easily with simple arrays of transformed objects.

2.10 Important Tips for Using C++View

First, avoid type recursion. Transformed objects cannot reference themselves either directly or indirectly. If class A has a member of type B items and type B items include members that are of type class A, that will cause problems for TotalView. Pointers between types A and B are ok because TotalView doesn't automatically display the details of something that is merely pointed to.

Although I have omitted error handling in the examples above for clarity, it is important. You may want to validate that the object you are transforming is in a consistent state. In particular, avoid following pointers that might point to unallocated memory as this can cause your target program to crash. If your display function can't validate that the data it is operating on is correct, you can simply return `TV_ttf_format_raw` instead of `TV_ttf_format_ok` to indicate that the debugger

should not format this variable but simply display it as it would with no transform.

Be sure your `TV_ttf_display_type` function doesn't change the target being debugged. It should not increment counters, allocate memory without freeing it, or overwrite anything within the part of the program you are debugging. Each time it needs to refresh the window, TotalView will call `TV_ttf_display_type`. This might be more frequently than you expect.

If you need some kind of persistent storage, for example to display previous values as part of the transformed data structure, we recommend creating a global buffer variable for storage.

There is really no limit to what can be performed within a C++View. Jeff Keasler, a scientist at LLNL, has already used it to prototype really interesting new functionality in support of comparative debugging. In this prototype the user runs two copies of TotalView on two copies of the program to be debugged, Copy A and Copy B. C++View uses network communication in the display_type routines to make data from Copy A available when debugging Copy B. Then the functions highlight any differences between Copy A and Copy B at the data level.

2.12 Using C++View in your program

C++View is distributed in the form of a c++ source and header file pair called tv_data_display.c and tv_data_display.h. This provides a function declaration for TV_ttf_display_type(), various enum values for return types, and a stub function for TV_ttf_add_row() which wil be intercepted when TotalView is used on the target program. These two files are licensed in such a way that you can freely embed and distribute them.

See page 249 of the TotalView Reference Guide for detailed instructions examples on compiling your program with C++View.

2.12 TCL TTF, a C++View Alternative

C++View is one of two type transformation mechanisms provided by TotalView, the other being TCL TTF (TCL Type Transformation Facility). Describing TCL TTF in detail is beyond the scope of this article (information is located in the TotalView Reference Guide) but an important trade off is worth discussing.

TCL TTF differs from C++View in that when users define TCL TTF transformations they are defining a series of steps that the debugger goes through to walk the data structure to be transformed and obtain each of the elements to be displayed. It is generally more challenging to write transformations using TCL TTF because it involves a detailed understanding of not just the data structure but also low-level details like how the elements of the data structure are laid out in memory and how the debugger would access them.

TCL TTF transformations don't involve calling any functions in the target process itself. Since TCL TTF transformations just involve the debugger reading values from memory they can be used with corefiles. C+++View won't work with corefiles because it involves calling functions within a live process.

In short C++View is easier to write while TCL TTF works with corefiles.

3. TVScript

3.1 TVScript Overview

TVScript is a framework for easily doing non-interactive debugging with TotalView. TVScript is conceptually very straightforward. You define a program you want to debug and a series of events that may occur within that target program. TVScript loads the program you want to debug under its own control so that it can start it and stop it as needed. TVScript may or may not set one or more breakpoints within the program. Then TVScript runs the program. Each time the program stops TVScript compares its state with the list of established breakpoints and for each breakpoint (or other reason for stopping) TVScript can perform different operations such as gathering data or running specific reports. It logs output to a set of files and continues the program until it exits.

Scientists will typically want to submit jobs that use TVScript to the Cray as non-interactive batch queue jobs. These jobs will run without any kind of user interaction and the resulting logfiles can be reviewed to see the behavior of the parallel job on the Cray.

3.2 TVScript Syntax

TVScript is run as a UNIX command line utility with the following general syntax:

```
tvscript <tvscript-options> \
   <program-to-debug> \
    -a <args-for-program-to-debug>
```

If you want to debug an MPI job that you would normally start with

```
aprun -n 64 my_parallel_application
```

you might start an interactive job with TotalView using a command like

```
totalview -mpi aprun \
   -np 64 my_parallel_application
```

to start up TotalView, load my_parallel_application and then allow you to set breakpoints, etc. Then when you hit "go" it will use aprun to launch your parallel application with 64 processes. If that is the case then the basic command for using TV script with that application looks like:

```
tvscript -mpi aprun \
   -np 64 my_parallel_application
```

to which you will add command line options to define events that you want to track and actions you want to have happen when those events occur.

At any point you can call tvscript with no arguments and it will respond with usage guidelines including a brief listing of events and actions that you can specify.

3.3 TVScript Crash Reporting

The general command option you will use for looking at errors is:

```
-event_action “event=>action1,action2"
```

The simplest thing that you might want to do with TVScript is to get improved crash reporting. You can do this by specifying "error" and then you might want to perform the action "display_backtrace". So when the program encounters any kind of error TVScript knows that you want to look at a backtrace from the program. That might look like:

```
tvscript -mpi aprun -np 64 \
-event_action \
"error=>display_backtrace" \
my_parallel_application
```

As specified here the backtrace will simply be a list of functions, not too different than you might get from other backtrace mechanisms. However TotalView can gather much more information. You might want, for example, to look at function call arguments and/or local variables at the point where the program encounters an error. This can be done with the qualifiers

```
-show_locals -show_arguments
```

which would look like:

```
tvscript -mpi aprun \
-np 64 \
-event_action \
"error=>display_backtrace  -show_locals
-show_arguments" \
my_parallel_application
```

That will generate a much more verbose and potentially helpful logfile when the program you are debugging encounters some kind of error.

Figure 1
The output of

```
display_backtrace –show arguments
```

looks like Figure 1. Note the header information that is presented and the fact that both the function main() and my_create_port() each have two arguments; all those arguments are included in full in the listing.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Backtrace
!
! Process:
!   ./server (Debugger Process ID:  1, System ID:  17031)
! Thread:
!   Debugger ID:  1.1, System ID:  3084126880
! Time Stamp:
!   06-25-2008 19:50:06
! Triggered from event:
!   actionpoint
! Results:
!   >  0 my_create_port   PC=0x080488f3, FP=0xbfdac718 [/tvscript/server.c#40] [/tvscript/server]Function "my_create_port" arguments:
!      0.1  port_number = 0x0000d585 (54661)
!      0.2  ipv4_addr_string = 0x0804947a -> '1'   (0x31, or 49)
!
!      1 main          PC=0x08049106, FP=0xbfdac748 [/tvscript/server.c#216] [/tvscript/server]Function "main" arguments:
!      1.1  argc = 0x00000001 (1)
!      1.2  argv = 0xbfdac7d4 -> 0xbfdada76 -> "./server"
!
!      2 __libc_start_main PC=0xb7d55e9f, FP=0xbfdac7a8 [/lib/tls/i686/cmov/libc.so.6]
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

3.4 TVScript Memory Debugging

TotalView also includes MemoryScape, which . provides heap memory debugging functionality that can be accessed from TVScript. Explaining this is beyond the scope of this paper (though you can see the CUG paper from 2010 for more details). One of the things that you might want to do is check for memory leaks when the program encounters any events or errors or is ready to exit.

```
tvscript -mpi aprun \
-np 64 -memory_debugging \
-event_action \
"any_event=>list_leaks" \
my_parallel_application
```

Other flags can be used to list allocations, export binary memory reports, and perform various kinds of array bounds checks.

3.5 Tracing your Program with TVScript

You can also use TVScript to automate gathering more detailed information from your program. You might, for example, want to trace the way some particular part of the program is behaving over a long running application. You can instruct TVScript to set a breakpoint at a location of interest within the program and provide a backtrace whenever that breakpoint is hit, or display the value of particular variables. For example, in examining a function, func1, if you want to display the value of variable1 each time the program executes func1, do that very simply with something like:

```
tvscript -mpi aprun \
 -np 64 \
-create_actionpoint \
"func1=>print variable1"
my_parallel_application
```

Every time any one of the 64 MPI tasks executes func1 tvscript will record an event whose record will be accompanied by the value of the associated variable1. If variable1 is a structure or an array all the values that make up that structure or array will be printed.

Printing a value in TVScript generates output like that shown in Figure 2. Note that printing a structure results in a display of all the fields in the structure.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Print
!
! Process:
!   ./server (Debugger Process ID:  1, System ID:  12110)
! Thread:
!   Debugger ID:  1.1, System ID:  3083946656
! Time Stamp:
!   06-26-2008 14:04:09
! Triggered from event:
!   actionpoint
! Results:
!   foreign_addr = {
!     sin_family = 0x0002 (2)
!     sin_port = 0x1fb6 (8118)
!     sin_addr = {
!       s_addr = 0x6658a8c0 (1717086400)
!     }
!     sin_zero = ""
!   }
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Figure 1

The actionpoint can be associated with a conditional expression so that it only triggers when conditions are met. The value to be printed can be a simple variable or an expression derived from one or more variables. The expression can include function calls. For example, you might set a breakpoint on a line of code in a loop, configuring the breakpoint to stop only on certain loop index values. Then you might use the expression mechanism to look up elements in various different arrays, perhaps passing those values into a function call and reporting the result of that function call in the log.

3.6 Tips for Using TVScript

TVScript itself is based on a very general TCL scripting mechanism and is provided to users in the form of a user-readable and user-editable TCL script. TVScript can be freely modified and adapted, so if you run into limitations in TVScript you may find that you can examine TVScript itself and find a way to extend it and overcome those limitations.

Finally, it is worth noting that C++View (and TCL TTF) transforms that are available to TVScript (either in the target program for C++View or in the TV configuration files for TCL TTF) will be applied to data that is presented in TVScript.

4. Conclusion

This paper has introduced two extremely powerful automation facilities that are unique to TotalView. C++View and TVScript both allow scientists and developers to streamline operations that might otherwise be very cumbersome and tedious. C++View augments and streamlines the interactive debugging experience and allows developers to customize what data they see when they look at objects and structures that are defined within their programs. C++View can be used to walk complex data structures and automatically perform validation of data structures and data contained within those structures. TVScript makes it possible for scientists to very easily do non-interactive batch debugging of parallel programs on Cray supercomputers. This frees developers from the confines of interactive queues, which are typically rather limited in terms of scale. It can also provide a powerful mechanism for dealing with errors that manifest stochastically and/or only after the program has been running for a fairly long time.

As supercomputers and the programs that run on them continue to scale up, the need for automation in debugging can be expected to increase. These advanced capabilities are part of TotalView now and will serve as a solid foundation for features in the future that will do even more to boost the productivity of scientists and developers.

Acknowledgments

Thanks to Gayle Procopio for aid in preparing this document.

About the Authors

Chris Gottbrath is Principal Product Manager responsible for TotalView, MemoryScape and ReplayEngine at Rogue Wave Software. He's worked with the TotalView debugger for a decade in a range of technical and marketing roles. Prior to that he wrote his fair share of bugs in Linux-based numerical simulations of galaxy dynamics and large scale structure as a graduate student in Tucson, AZ. He has a Masters of Science in Astronomy and Astrophysics from the University of Arizona. He can be reached at 200 Honeysuckle Lane, Starkville, MS 39759 or by email: chris.gottbrath@roguewave.com.

[i] Rogue Wave Software web site, TotalView Product Page. http://www.roguewave.com/products/totalview-family/totalview.aspx

[ii] Gottbrath, Chris. "Reverse Debugging with the TotalView Debugger." *Proc. Cray Users Group, 30, 2008.*

[iii] Gottbrath, Chris. "Debugging Scalable Applications on the Cray XT." *Proc. Cray Users Group, 31, 2009.*

[iv] Gottbrath, Chris. "Improving the Productivity of Scalable Application Development with TotalView." *Proc. Cray Users Group, 32, 2010*