Debugging with DDT

Woo-Sun Yang
NERSC User Services Group

NUG Training 2012
February 1-2, 2012
NERSC Oakland Scientific Facility

Revised on 2/10/2012; Based on ddt/3.1-21354
Why a Debugger?

• It makes it easy to find a bug in your program, by controlling pace of running your program
  – Examine execution flow of your code
  – Check values of variables

• Typical usage scenario
  – Set breakpoints (places where you want your program to stop) and let your program run
  – Or advance one line in source code at a time
  – Check variables when a breakpoint is reached
• Distributed Debugging Tool by Allinea
• Graphical parallel debugger capable of debugging
  – Serial
  – OpenMP
  – MPI
  – CAF
  – UPC
  – CUDA – NERSC doesn’t have a license on Dirac
• Intuitive and simple user interfaces
• Scalable
• Available on Hopper, Franklin and Carver
• Can use it for up to 8192 tasks at NERSC
  – Shared among users and machines
Starting DDT

- Start DDT in an interactive batch session
- Compile the code with the –g flag

```bash
% qsub –I –lmppwidth=24 –q debug –V   # Hopper/Franklin
% qsub –I –lnodes=1:ppn=8 –q debug –V # Carver
...
% cd $PBS_O_WORKDIR
% module load ddt

% ftn –g prog.f            # Hopper/Franklin
% mpif90 –g prog.f         # Carver

% ddt ./a.out
```
Starting DDT (cont’d)

- Click on ‘Run and Debug a Program’ in the Welcome window
- Set in the Run window
  - Program name
  - Programming mode (MPI, OpenMP,…)
  - Number of processes and/or threads
DDT Window

- Action buttons
- Process/thread control
- Process groups

“Project Navigator” for source files

Can have multiple tabs

Source Code

- Process/thread control
- Process groups

“Project Navigator” for source files

- Local Variables for the current stack
- Variables in the current line(s)
- Current Stack

Input/Output to and from program
Breakpoints that you set
Watchpoints that you set
Parallel stacks
Tracepoints you set
Output from the tracepoints
Navigating Through a Program

- **Play/Continue**
  - Run the program

- **Pause**

- **Step Into**
  - Move to the next line of source
  - If the next line is a function call, step to the first line of the function

- **Step Over**
  - Move to the next line of source code
  - If the next line is a function call, step over the function

- **Step Out**
  - To execute the rest of the function and then move to the next line of the calling function

- **Run To Line**
  - Run to the selected line
Process Groups

• Group multiple processes so that actions can be performed on more than one process at a time

• Group ‘All’ by default

<table>
<thead>
<tr>
<th>All</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Create Group

• Can create, edit and delete groups

![Create a New Group]

<table>
<thead>
<tr>
<th>All</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workers</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
| Create Group

• Select group
  – ‘Current Group’ displays the name of the group in focus
  – Source Code Viewer shows the corresponding group color

<table>
<thead>
<tr>
<th>Current Group: All</th>
<th>Focus on current:</th>
<th>Group</th>
<th>Process</th>
<th>Thread</th>
<th>Step Threads Together</th>
</tr>
</thead>
</table>

• Cannot create a thread group

U.S. DEPARTMENT OF ENERGY Office of Science
Parallel Stack

- ‘Stacks’ tab in the lower left panel
- Combines the call trees from many processes and display them together
- This compact display is invaluable when dealing with a large numbers of processes
- Move up and down the stack frame to choose the function (frame) of interest
  - The source pane shows the relevant location accordingly
- Hovering your mouse over it reveals the number of processes, process ranks and thread IDs that are at that location
- Can create a new process group by right-clicking on any function

<table>
<thead>
<tr>
<th>Processes</th>
<th>Threads</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>sub_op (memory_leak.f.27)</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>bti_openib_async_thread</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>sub_op (memory_leak.f.27)</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>/global/homes/w/wyang/ddt/md/memory_leak.f.27</td>
</tr>
</tbody>
</table>
• **Many ways to set a breakpoint**
  – Double-click on a line
  – Right click on a line and set from the menu
  – Click the Add Breakpoint icon from the tool bar
  – …

• **Red dot next to the line where a breakpoint is set**

• **Can be deleted similarly**

• **Selected breakpoints listed under the breakpoints tab in the lower left panel**
  – Can be made active or inactive (1st column)
  – Can set a condition using the language’s syntax
  – Can set frequency parameters for activation (start, interval, end)
  – Can save to or load from a file (right-click)
Watchpoints

- Watchpoints for variables or expressions (not lines)
  - Stops every time a variable or expression changes
- Again, many ways to set
  - Right-click on a variable in the Source pane and set
  - Right-click in the watchpoints table and select the Add Watchpoint menu item, or
  - Drag a variable to the watchpoints table from the Local Variables, Current Line or Evaluate views
- Selected watchpoints listed under the watchpoints tab in the lower left panel
  - Can be made active or inactive (1st column)
  - Can set a condition using the language’s syntax
  - Can save to or load from a file (right-click)
- A watchpoint is automatically removed once the target variable goes out of scope
  - To watch the value pointed to by a pointer \( p \) when \( p \) goes out of scope: right-click on \( *p \) in the watchpoints table and select the Pin to address menu item
Tracepoints

• When a tracepoint is reached
  – Prints the file and line number of the tracepoint and the value of variables or expressions (if requested)
  – Similar to putting a print statement: `print *, “Hi, I’m here.”`

• Many ways to set
  – Right-click on a line in the source code and select the Add Tracepoint…
  – Right-click in the Tracepoints table and select Add Tracepoint, and more…

• Green dot next to the line where a tracepoint is set
• Selected tracepoints listed under the Tracepoints tab
• Tracepoint output under the Tracepoint Output tab
• Considerable resource consumption if placed in areas that generate a lot of passing
• Alike tracepoints merged across processes: can lose the order/causality between different processes in the tracepoint output
• Source code pane: Right-click on a variable for a quick summary (i.e., type and value)

• Variable pane: Generally for local variables
  – Locals
  – Current Line(s): In the current lines
    • Can select multiple lines by clicking and dragging
  – Current Stack: For stack arguments

• Evaluate pane: Variables or expressions
  – Many ways to add to the Evaluate pane...
  – Limited Fortran intrinsic functions allowed

• Sparklines for variation over processes

• Can change the value of a variable in the Evaluate pane
• If a value varies across processes, the value is highlighted in green
• When the value changes due to stepping or switching processes, the value is highlighted in blue.
• Viewing an array
  – Innermost index = fastest moving index

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_array</td>
<td></td>
</tr>
<tr>
<td>[0]</td>
<td>[0]</td>
</tr>
<tr>
<td>[1]</td>
<td>[1]</td>
</tr>
<tr>
<td>[2]</td>
<td>[2]</td>
</tr>
<tr>
<td>[3]</td>
<td>[3]</td>
</tr>
<tr>
<td>[4]</td>
<td>[4]</td>
</tr>
</tbody>
</table>

C, C++:
c_array[3][5]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_array</td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td>[1]</td>
</tr>
<tr>
<td>[2]</td>
<td>[2]</td>
</tr>
<tr>
<td>[3]</td>
<td>[3]</td>
</tr>
<tr>
<td>[4]</td>
<td>[4]</td>
</tr>
<tr>
<td>[5]</td>
<td>[5]</td>
</tr>
</tbody>
</table>

Fortran:
f_array(3,5)
MDA (Multi-dimensional Array)

- Right click on an array and select ‘View Array (MDA)’
- Index order: $i, j,...$, same as what’s used in the code
- Can filter by value: use an expression using $value, i, j$ in the Only show if window
- Can visualize the array (showing only the filtered values)
- Can show statistics (for the filtered data only)
- Distributed Array Dimension to show data of other processes as well
  - The overall size must be the same as the number of processes
CPC and CTC

• Cross-Process Comparison or Cross-Thread Comparison
  – comparison of the value of a variable across MPI processes or OpenMP threads
  – Good for comparing scalars
• How to use: Right-click on a variable and select the View Across Processes (CPC) or View Across Threads (CTC) option
Message Queues

- Examining status of the internal MPI message buffers when a program hangs
  - Can be helpful in detecting deadlock
- Message queue debugging only available on Carver
- Three queues can be examined
  - Send Queue:
    - Calls to MPI send not yet completed
    - Red arrow
  - Receive Queue:
    - Calls to MPI receive not yet completed
    - Green arrow
  - Unexpected Message Queue:
    - Messages received but MPI receive not yet posted
      - This queue not relevant on Carver?
    - Dashed blue arrow
- How to use: select ‘View > Message Queues’
Message Queues Examples

- A loop because of deadlock
- A loop does not necessarily mean deadlock as MPI send can complete for a small message
Message Queues Examples (cont’d)

Receive queue messages look ambiguous

call mpi_recv(rbuf,n,mpi_real,mpi_comm_world,req,ier)
call do something(tbuf,n,t)
call mpi_send(sbuf,n,mpi_real,mpi_comm_world,req,ier)
call mpi_waitall(2,req,stat,ier)

OK, ‘always’ is a strong word as a message can end up as an unexpected message

Receive queue messages look ambiguous because of use of MPI_ANY_SOURCE
• The number of threads is specified in the Run window
  – Don’t set OMP_NUM_THREADS before starting DDT GUI
• Allinea’s warnings:
  – Stepping often behaves unexpectedly inside parallel regions
  – Some compilers optimise parallel loops regardless of the options your specified on the command line
• General impression: difficult to coordinate threads in a moderate to complex parallel region
DDT’s thread IDs: 1,2,… (not 0,1,…)  
Random thread in focus (not thread 1) when a parallel region is entered  
Synchronize threads using ‘Step Threads Together’ – otherwise can lose track of threads in a complex loop  
Need to follow a specific way; otherwise, things can become difficult to control (or run into a prob)  
Stepping into a parallel region  
   – Set the ‘Step threads together’ box first  
   – ‘Run to here’  
Stepping out of a parallel region  
   – Keep the ‘Step threads together’ box on  
   – ‘Run to here’ to a line outside the parallel region  
Outside a parallel region  
   – Leave ‘Step threads together’ off manually  
Hover the mouse over to see thread IDs
Debugging MPI + OpenMP Programs

- **MPI + OpenMP debugging supported**
  - Set # of processes and threads in the Run window
- **But cannot ‘Step Threads Together’ as a process group in a parallel region**
  - Each MPI rank needs to take turns to step threads together; select ‘Process’ for ‘Focus on current’ to do that
  - Not easy to debug a complex parallel region for a large tasks
  - The feature may be available in a future release

![Image of debugging interface with 4 MPI tasks and 6 OpenMP threads each; Rank 0 and its thread 3 in focus]
Debugging CAF Programs

- Change ‘MPI/UPC Implementation’ from default ‘Cray XT/XE/XK (MPI/shmem)’ to ‘Cray XT/XE/XK (Co-Array Fortran)’ using the Run and Options windows.
‘Distributed Array Dimensions’ in MDA used for coarrays automatically
• Similarly, set ‘MPI/UPC Implementation’ to ‘Cray XT/XE/XK (UPC)’
• “UPC Thread” = Process
• Using CPC or the *Distributed Array Dimension* for MDA for a shared variable is meaningless since all the values are displayed already (an exception could be: local shared memory allocation done with upc_alloc).

'MDA shows all elements for a shared variable, including those that have affinity with other threads.'
Memory Debugging

• Intercept calls to the system memory allocation library to get memory usage and monitor correct usage

• Overview for the next slides
  – Building code for memory debugging
  – Getting memory usage info
    • Current Memory Usage
    • Overall Memory Stats
  – Detecting memory leaks (example)
  – Detecting heap overflow/underflow + example
  – One more memory debugging example
Building Code for Memory Debugging

• Carver
  – Build as usual
  – Select ‘Preload the memory debugging library’ in DDT’s Memory Debugging Option window (shown later)

• Hopper/Franklin: See next slides
  – See Appendices B and C of the DDT User Guide for all the complexities
## Statically Liked Binary on Hopper/Franklin for Memory Debugging

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Fortran</th>
<th>C, C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGI</td>
<td>% ftn -g -c prog.f</td>
<td>% cc -g -c prog.c</td>
</tr>
<tr>
<td></td>
<td>% ftn -Bddt -o prog prog.o</td>
<td>% cc -Bddt -o prog prog.o</td>
</tr>
<tr>
<td>GNU</td>
<td>% ftn -g -c prog.f</td>
<td>% cc -g -c prog.c</td>
</tr>
<tr>
<td></td>
<td>% ftn -v -o prog prog.o</td>
<td>% cc -v -o prog prog.o</td>
</tr>
<tr>
<td></td>
<td># -v to get the last linker line</td>
<td></td>
</tr>
<tr>
<td>Cray</td>
<td>% ftn –g –c prog.f</td>
<td>% cc –g –c prog.c</td>
</tr>
<tr>
<td></td>
<td>% ftn –v –o prog prog.o</td>
<td>% cc –v –o prog prog.o</td>
</tr>
<tr>
<td></td>
<td># -v to get the last linker line</td>
<td></td>
</tr>
<tr>
<td>Intel</td>
<td>% ftn –g –c prog.f</td>
<td>% cc –g –c prog.c</td>
</tr>
<tr>
<td></td>
<td>% ftn –v –o prog prog.o</td>
<td>% cc –v –o prog prog.o</td>
</tr>
<tr>
<td></td>
<td># -v to get the last linker line</td>
<td></td>
</tr>
<tr>
<td>PathScale</td>
<td>% ftn -g -Wl,--export-dynamic -TENV:frame_pointer=ON -funwind-tables -c prog.f</td>
<td>% cc -g -c prog.c</td>
</tr>
<tr>
<td></td>
<td>% ftn –v -Wl,--export-dynamic -TENV:frame_pointer=ON -funwind-tables -o prog prog.o</td>
<td>% cc -v -o prog prog.o</td>
</tr>
<tr>
<td></td>
<td># -v to get the last linker line</td>
<td></td>
</tr>
</tbody>
</table>

- **Deselect ‘Preload the memory debugging library’ in the Memory Debugging Option window**
Dynamically Linked Binary on Hopper for Memory Debugging

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Fortran</th>
<th>C, C++</th>
</tr>
</thead>
</table>
| PGI, Cray | % ftn -g -c prog.f 
% ftn –dynamic -o prog prog.o \ 
${DDT_LINK_DMALLOC} -Wl,--allow-multiple-definition | % cc -g -c prog.c 
% cc -dynamic -o prog prog.o \ 
${DDT_LINK_DMALLOC} -Wl,--allow-multiple-definition |
| GNU, Intel | % ftn -g -c prog.f 
% ftn –dynamic –o prog prog.o \ 
${DDT_LINK_DMALLOC} –zmuldefs | % cc -g -c prog.c 
% cc –dynamic -o prog prog.o \ 
${DDT_LINK_DMALLOC} -zmuldefs |
| PathScale | % ftn -g -Wl,--export-dynamic -TENV:frame_pointer=ON \ 
-funwind-tables -c prog.f 
% ftn –dynamic \ 
-Wl,--export-dynamic -TENV:frame_pointer=ON \ 
-funwind-tables -o prog prog.o \ 
${DDT_LINK_DMALLOC} -Wl,--allow-multiple-definition | % cc -g -c prog.c 
% cc -dynamic -o prog prog.o \ 
${DDT_LINK_DMALLOC} -Wl,--allow-multiple-definition |

- ‘Preload the memory debugging library’ in the Memory Debugging Option window
- Before running ddt
  - Load the same PrgEnv
  - Set CRAY_ROOTFS before running ddt

    setenv CRAY_ROOTFS DSL # for csh/tcsh shell
    export CRAY_ROOTFS=DSL # for sh/ksh/bash shell
• Enable ‘Memory Debugging’ in the Run window
• Clicking on ‘Details’ opens the ‘Memory Debugging Options’ window for setting options (next slide)
• ‘Preload the memory debugging library’ on Carver and only for a dynamically linked executable on Hopper
• Select the Language option that best matches your program
  – “Often sufficient to leave this to C++/Threaded”
• Heap Debugging level: debugging runs “fast up to Low”
• Heap Overflow/Underflow Detection
  – To detect out of bounds array references
• Heap interval:
  – check the entire heap for consistency every specified number of memory allocations
• Can enable memory debugging for selected processes only
• Options can be changed at run time
  – Select ‘Control > Memory Debugging Options’
  – Can increase debugging option level after a problem area is reached
Current Memory Usage

- Live memory usage data
- Select ‘View > Current Memory Usage’
- Graphical View
  - Pie-chart for total memory usage (in Bytes) for processes
  - Stacked bar chart
    - Broken down by functions
    - Clicking on a block in a bar shows details in the ‘Allocation Details’ box
    - Clicking on an item in the ‘Allocation Details’ box shows the stack backtrace
- Table View
  - Detailed data in numbers
  - Can be exported to spreadsheet
• Select ‘View > Overall Memory Stats’
• ‘Graph View’ tab for
  – **Total Bytes**: total allocated and freed bytes so far
  – **Total Calls**: total allocation and deallocation calls so far
  – **Current**
• **Table View**
  – Data in numbers
  – Can be exported to spreadsheet
Detecting Memory Leaks - Example

`program memory_leak
  !... Buggy code prepared for a debugger tutorial.
  implicit none
  include 'mpif.h'
  integer, parameter :: n = 1000000
  real val
  integer i, ierr
  call mpi_init(ierr)
  val = 0.
  do i=1,10
    call sub_ok(val,n)
    end do
  do i=1,10
    call sub_bad(val,n)
    end do
  do i=1,10
    call sub_badx2(val,n)
    end do
  print *, val
  call mpi_finalize(ierr)
end

subroutine sub_ok(val,n)
  integer n
  real val
  real, allocatable :: a(:)
  allocate (a(n))
  call random_number(a)
  val = val + sum(a)
  deallocate(a)
end

subroutine sub_bad(val,n)
  integer n
  real val
  real, pointer :: a(:)
  allocate (a(n))
  call random_number(a)
  val = val + sum(a)
  deallocate(a)
end

subroutine sub_badx2(val,n)
  integer n
  real val
  real, pointer :: a(:)
  allocate (a(n))
  call random_number(a)
  val = val + sum(a)
  allocate (a(n))
  call random_number(a)
  val = val + sum(a)
  deallocate(a)
end`

‘Run to here’ at line 10, 14, 17 and 20 and check memory usage.

no memory leak

memory leak of 4*n bytes per call

memory leak of 8*n bytes per call

! not ok to deallocate

! not ok not to deallocate again

! not ok not to deallocate
Detecting Memory Leaks - Example (cont’d)

- ‘Current Memory Usage’ at, for example, line 20 (gcc/4.6.2, mpich2/5.4.1, …)
  - Large (~ 120 million bytes) “unexpected” memory usage
  - Many pointers for 4 million bytes allocated in subroutines
  - Useful info under the Table View, too
Detecting Memory Leaks - Example (cont’d)

- ‘Overall Memory Stats’ results for rank 0 for a 8-PE run on Hopper (gcc/4.6.2, xt-mpich2/5.4.1, …)

<table>
<thead>
<tr>
<th>Location</th>
<th>Total Allocated bytes</th>
<th>Total Freed bytes</th>
<th>Current Memory Usage (TAB-TFB)</th>
<th>Total Allocation Calls</th>
<th>Total Free Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before sub_ok loop</td>
<td>36,348</td>
<td>13,702</td>
<td>22,646</td>
<td>166</td>
<td>72</td>
</tr>
<tr>
<td>After 10 calls to sub_ok</td>
<td>40,036,348</td>
<td>40,013,702</td>
<td>22,646</td>
<td>176</td>
<td>82</td>
</tr>
<tr>
<td>After 10 calls to sub_bad</td>
<td>80,036,348</td>
<td>40,013,702</td>
<td>40,022,646</td>
<td>186</td>
<td>82</td>
</tr>
<tr>
<td>After 10 calls to sub_badx2</td>
<td>160,036,348</td>
<td>40,013,702</td>
<td>120,022,646</td>
<td>206</td>
<td>82</td>
</tr>
</tbody>
</table>

- Memory leak of 4 million bytes after each call to sub_bad
- Memory leak of 8 million bytes after each call to sub_badx2
• Detect an out of bound reference with dynamically allocated arrays
• Set guard pages (page=4 KB) before or after (but not both) allocated blocks for detection
  – For reads and writes
  – Instant detection
• Fence post checking even if guard pages are not used
  – As long as the debugging level is ‘Runtime’ or above
  – A pattern is written into a very small extra portion around an allocated block and DDT checks this area for corruption
  – Thus, checking only for writes
  – “Very small” - Not to be relied on as the primary detection tool
  – Program stops at the next heap consistency check; thus, the location can be slightly inaccurate
Heap Overflow/Underflow Detection

Example

- With the settings in the previous slide (i.e., 1 guard page set ‘After’), a heap overflow (but not a underflow) is detected.
- Guard pages may not function correctly with PGI Fortran as it wraps F90 allocations in a compiler-handled allocation area.
More Memory Debugging Example

- Deallocating the same memory block twice
- Not sure if it’s DDT who stops the program in this example
Opening Core Files

- For post-mortem debugging
- Select ‘Open Core Files’ in the Welcome window
- Specify core files and the executable that generated them

Note: Do the following in a batch script in order to generate core files

- Set coredumpsize to unlimited
  
  ```
  limit coredumpsize unlimited # for csh/tcsh
  ulimit -c unlimited            # for sh/ksh/bash
  ```

- Remove existing core files
- Run the code
Opening Core Files (cont’d)

- Cannot run, pause or step
- Can only check variables, stack frames in the core file, and evaluate expressions

Stack backtrace when the code crashed

That was the problem!

Can evaluate expressions
Offline Debugging

• Run a code under DDT control, but without user intervention and without a user interface
  – Command-line mode
  – Good for quickly getting tracepoint and stacktrace output in a batch job
  – Good for a “parameter study” – checking for an error condition for a range of a parameter value – which can become tedious with GUI
  – Stacktrace of crashing processes recorded when an error occur

• Use ‘ddt -offline…’ in place of ‘aprun …’ or ‘mpirun …’ in your batch script

```bash
#!/bin/csh
#PBS -l mppwidth=...
...
cd $PBS_O_WORKDIR
module load ddt
ddt -offline filename.html -n 4 myprogram arg1 ... # to get HTML output file
ddt -offline filename -n 4 myprogram arg1 ... # to get plain text output file
```
Offline Debugging Options

- `–n ...`: # of processes
  - # of OpenMP threads must be specified via OMP_NUM_THREADS environment variable

- `–ddtsession sessionfile`
  - Use the DDT session file saved during a GUI session using the “Saved Session”
  - Session file defines tracepoints and breakpoints

- `–memory`
  - Enable memory debugging

- `–trace-at LOCATION[,N:M:P],VAR1,VAR2,...`
  - Set a tracepoint at LOCATION (either file:line or function name), beginning recording after N visits, and recording every M-th subsequent pass until it has been triggered P times
  - Use ‘-’ in N:M:P for the default value
  - Record the value of the variable VAR1, VAR2,...

- `–break-at LOCATION[,N:M:P]`
  - Set a breakpoint similarly
  - Stack trace is recorded
  - Continue after reaching the breakpoint

- **No process group control yet?**
  - Just group ‘All’
Offline Debugging Example

Incomplete tracepoint output may be seen on Hopper with a tracepoint near the end of program if ‘Stop at exit/_exit’ is not selected in the ‘Control > Default Breakpoints’ menu during a GUI session before running the offline debugging command.
More Help?

• User guide on each machine
  – \$DDT_DOCDIR/userguide.pdf

  % module load ddt
  % ls -l $DDT_DOCDIR/userguide.pdf

  – From DDT: Help > User Guide

• http://www.nersc.gov/users/software/debugging-and-profiling/ddt/
  – OK, will update soon…

• http://www.allinea.com/
Acknowledgements

• Thanks to Allinea staff for answering many questions while preparing this talk
• Using the PGI compiler, run memory debugging on Hopper to get memory usage stats at the 4 locations in memory_leak.f, mentioned in the tutorial. Are they correct?
• Repeat after fixing the bugs.
• Do these with both statically- and dynamically-linked executables.