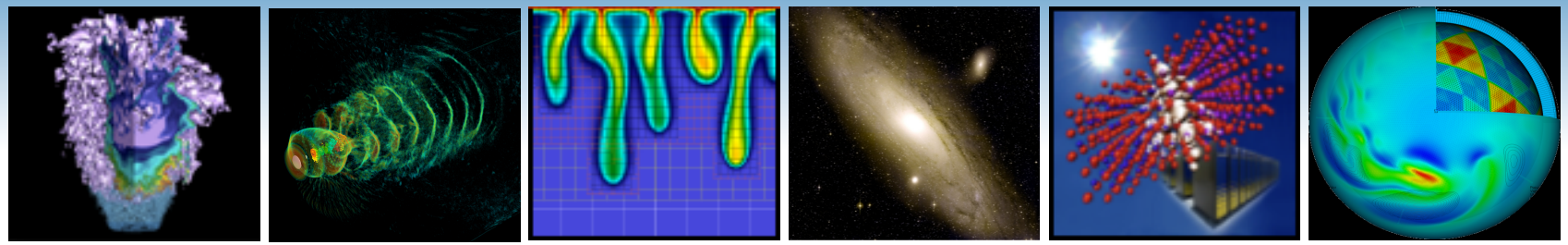


Debugging with DDT

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

U.S. DEPARTMENT OF ENERGY | Office of Science

NERSC National Energy Research Scientific Computing Center

BERKELEY LAB

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

DDT

- **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

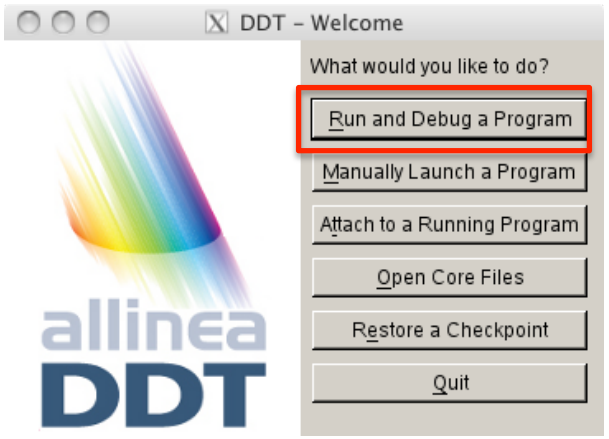
Starting DDT

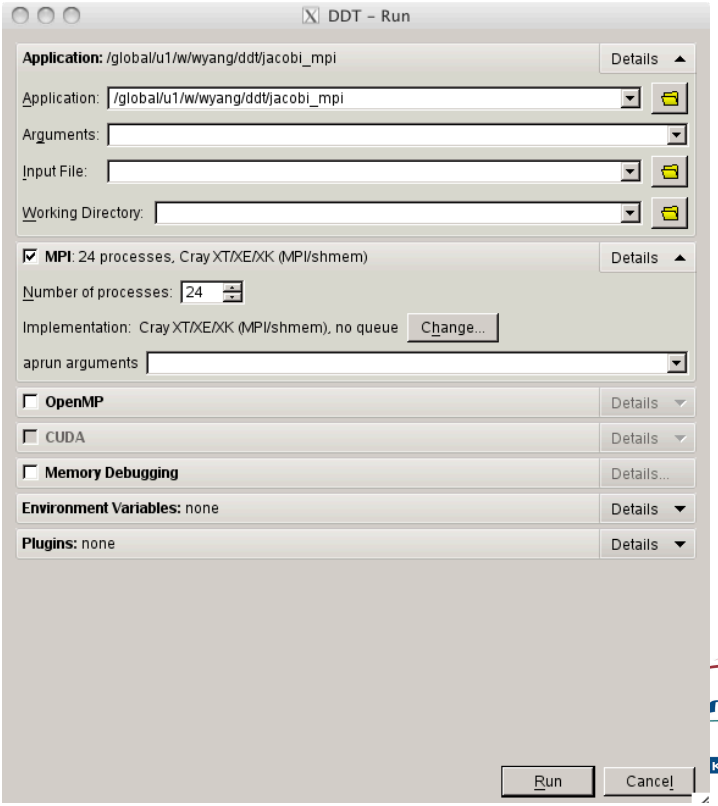
- Start DDT in an interactive batch session
- Compile the code with the –g flag

```
% 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
```

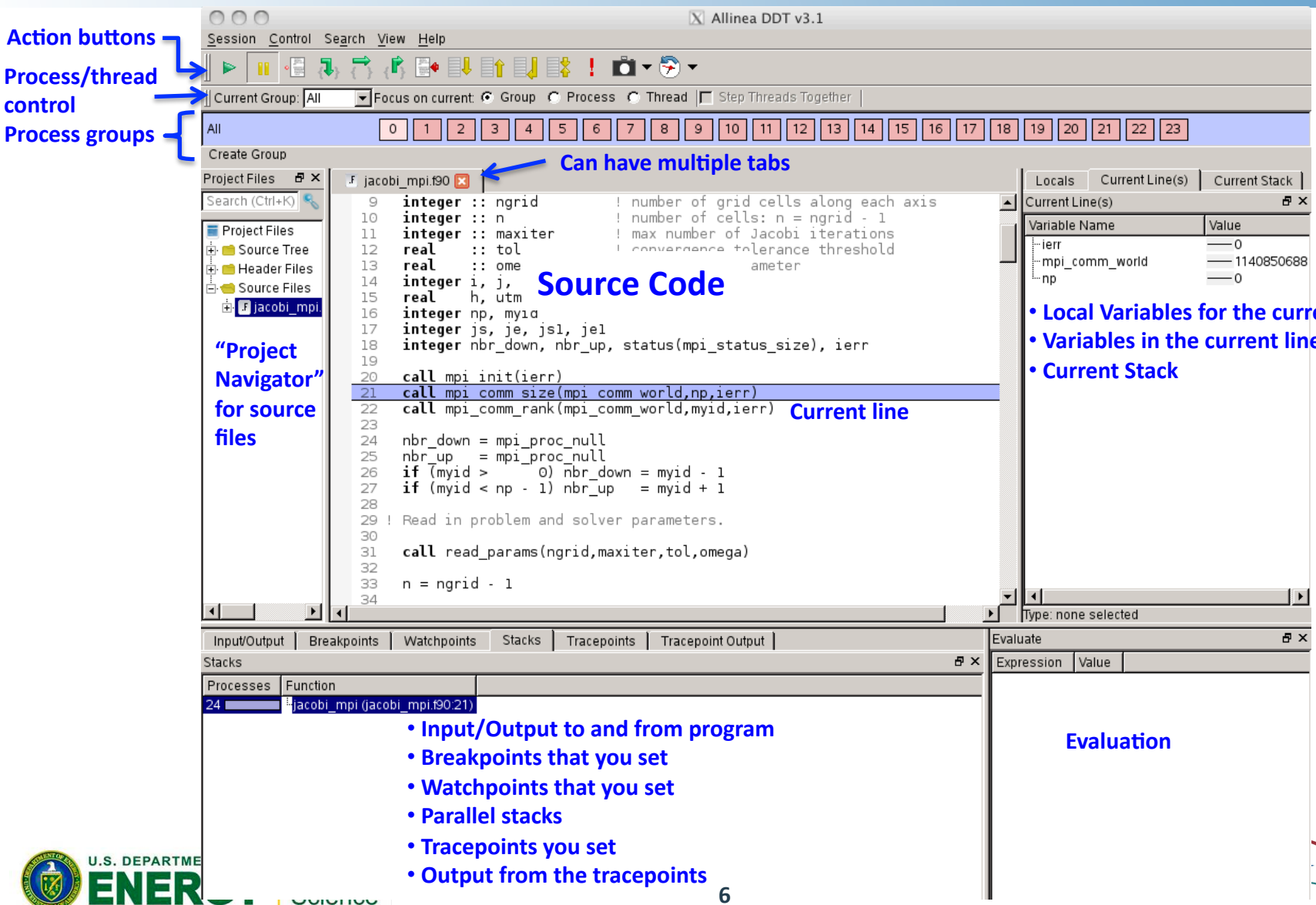
Starting DDT (cont'd)

- **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
Allinea DDT v3.1
Session Control Search View Help
Current Group: All
Focus on current: Group Process Thread
Step Threads Together
All
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Create Group
Can have multiple tabs
Project Files
Search (Ctrl+K)
Project Files
Source Tree
Header Files
Source Files
jacobi_mpi
"Project Navigator" for source files
jacobi_mpi.f90
Source Code
Current line
Locals
Current Line(s)
Current Stack
Variable Name
Value
ierr
mpi_comm_world
np
• Local Variables for the current stack
• Variables in the current line(s)
• Current Stack
Input/Output
Breakpoints
Watchpoints
Stacks
Tracepoints
Tracepoint Output
Processes
Function
jacobi_mpi (jacobi_mpi.f90:21)
• Input/Output to and from program
• Breakpoints that you set
• Watchpoints that you set
• Parallel stacks
• Tracepoints you set
• Output from the tracepoints
Evaluate
Expression
Value
Evaluation

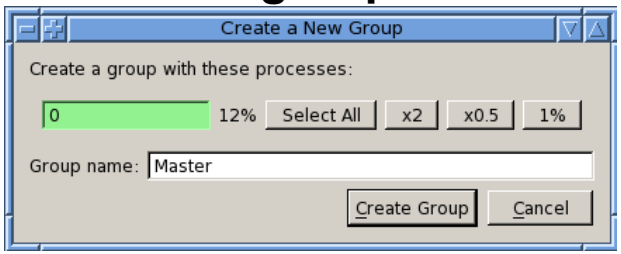
Navigating Through a Program

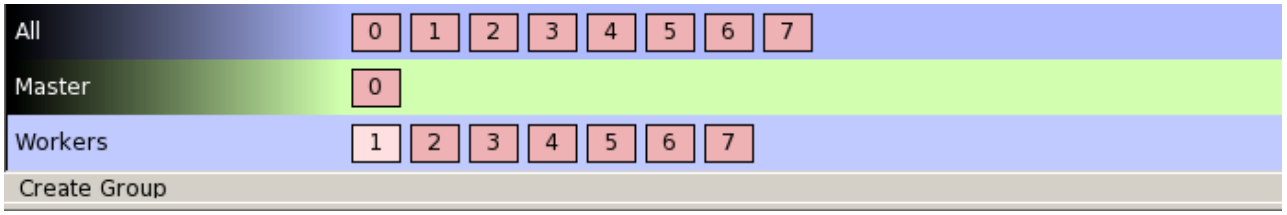
- **Play/Continue**
  - Run the program
- **Pause**
- **Step Into**
  - Move to the next line of source
  - If the next line is a function call, step to the first line of the function
- **Step Over**
  - Move to the next line of source code
  - If the next line is a function call, step over the function
- **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**

- **Can create, edit and delete groups**





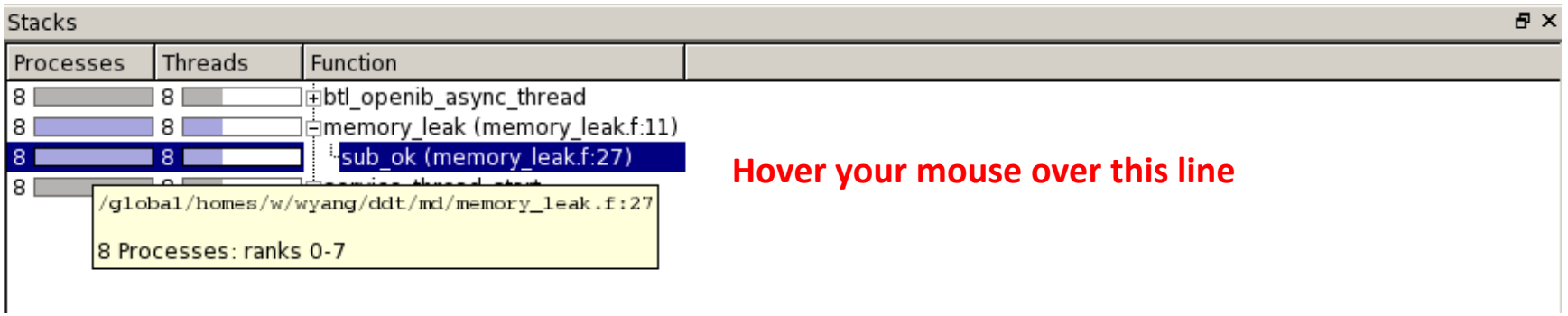
- **Select group**
  - 'Current Group' displays the name of the group in focus
  - Source Code Viewer shows the corresponding group color

- **Cannot create a thread group**

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**



Breakpoints

- **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)

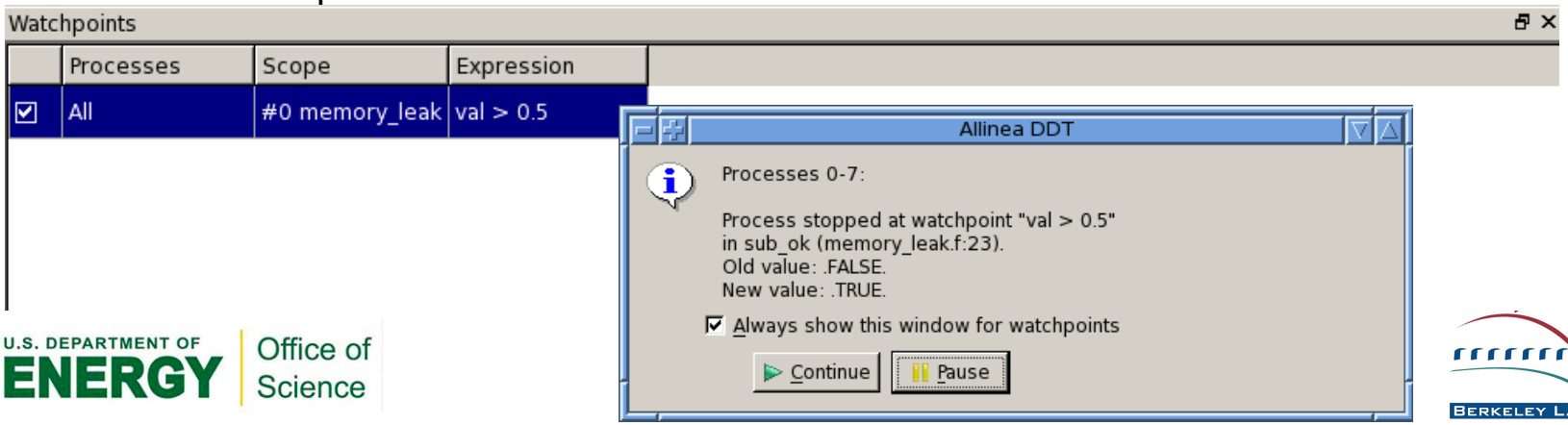
Input/Output | Breakpoints | Watchpoints | Stacks | Tracepoints | Tracepoint Output

Breakpoints

	Processes	Threads	File	Line	Function	Condition	Start After	Trigger Every	Stop Aft
☑	All	all	memory_leak.f	29		val > 0.5	0	1	-
☑	All	all	memory_leak.f	39			0	1	-
☑	All	all	memory_leak.f	49			0	1	-

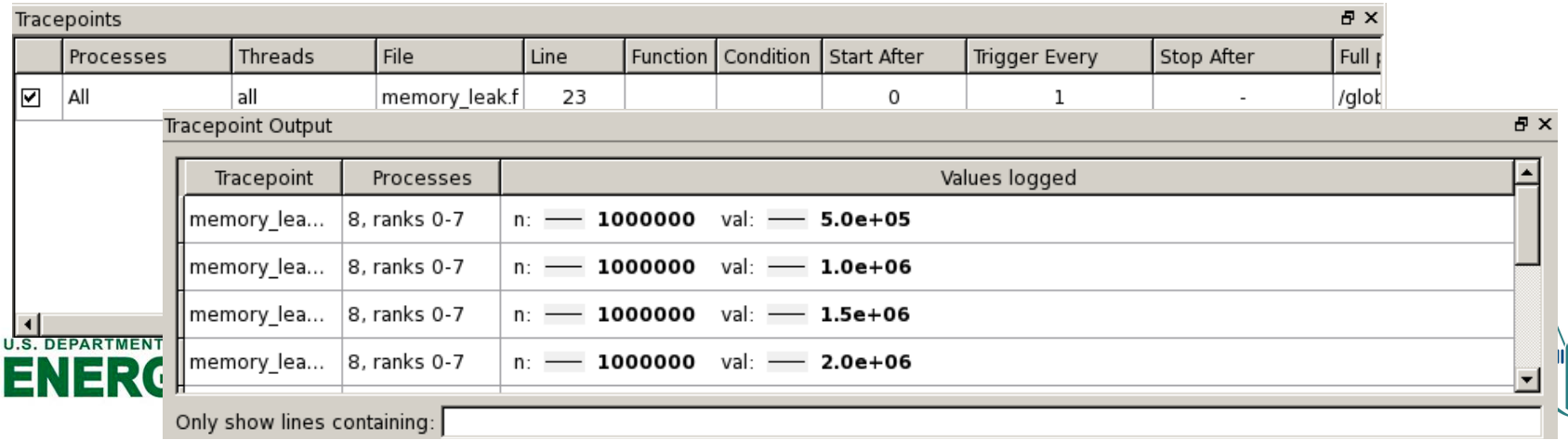
Watchpoints

- **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



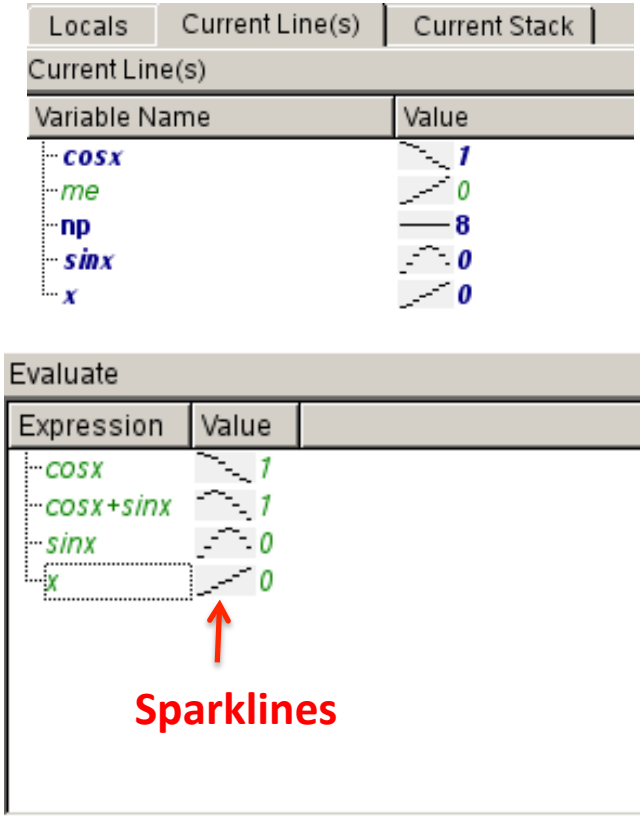
Tracepoints

- **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 ouput 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**



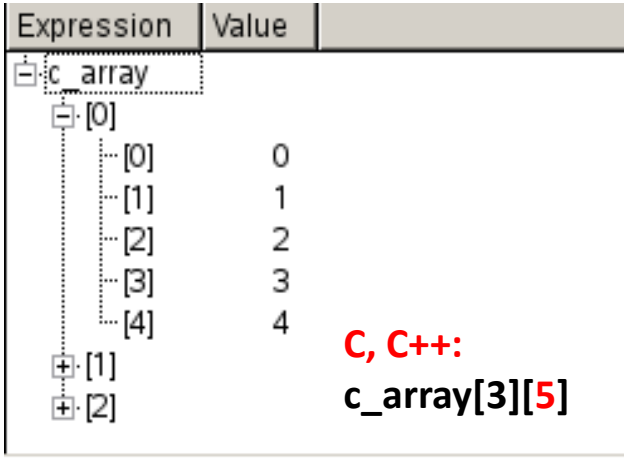
Checking Data

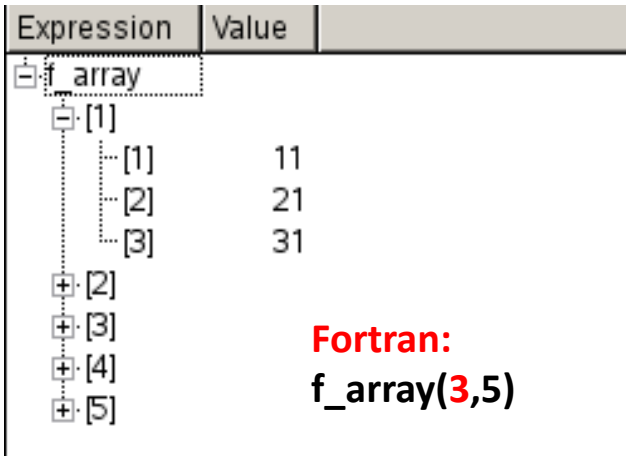
- **Source code pane: Right-click on a variable for a quick summary (i.e., type and value)**
- **Variable pane: Generally for local variables**
  - Locals
  - Current Line(s): In the current lines
    - Can select multiple lines by clicking and dragging
  - Current Stack: For stack arguments
- **Evaluate 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**



Checking Data (cont'd)

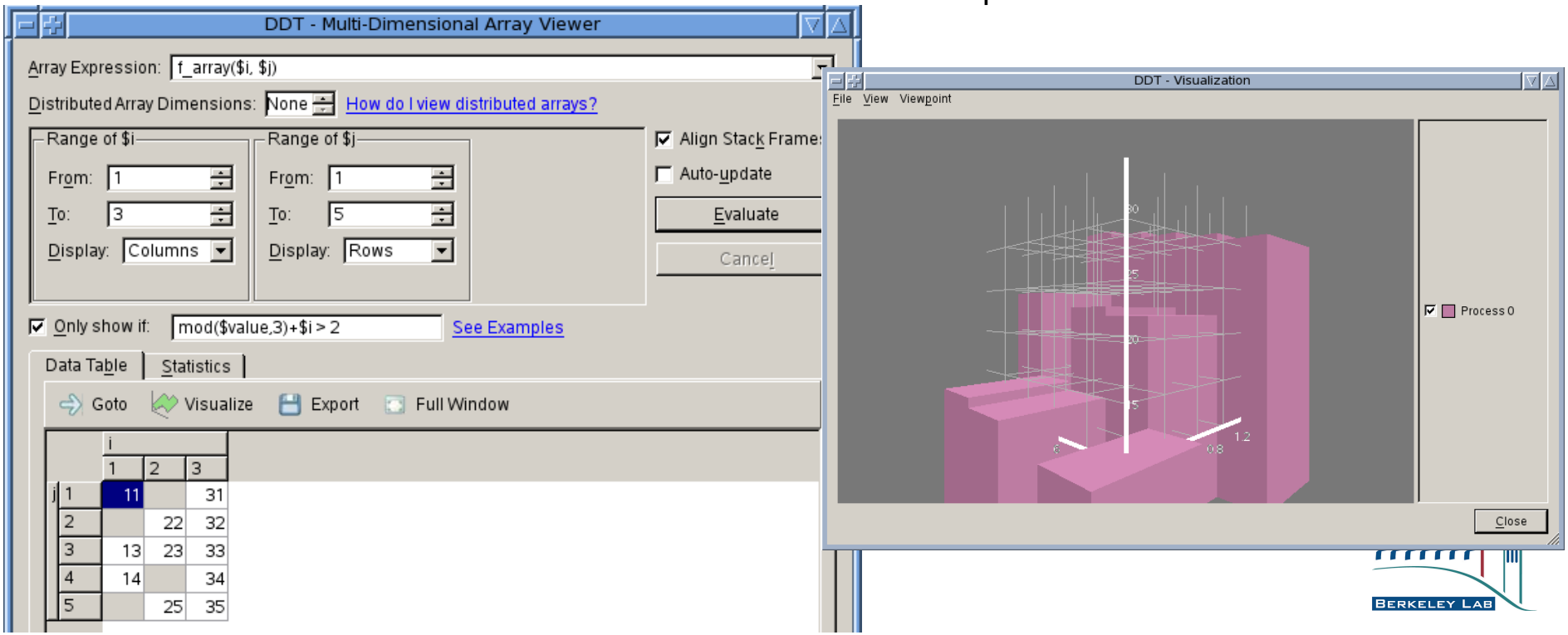
- **If a value varies across processes, the value is highlighted in green**
- **When the value changes due to stepping or switching processes, the value is highlighted in blue.**
- **Viewing an array**
  - Innermost index = fastest moving index





MDA (Multi-dimensional Array)

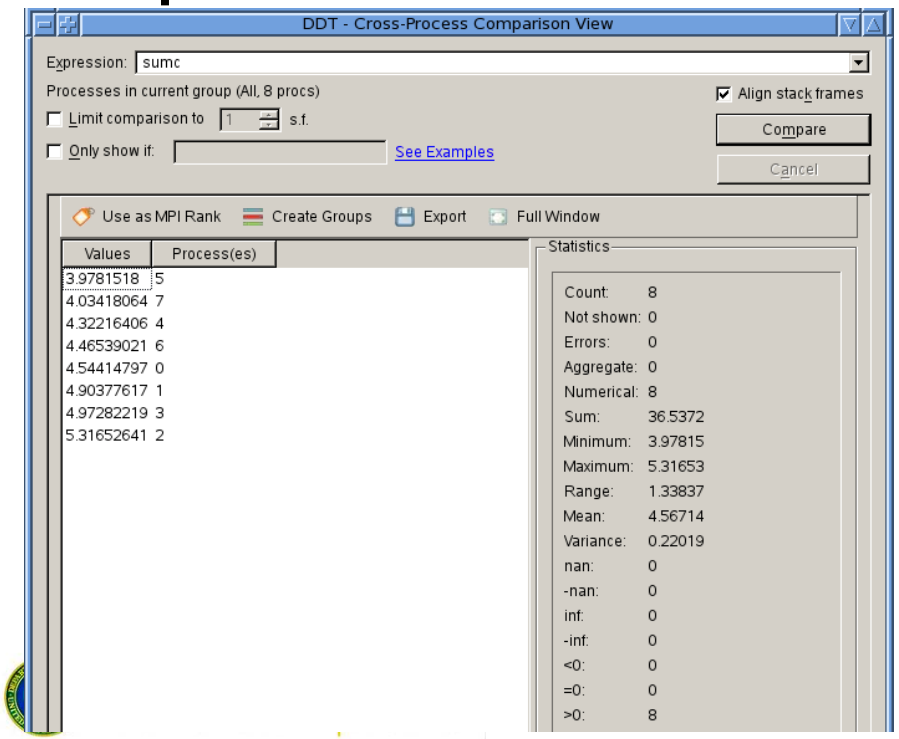
- **Right click on an array and select 'View Array (MDA)'**
- **Index order: *$i*, *$j*,…, same as what's used in the code**
- **Can filter by value: use an expression using *$value*, *$i*, *$j* in the *Only show if* window**
- **Can visualize the array (showing only the filtered values)**
- **Can show 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

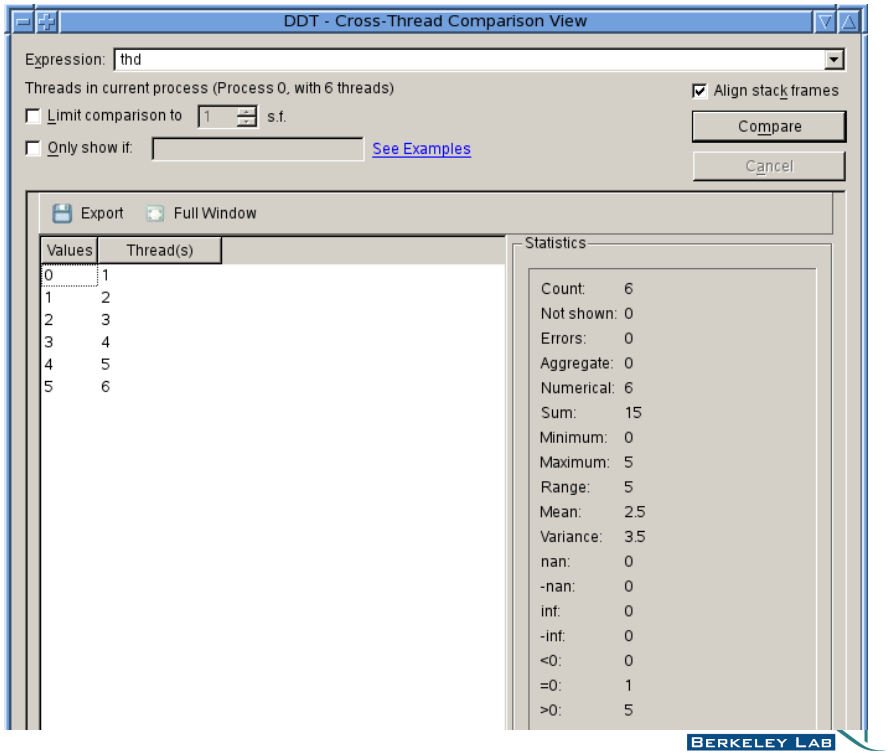


BERKELEY LAB

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'**

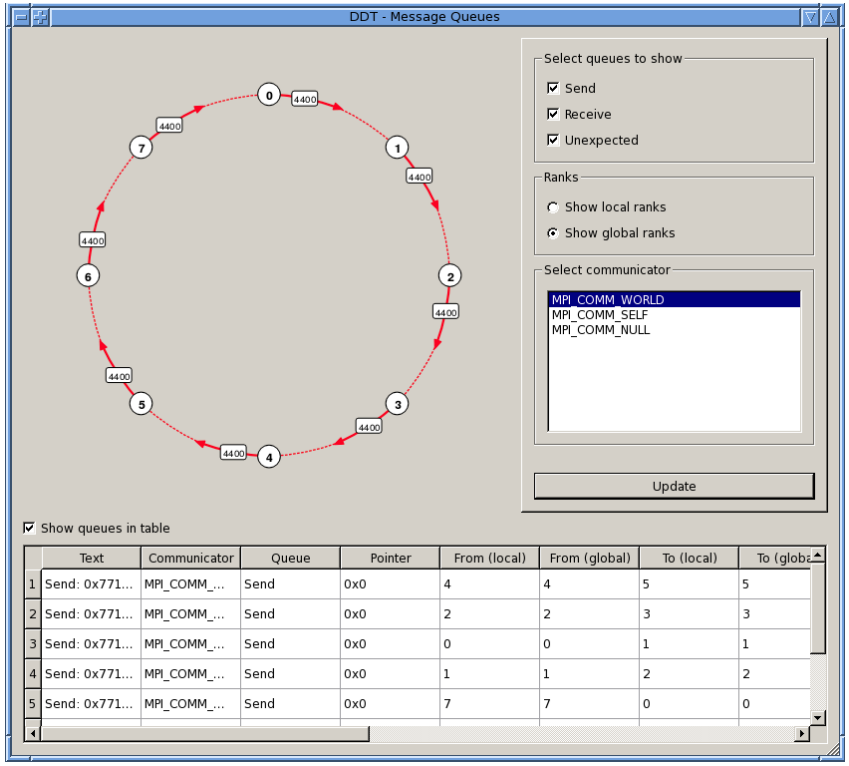
Message Queues Examples

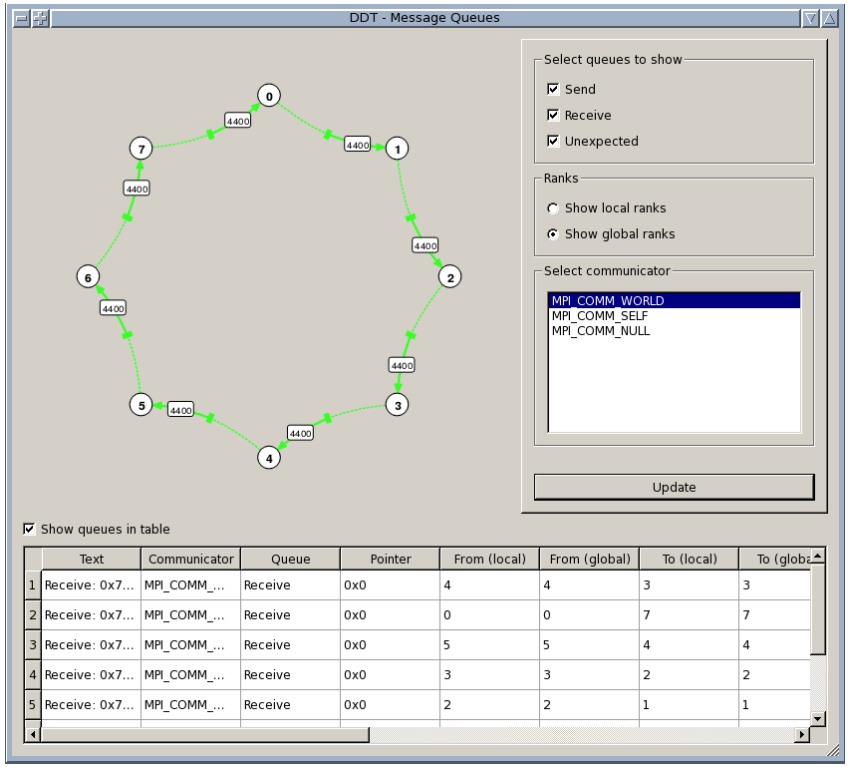
deadlock with a large message

```
call mpi send(sbuf,n,mpi real,nbr r,tag,mpi comm world,ierr)
call do_something(tbuf,n_t)
call mpi_recv(rbuf,n,mpi_real,nbr_l,tag,mpi_comm_world,stat,ierr)
```

always deadlock

```
call mpi recv(rbuf,n,mpi real,nbr l,tag,mpi comm world,stat,ierr)
call do_something(tbuf,n_t)
call mpi_send(sbuf,n,mpi_real,nbr_r,tag,mpi_comm_world,ierr)
```

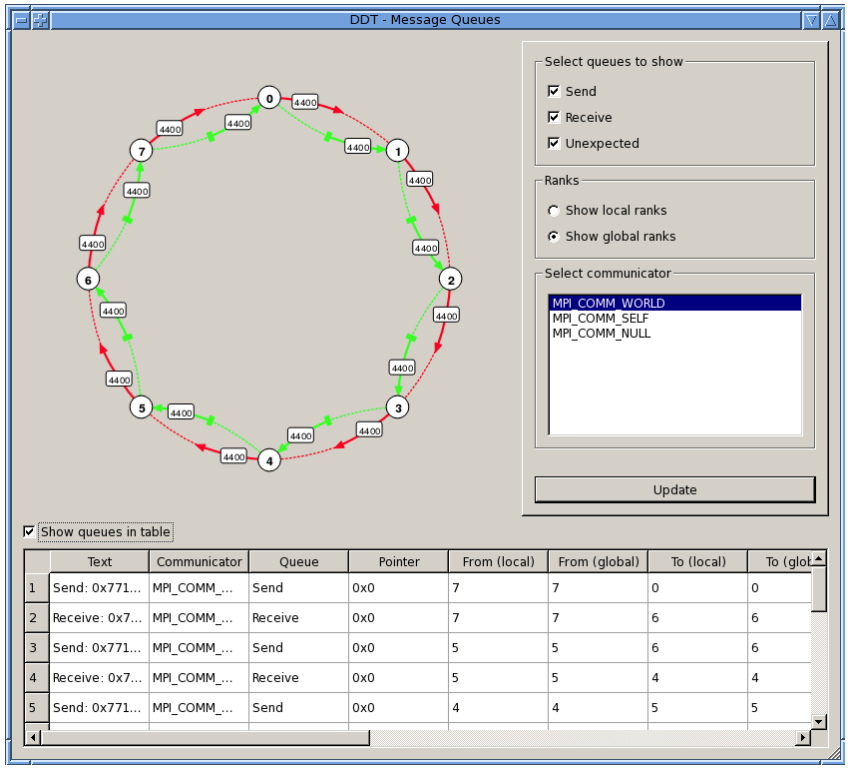




- **A loop because of deadlock**
- **A loop does not necessarily mean deadlock as MPI send can complete for a small message**

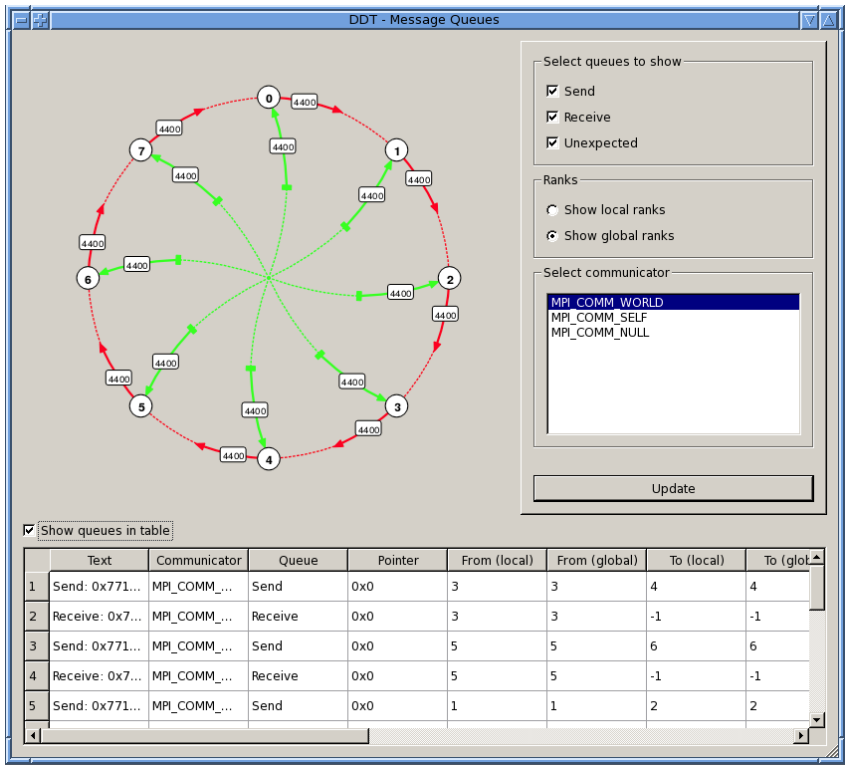
Message Queues Examples (cont'd)

```
always ok
call mpi_irecv(rbuf,n,mpi_real,nbr_l,tag,mpi_comm_world,req,ierr)
call do_something(tbuf,n_t)
call mpi_isend(sbuf,n,mpi_real,nbr_r,tag,mpi_comm_world,req(2),ierr
call mpi_waitall(2,req,stat,ierr)
```



OK, 'always' is a strong word as a message can end up as a unexpected message

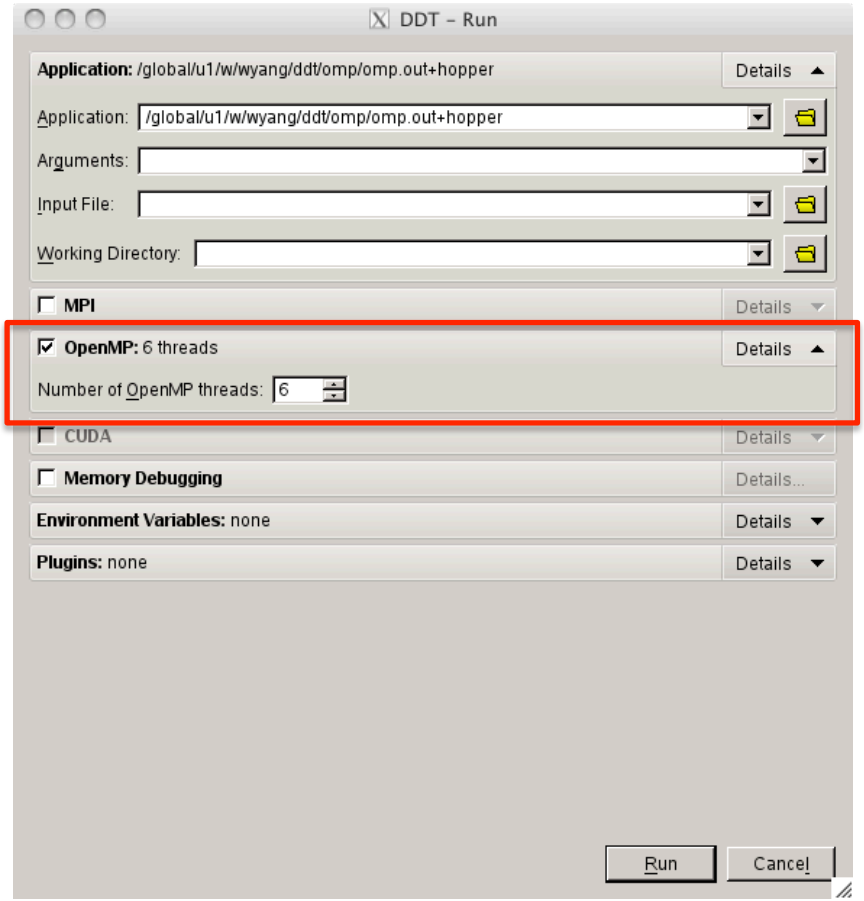
```
message queues look ambiguous
call mpi_irecv(rbuf,n,mpi_real,mpi_any_source,tag,mpi_comm_world,r
call do_something(tbuf,n_t)
call mpi_isend(sbuf,n,mpi_real,nbr_r,tag,mpi_comm_world,req(2),ier
call mpi_waitall(2,req,stat,ierr)
```



Receive queue messages look ambiguous because of use of MPI_ANY_SOURCE

Debugging OpenMP Programs

- **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**

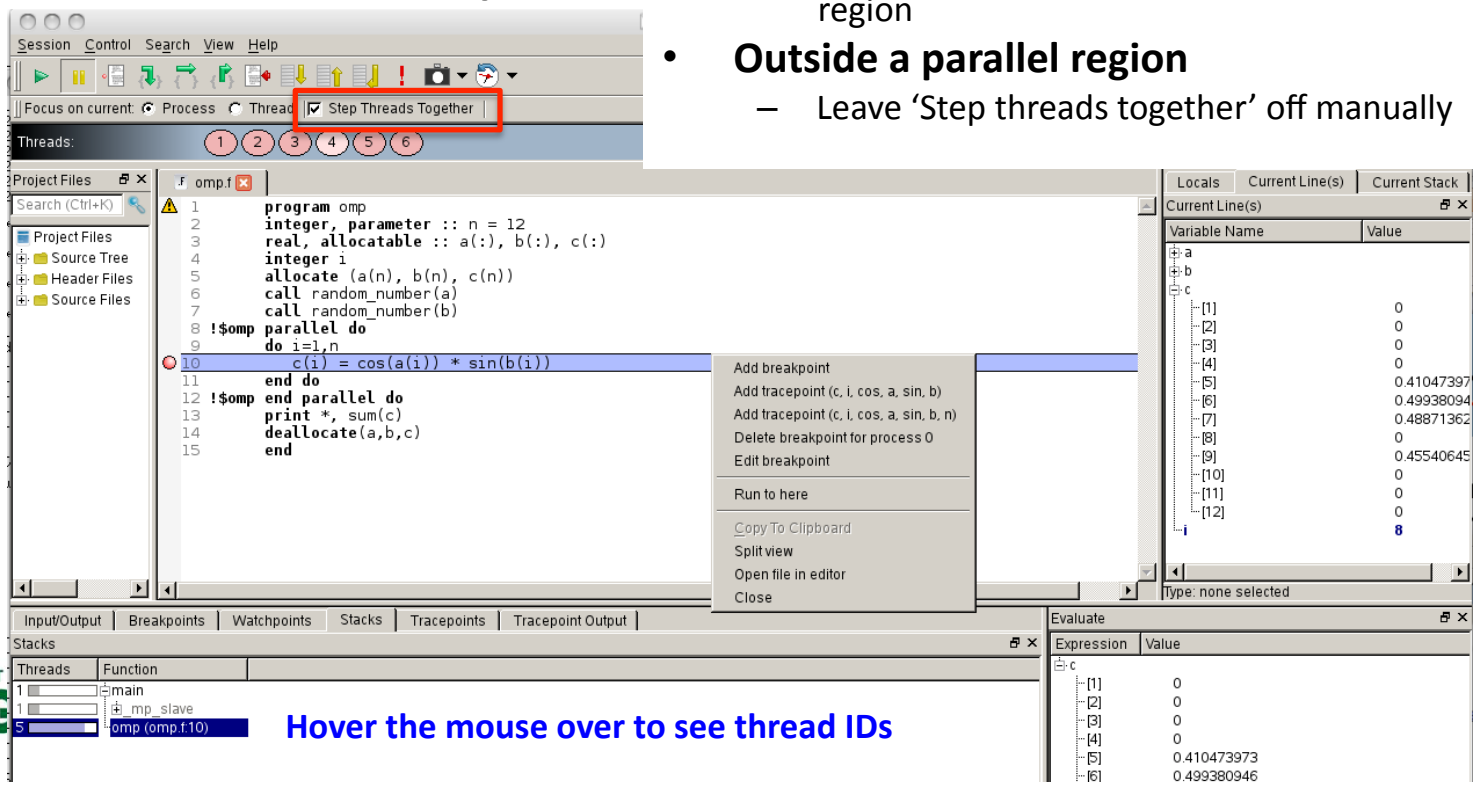


Debugging OpenMP Programs (cont'd)

- 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

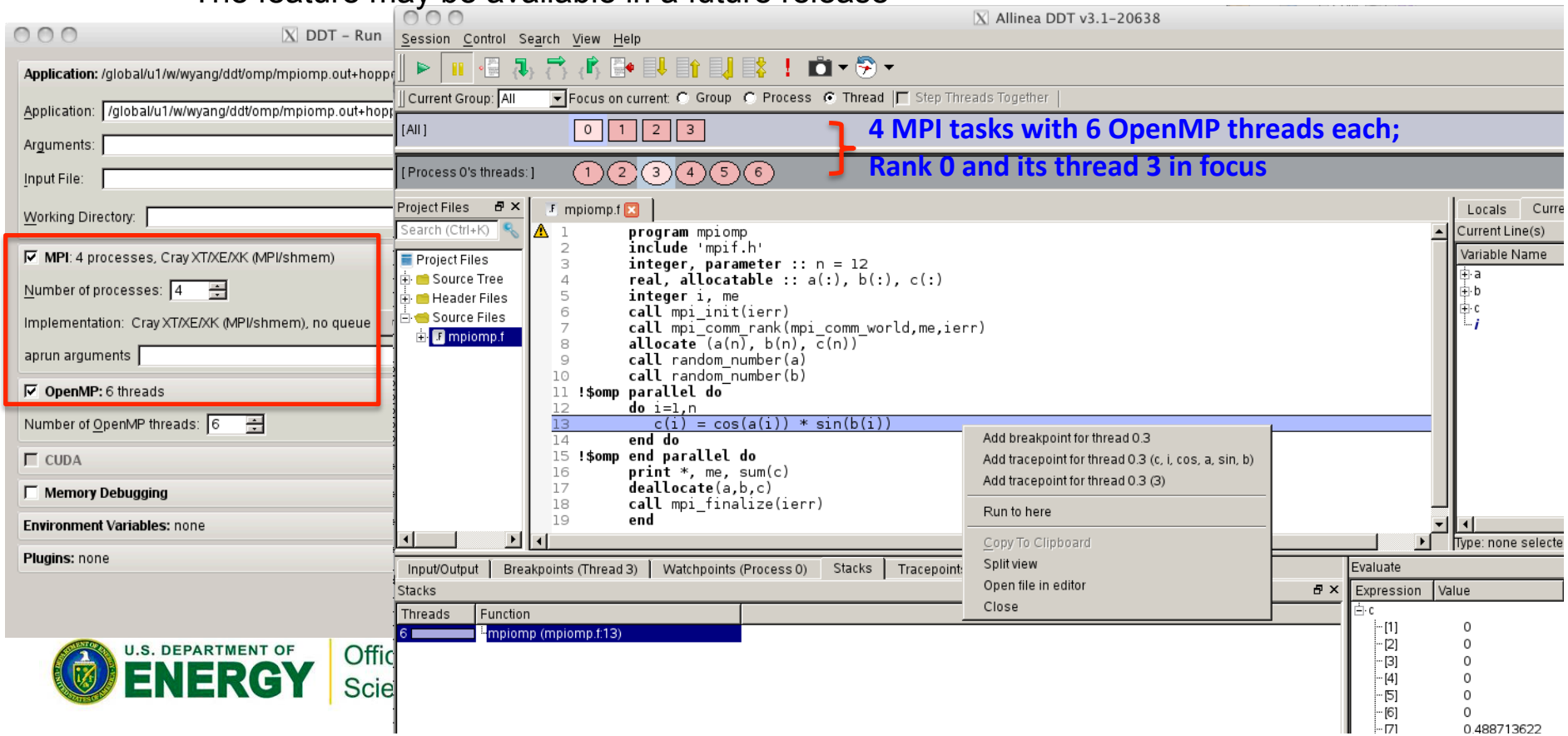
6 threads,
thread 4 in focus



Hover the mouse over to see thread IDs

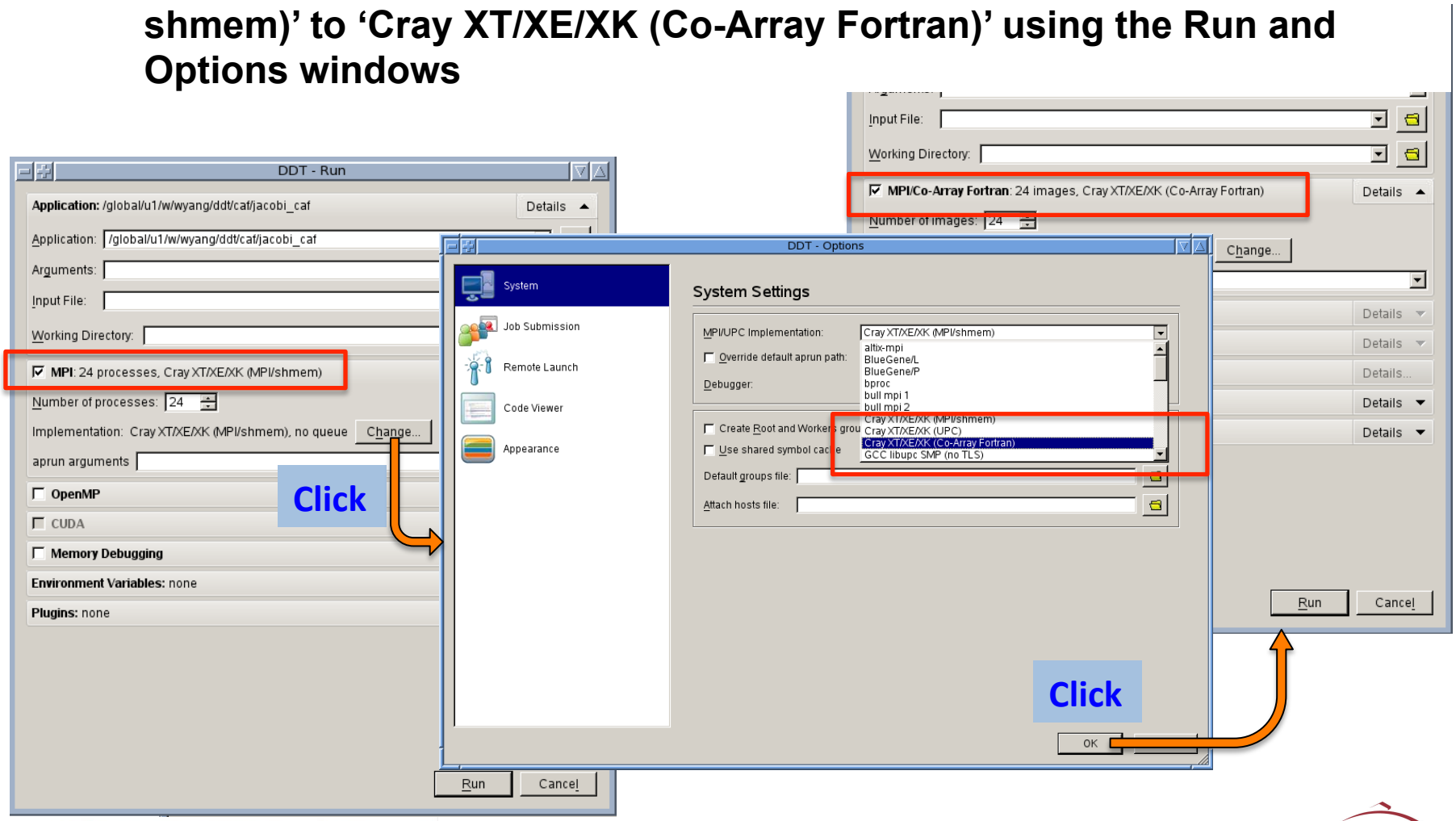
Debugging MPI + OpenMP Programs

- **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



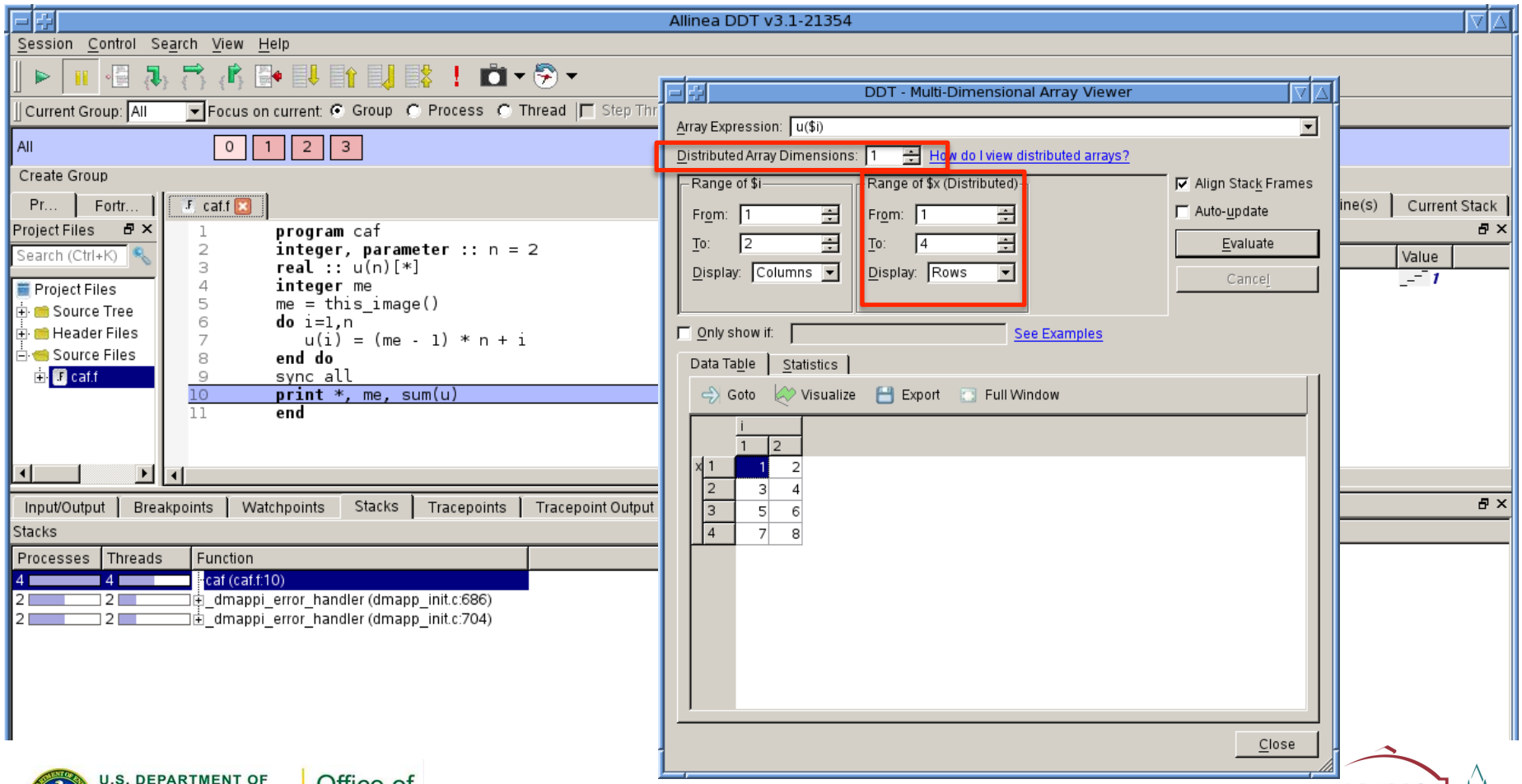
Debugging CAF Programs

- **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**



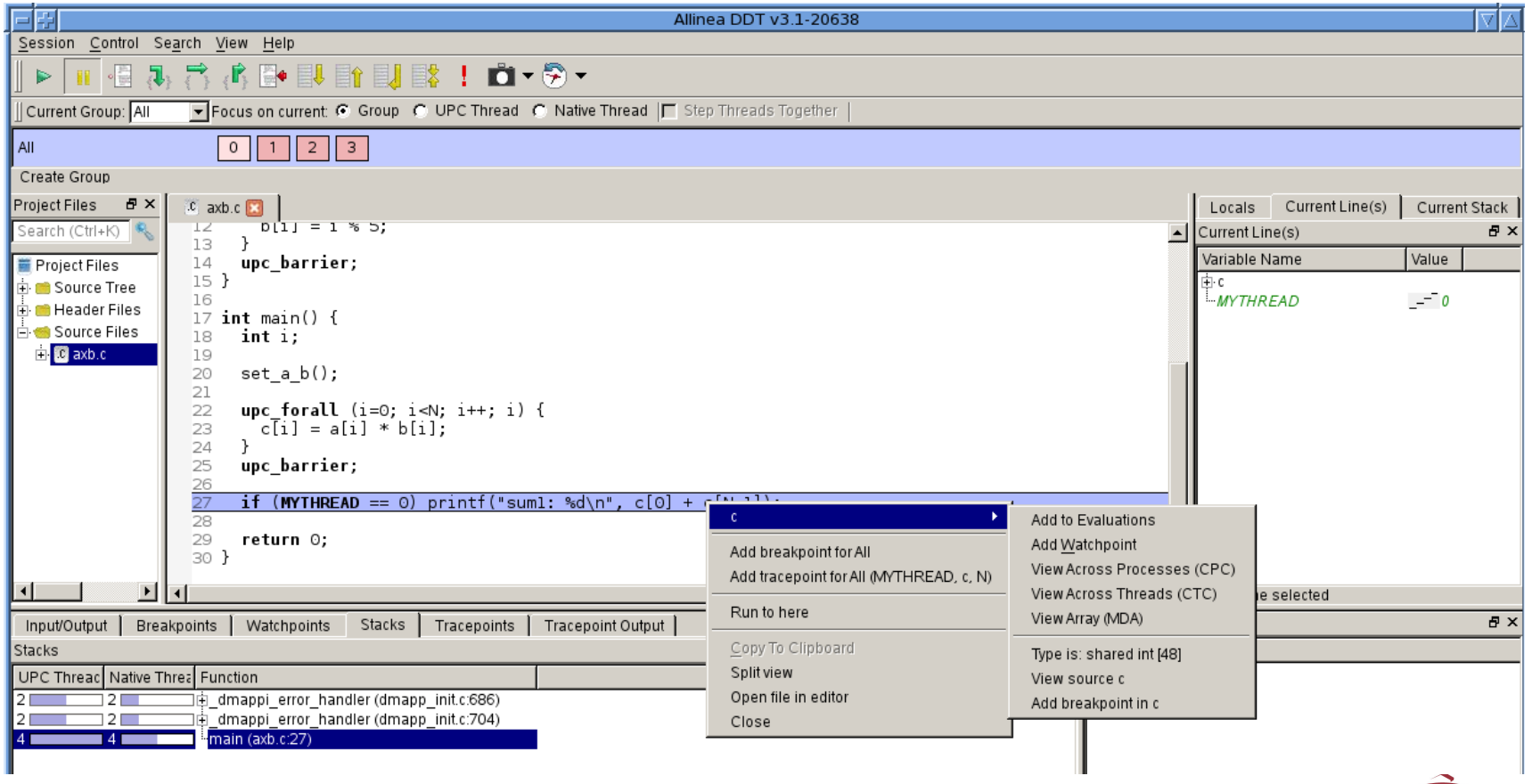
Debugging CAF Programs (cont'd)

- **'Distributed Array Dimensions' in MDA used for coarrays automatically**



Debugging UPC Programs

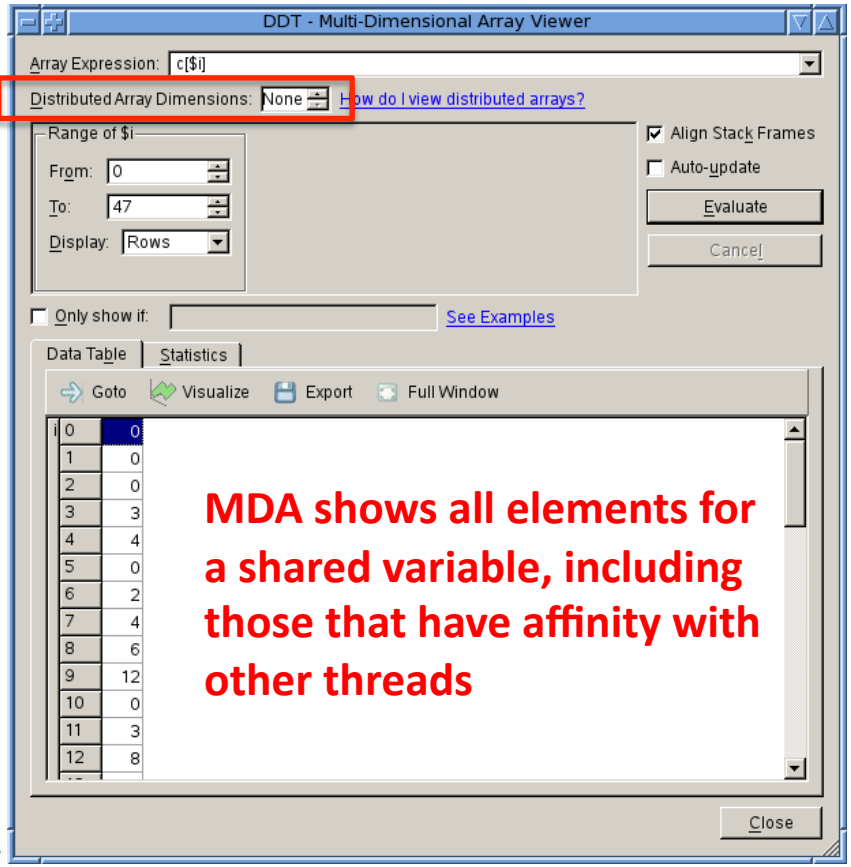
- Similarly, set 'MPI/UPC Implementation' to 'Cray XT/XE/XK (UPC)'
- "UPC Thread" = Process



Debugging UPC Programs (cont'd)

- **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)**

'None' for 'Distributed Array Dimensions'



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

Compiler	Fortran	C, C++
PGI	% ftn -g -c prog.f % ftn -Bddt -o prog prog.o	% cc -g -c prog.c % cc -Bddt -o prog prog.o
GNU	% ftn -g -c prog.f % ftn -v -o prog prog.o # -v to get the last linker line	% cc -g -c prog.c % cc -v -o prog prog.o
	Rerun the command after inserting '-zmuldefs' after the command and putting '${DDT_LINK_DMALLOC}' *just before* '–lc': % /opt/gcc/4.6.1/snos/libexec/gcc/x86_64-suse-linux/4.6.1/collect2 -zmuldefs ... ${DDT_LINK_DMALLOC} -lc …	
Cray	% ftn –g –c prog.f % ftn –v –o prog prog.o # -v to get the last linker line	% cc –g –c prog.c % cc –v –o prog prog.o
	Do similarly as above: % /opt/cray/cce/7.4.4/cray-binutils/x86_64-unknown-linux-gnu/bin/ld -zmuldefs ... ${DDT_LINK_DMALLOC} -lc …	
Intel	% ftn –g –c prog.f % ftn –v –o prog prog.o # -v to get the last linker line	% cc –g –c prog.c % cc –v –o prog prog.o
	Do similarly as above. There are *two* locations to put ${DDT_LINK_DMALLOC} as there are two -lc's. % ld -zmuldefs ... ${DDT_LINK_DMALLOC} -lc ... ${DDT_LINK_DMALLOC} -lc …	
PathScale	% ftn -g -Wl,--export-dynamic -TENV:frame_pointer=ON -funwind-tables -c prog.f % ftn –v -Wl,--export-dynamic -TENV:frame_pointer=ON -funwind-tables -o prog prog.o	% cc -g -c prog.c % cc -v -o prog prog.o
	Do similarly as above: % /usr/lib64/gcc/x86_64-suse-linux/4.3/collect2 -zmuldefs ... ${DDT_LINK_DMALLOC} -lc …	

- **Deselect 'Preload the memory debugging library' in the Memory Debugging Option window**

Dynamically Linked Binary on Hopper for Memory Debugging

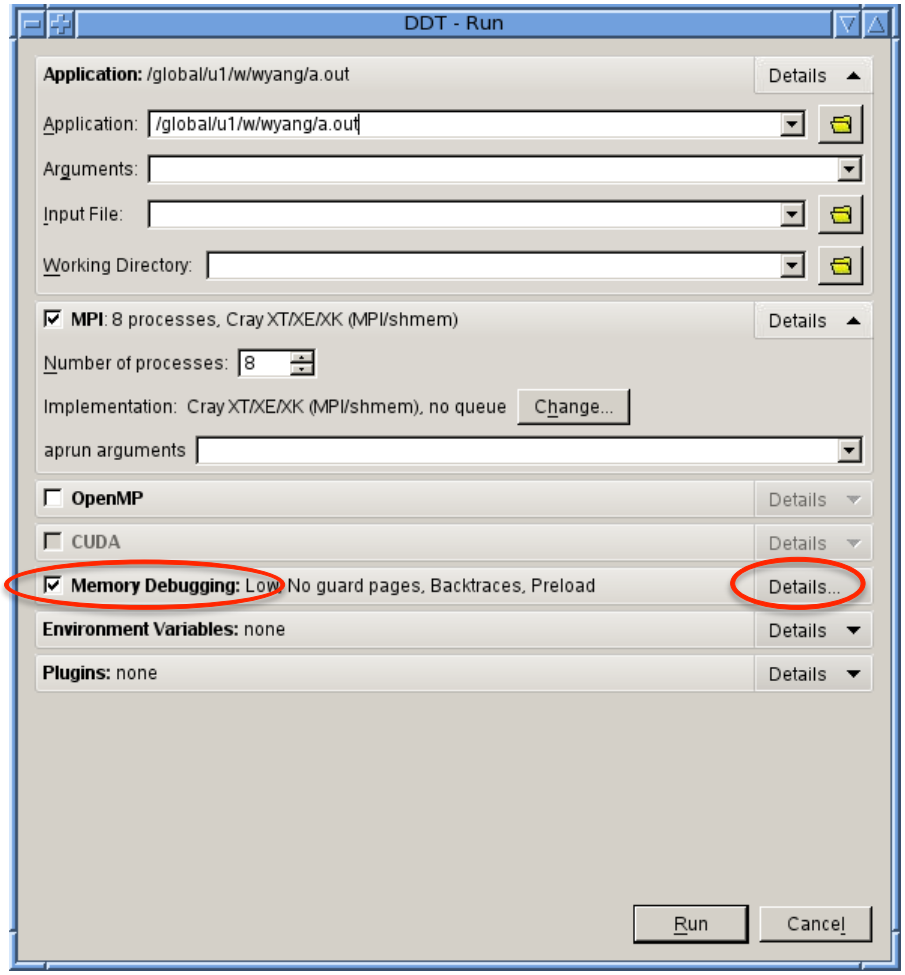
Compiler	Fortran	C, C++
PGI, Cray	% ftn -g -c prog.f % ftn –dynamic -o prog prog.o \ ${DDT_LINK_DMALLOC} –Wl,--allow-multiple-definition	% cc -g -c prog.c % cc -dynamict -o prog prog.o \ ${DDT_LINK_DMALLOC} –Wl,--allow-multiple-definition
GNU, Intel	% ftn -g -c prog.f % ftn –dynamic –o prog prog.o \ ${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
```

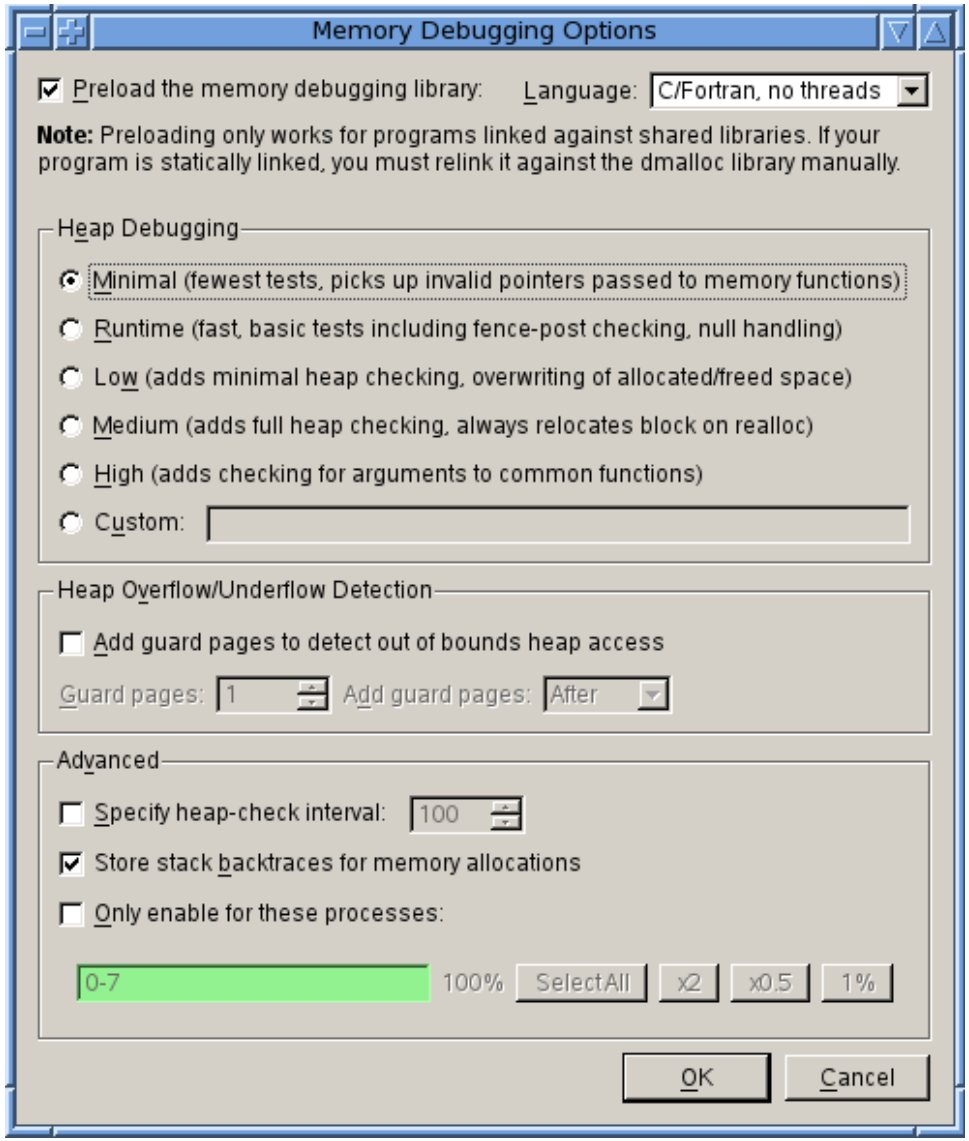
Starting Memory Debugging

- Enable 'Memory Debugging' in the Run window
- Clicking on 'Details' opens the 'Memory Debugging Options' window for setting options (next slide)



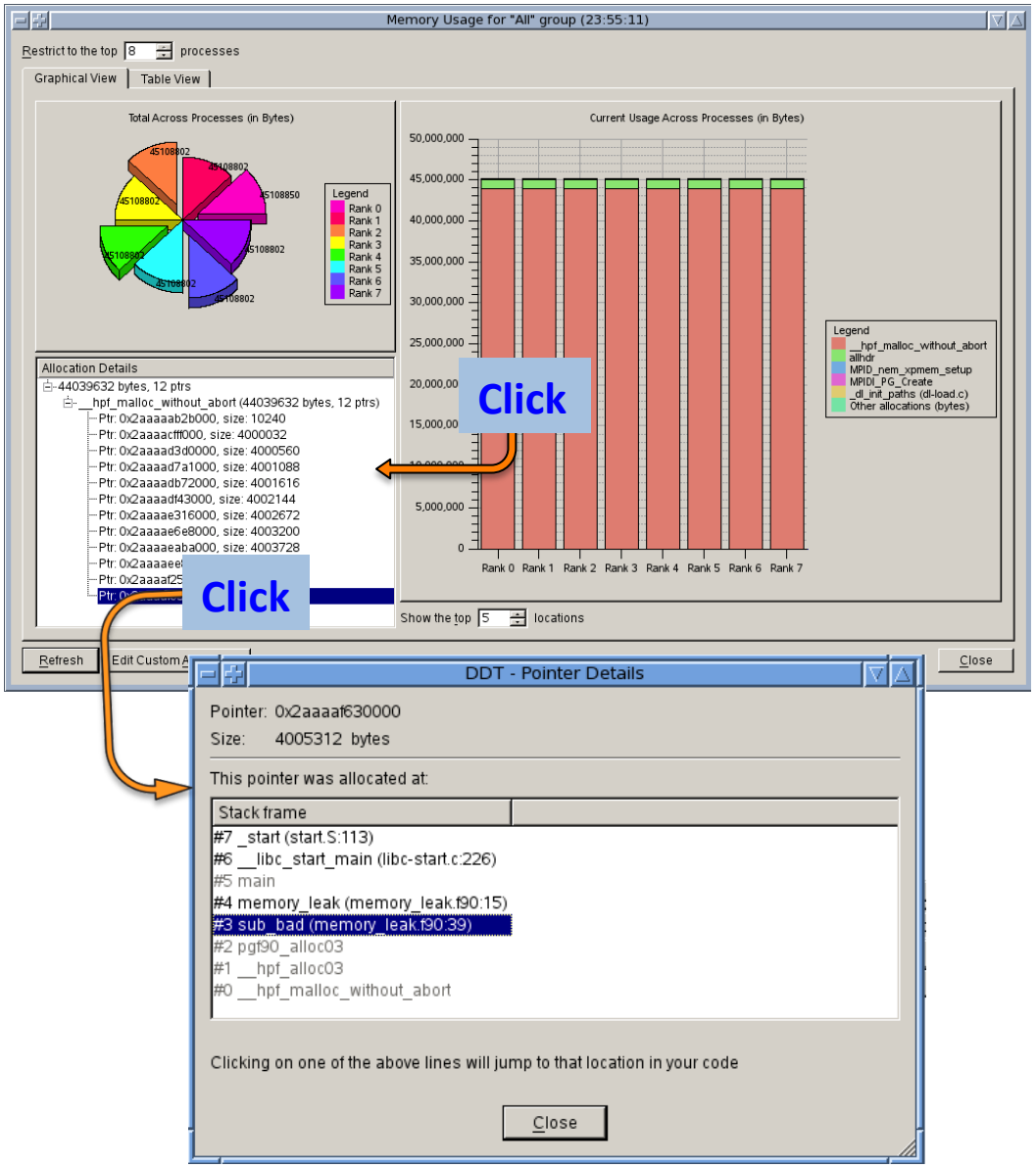
Memory Debugging Options

- **'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



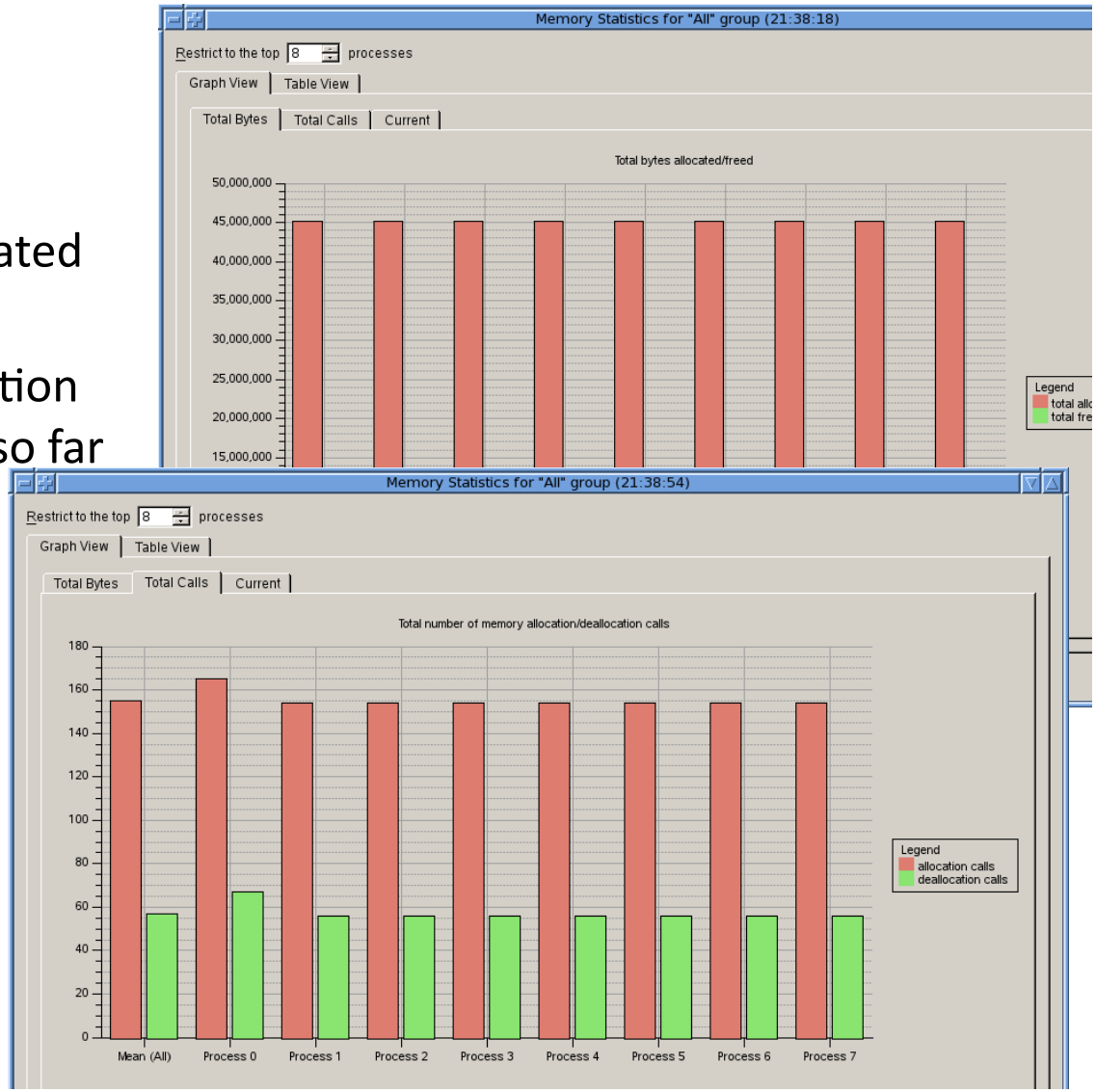
Current Memory Usage

- **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



Overall Memory Stats

- **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



U.S. DEPARTMENT OF ENERGY | Office of Science

Detecting Memory Leaks - Example

```
1       program memory_leak
2 !...  Buggy code prepared for a debugger tutorial.
3 !...  NERSC
4       implicit none
5       include 'mpif.h'
6       integer, parameter :: n = 1000000
7       real val
8       integer i, ierr
9       call mpi_init(ierr)
10      val = 0.
11      do i=1,10
12         call sub_ok(val,n)
13      end do
14      do i=1,10
15         call sub_bad(val,n)
16      end do
17      do i=1,10
18         call sub_badx2(val,n)
19      end do
20      print *, val
21      call mpi_finalize(ierr)
22      end
23
```

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

```
24      subroutine sub_ok(val,n)
25      integer n
26      real val
27      real, allocatable :: a(:)
28      allocate (a(n))
29      call random_number(a)
30      val = val + sum(a)
31 !    deallocate(a)              ! ok not to deallocate
32      end
33
34      subroutine sub_bad(val,n)
35      integer n
36      real val
37      real, pointer :: a(:)
38      allocate (a(n))
39      call random_number(a)
40      val = val + sum(a)
41 !    deallocate(a)              ! not ok not to deallocate
42      end
43
44      subroutine sub_badx2(val,n)
45      integer n
46      real val
47      real, pointer :: a(:)
48      allocate (a(n))
49      call random_number(a)
50      val = val + sum(a)
51      allocate (a(n))            ! not ok to allocate again
52      call random_number(a)
53      val = val + sum(a)
54 !    deallocate(a)              ! not ok not to deallocate
55      end
```

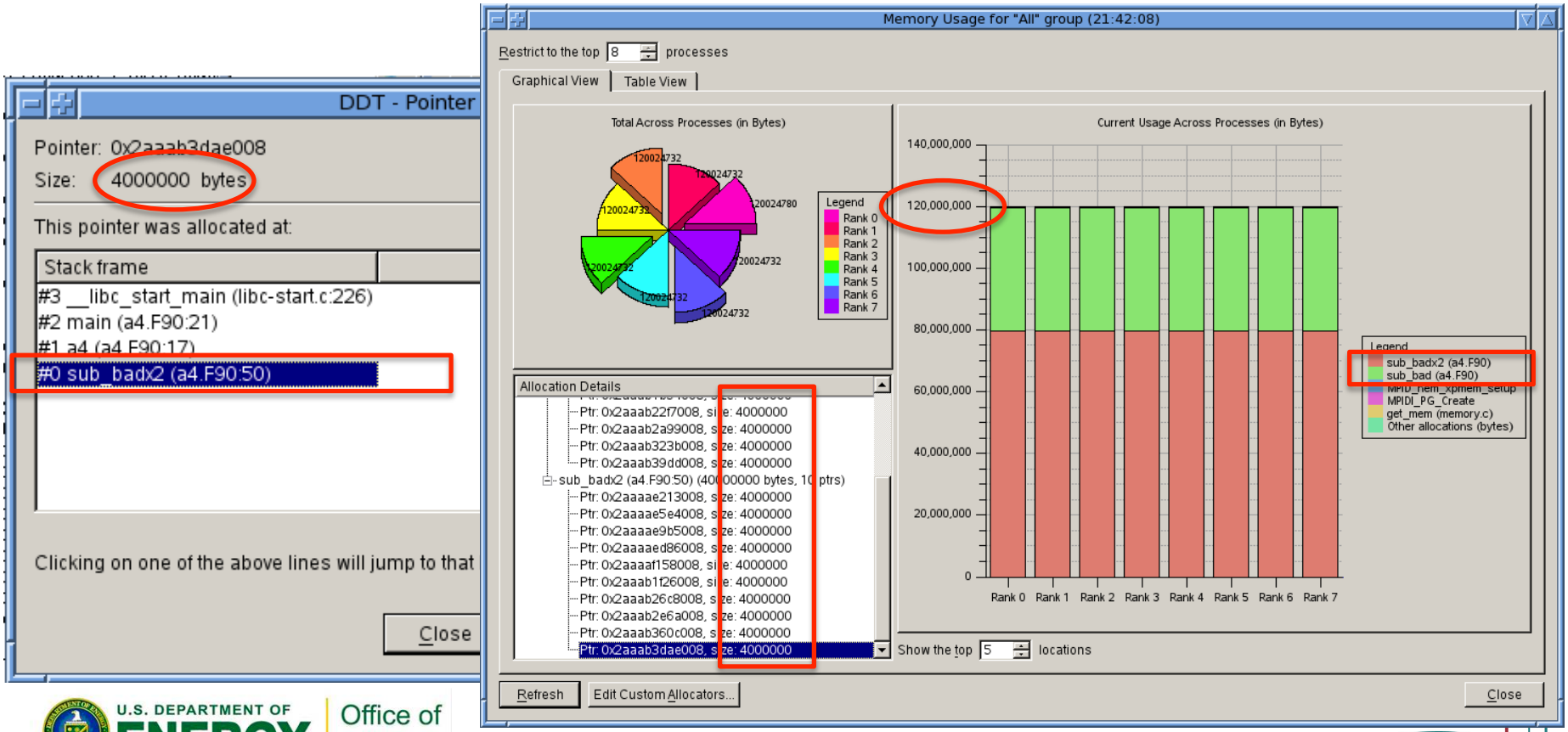
no memory leak

memory leak of 4*n bytes per call

memory leak of 8*n bytes per call

Detecting Memory Leaks - Example (cont'd)

- **'Current Memory Usage' at, for example, line 20 (gcc/4.6.2, xt-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)

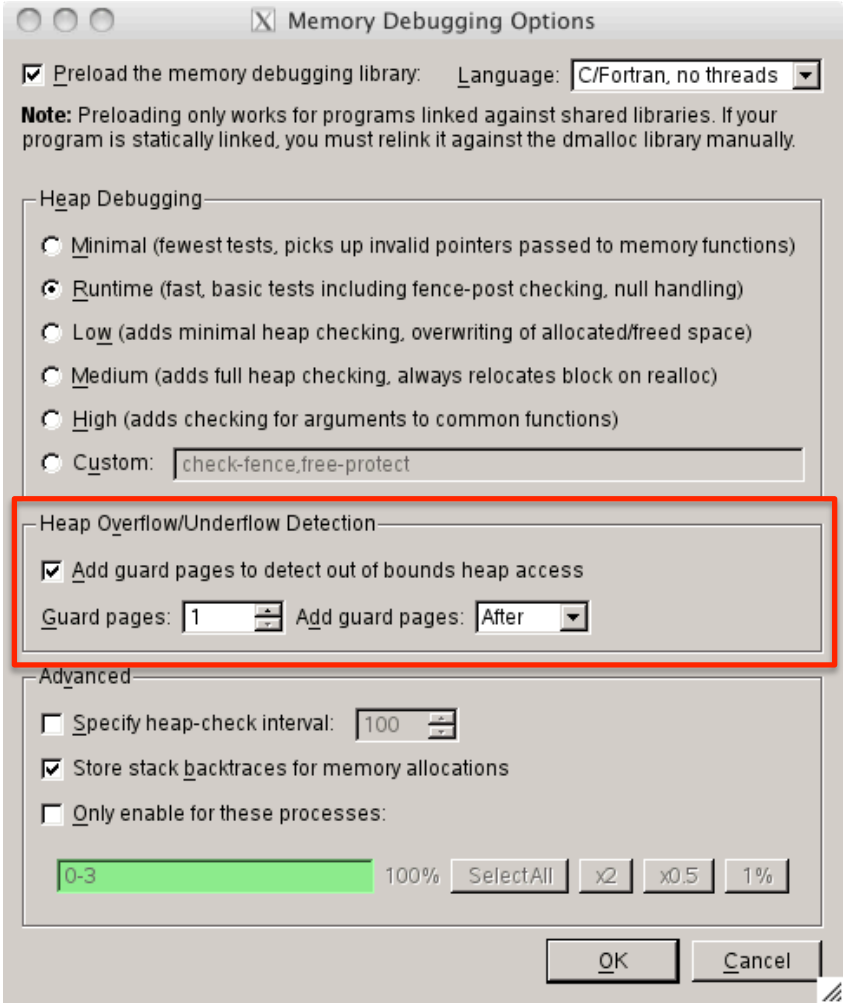
- **'Overall Memory Stats' results for rank 0 for a 8-PE run on Hopper (gcc/4.6.2, xt-mpich2/5.4.1, …)**

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

- **Memory leak of 4 million bytes after each call to sub_bad**
- **Memory leak of 8 million bytes after each call to sub_badx2**

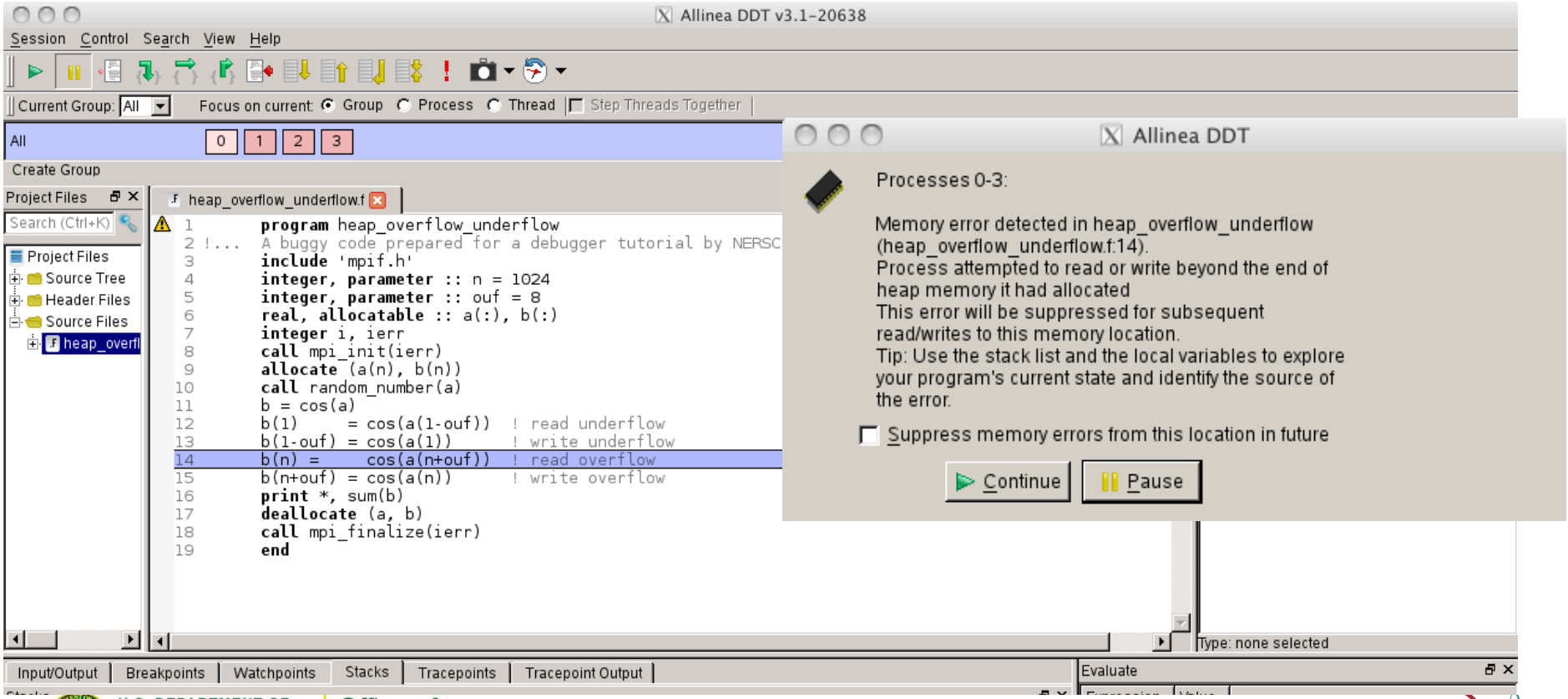
Heap Overflow/Underflow Detection

- **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



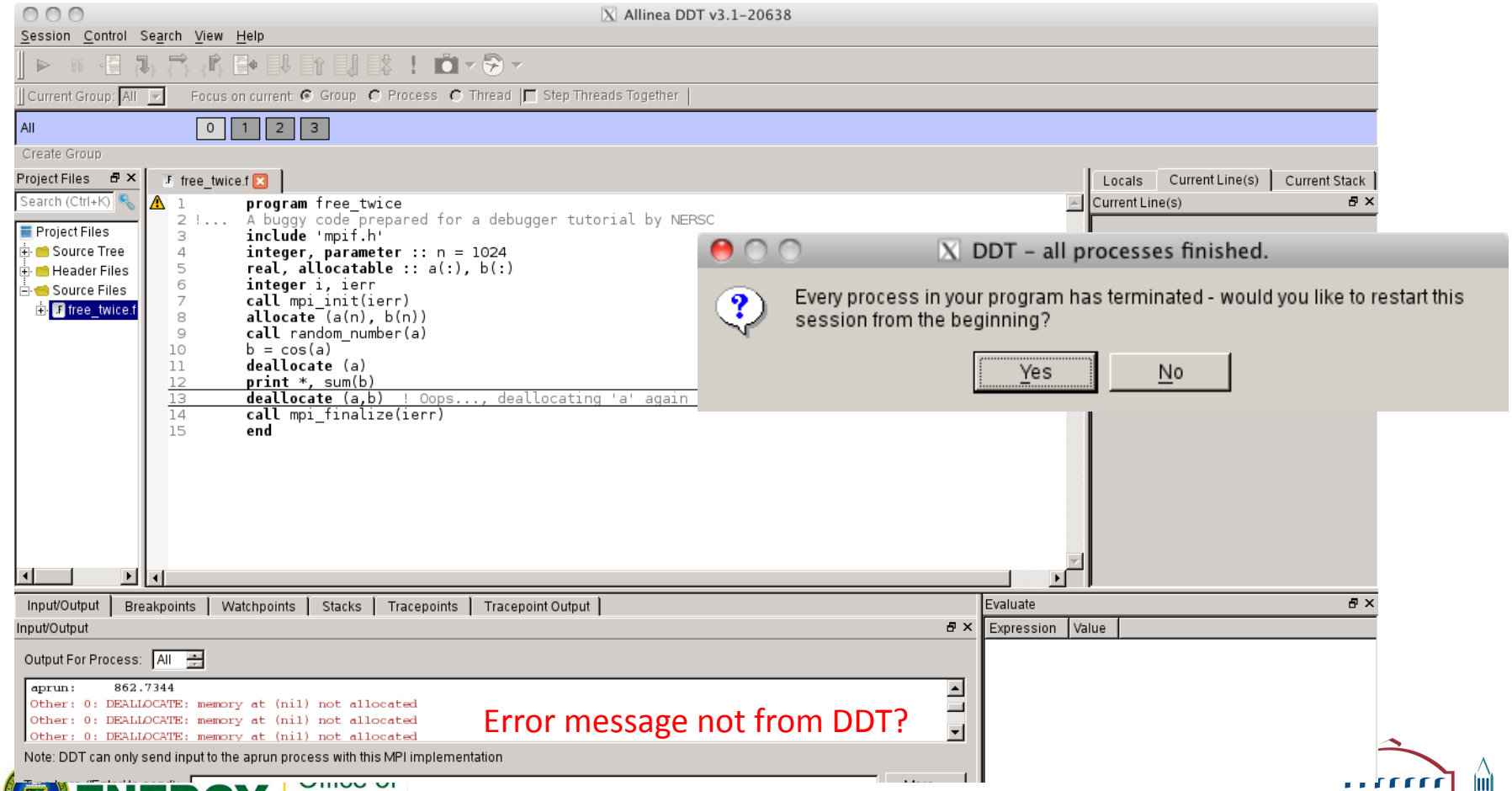
Heap Overflow/Underflow Detection Example

- 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



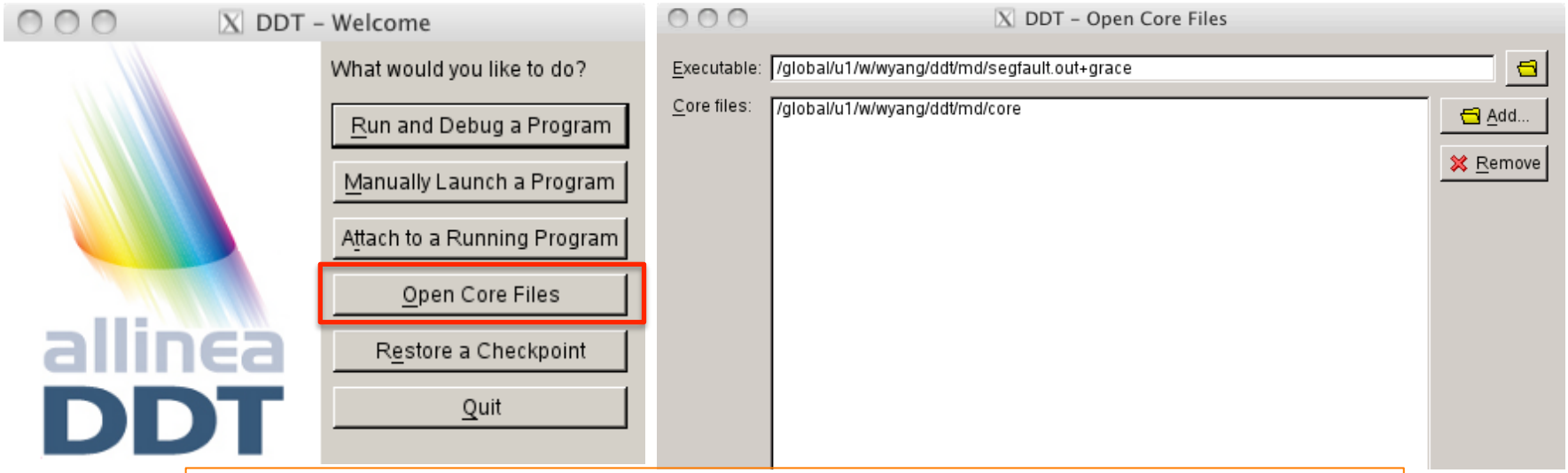
More Memory Debugging Example

- 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

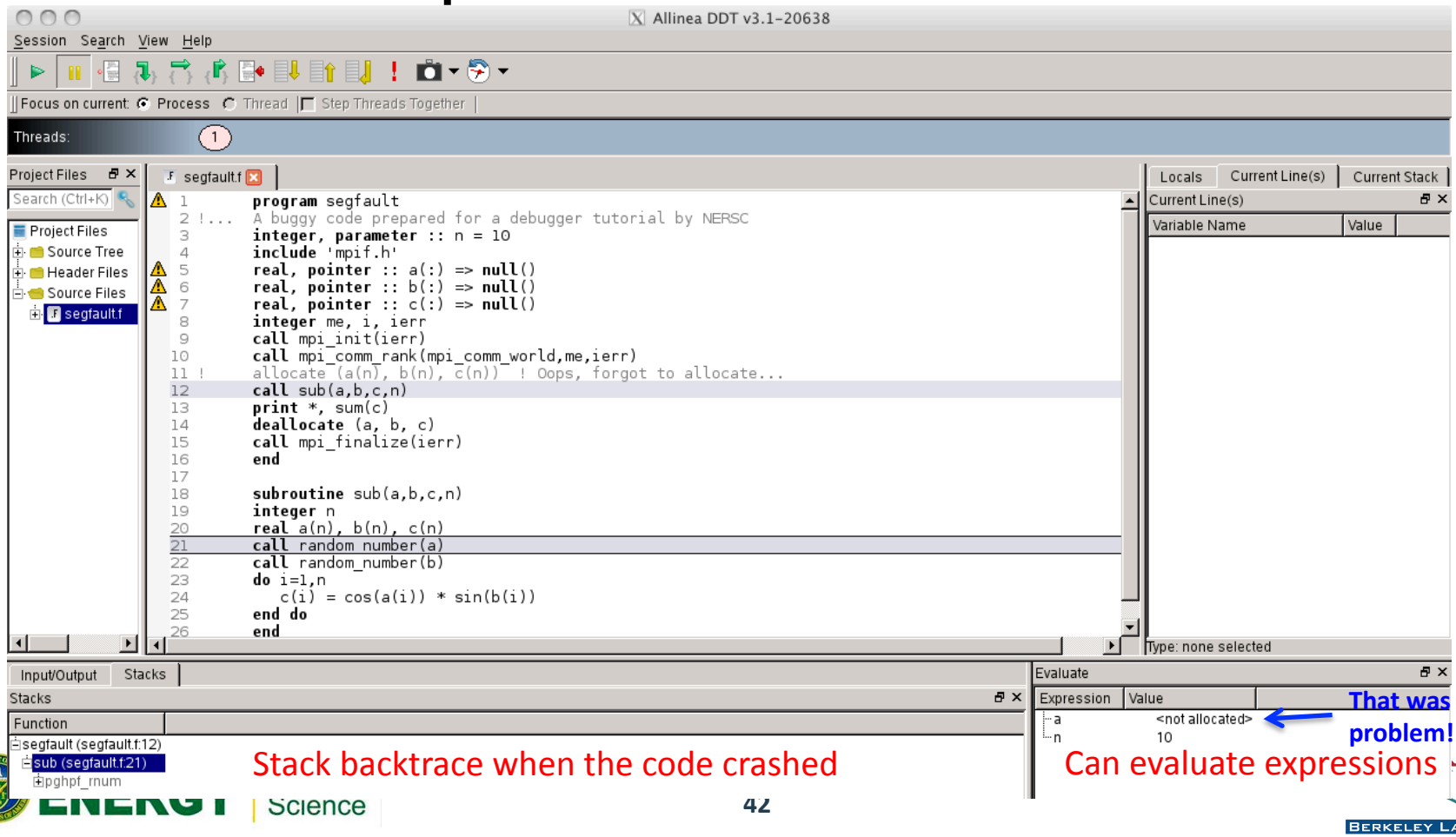
- Set coredumpsize to unlimited

```
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



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**

```
#!/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

```
% cat -n offline.f
     1        program offline
              ...
    10        do i=1,n
    11           if (mod(i,2) == 1) call sub(i,n,a)
    12        end do
              ...
    16        end
    17
    18        subroutine sub(i,n,a)
              ...
    21        do j=1,n
    22           a(j) = cos(a(j))
    23        end do
    24        end
```

```
ddt -offline offline.html -n 4 \
    -trace-at sub,i,a\(1\) \
    -break-at offline.f:22,5:3:2 a.out
```

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

Allinea DDT Offline Log

Click here for details

Messages | Tracepoints | Output

Messages

[+] Expand All [-] Collapse All

#	Type	Time	Processes	Message
1		00:00.001	n/a	Launching program /global/u1/w/wyang/ddt/od/offline.out+carver.
2		00:05.946	0-3	Startup complete. ▶ Stacks
3		00:08.466	0-3	Process stopped at breakpoint in sub (offline.f:22). ▶ Stacks ▶ Stack for process 0 ▶ Local variables for process 0 (ranges shown for 0-3)
4		00:10.991	0-3	Process stopped at breakpoint in sub (offline.f:22). ▶ Stacks ▶ Stack for process 0 ▶ Local variables for process 0 (ranges shown for 0-3)
5		00:11.046	n/a	Every process in your program has terminated.

i = 1 & j = 6

i = 1 & j = 9

Messages | Tracepoints | Output

Tracepoints

#	Time	Tracepoint	Processes	Values
1	00:06.756	sub	0-3	i: — 1 a(1): — 0.907922983
2	00:11.810	sub	0-3	i: — 3 a(1): — 0.615384221
3	00:11.810	sub	0-3	i: — 5 a(1): — 0.816551685
4	00:11.810	sub	0-3	i: — 7 a(1): — 0.684738338
5	00:11.810	sub	0-3	i: — 9 a(1): — 0.774584591
6	00:11.810	sub	0-3	i: — 11 a(1): — 0.714711666
7	00:11.810	sub	0-3	i: — 13 a(1): — 0.755282223
8	00:11.810	sub	0-3	i: — 15 a(1): — 0.728078127
9	00:11.810	sub	0-3	i: — 17 a(1): — 0.746454656
10	00:11.810	sub	0-3	i: — 19 a(1): — 0.734100938
11	00:11.810	sub	0-3	i: — 21 a(1): — 0.742433369
12	00:11.810	sub	0-3	i: — 23 a(1): — 0.736825585

Messages | Tracepoints | Output

Output

```
(00:11.823) Other: 2 17.70307
(00:11.823) Other: 0 17.70307
(00:11.823) Other: 1 17.70307
(00:11.823) Other: 3 17.70307
```

U.S. DEPARTMENT OF ENERGY | Office of Science

More Help?

- **User guide on each machine**
  - `$DDT_DOCDIR/userguide.pdf`

```
% module load ddt
% ls -l $DDT_DOCDIR/userguide.pdf
-rw-rw-r-- 1 wyang usg 2746551 Jan 11 12:12 /usr/common/usg/ddt/3.1-20638/doc/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

Hands-on Lab

- 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.