

Debugging with DDT

Woo-Sun Yang
NERSC User Services Group

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



National Energy Research
Scientific Computing Center





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

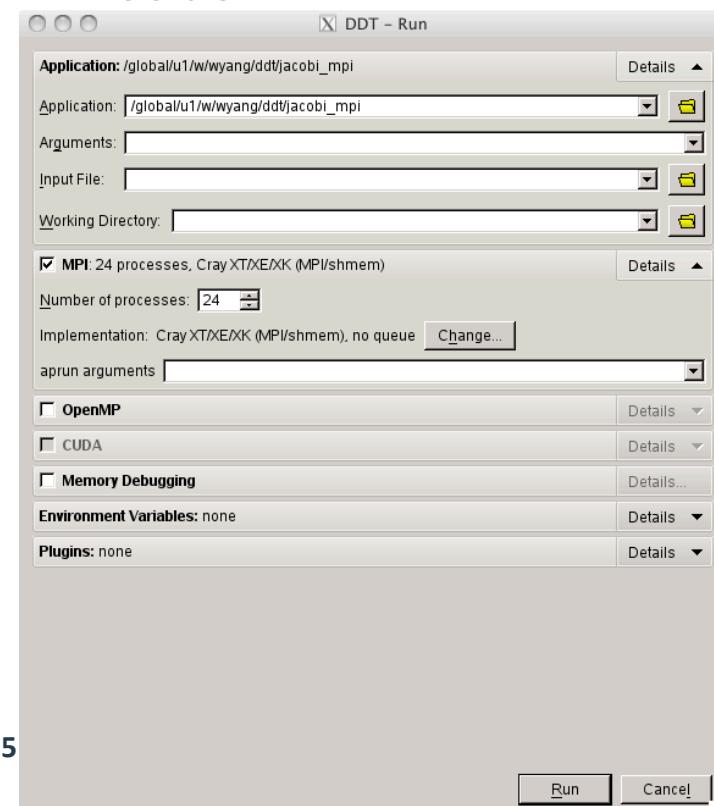
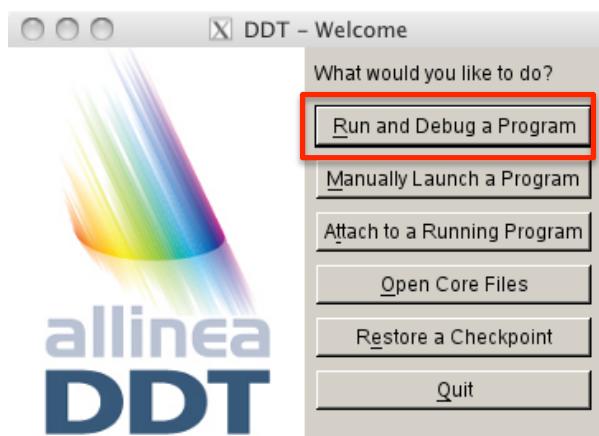
% ftn -g prog.f                      # Hopper/Franklin
% mpif90 -g prog.f                    # Carver

% 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

Can have multiple tabs

Source Code

• 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

Evaluation

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

Project Files Search (Ctrl+K)

Project Files Source Tree Header Files Source Files jacobi_mpi.f90

9 integer :: ngrid ! number of grid cells along each axis
10 integer :: n ! number of cells: n = ngrid - 1
11 integer :: maxiter ! max number of Jacobi iterations
12 real :: tol ! convergence tolerance threshold
13 real :: omega
14 integer i, j, k
15 real h, utm
16 integer np, myid
17 integer js, je, js1, je1
18 integer nbr_down, nbr_up, status(MPI_STATUS_SIZE), ierr
19
20 call MPI_Init(ierr)
21 call MPI_Comm_Size(MPI_Comm_World,np,ierr)
22 call MPI_Comm_Rank(MPI_Comm_World,myid,ierr) **Current line**
23
24 nbr_down = MPI_Proc_Null
25 nbr_up = MPI_Proc_Null
26 if (myid > 0) nbr_down = myid - 1
27 if (myid < np - 1) nbr_up = myid + 1
28
29 ! Read in problem and solver parameters.
30
31 call read_params(ngrid,maxiter,tol,omega)
32
33 n = ngrid - 1
34

Locals Current Line(s) Current Stack

Variable Name Value

ierr 0
MPI_Comm_World 1140850688
np 0

Current Line(s)

Type: none selected

Evaluate

Expression Value

Stacks

Processes Function

24 jacobi_mpi (jacobi_mpi.f90:21)

U.S. DEPARTMENT OF ENERGY SCIENCE BERKELEY LAB



U.S. DEPARTMENT OF ENERGY SCIENCE





Navigating Through a Program

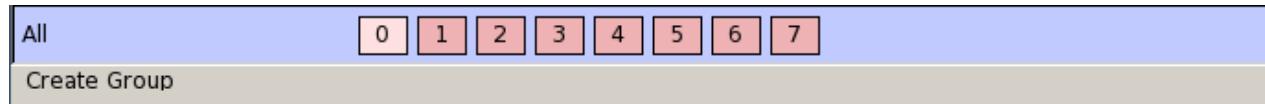


- **Play/Continue**
 - Run the program
- **Pause**
- **Step Into**
 - Move to the next line of source
 - If the next line is a function call, step to the first line of the function
- **Step Over**
 - Move to the next line of source code
 - If the next line is a function call, step over the function
-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

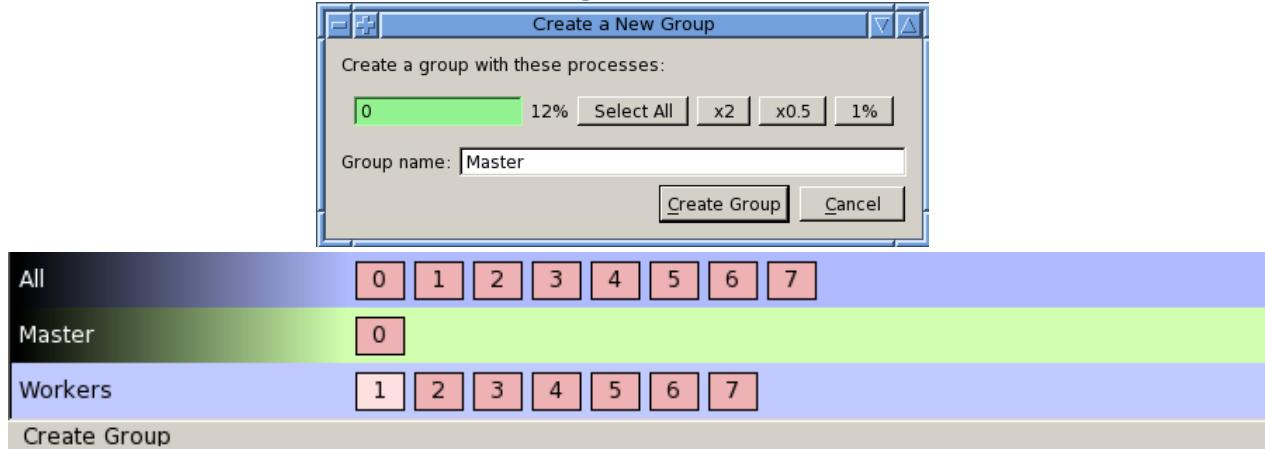


Process Groups

- 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

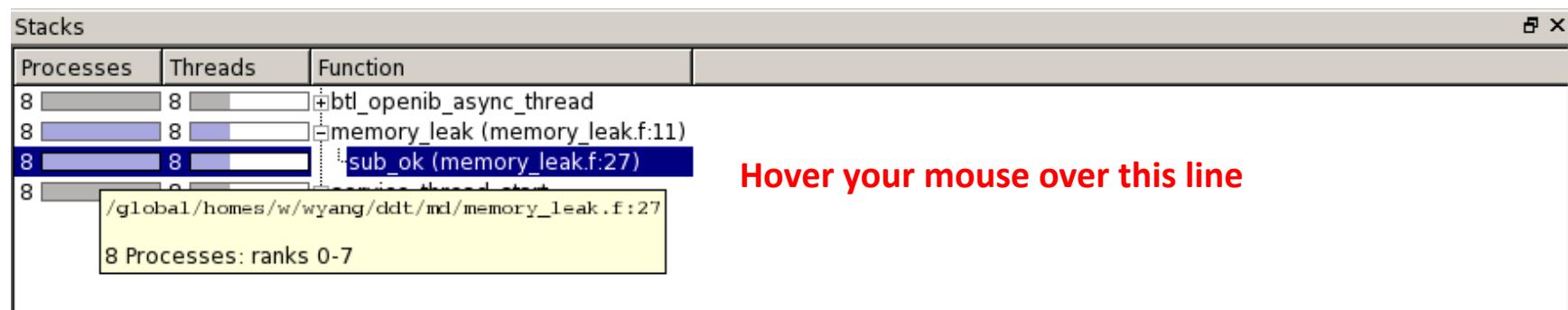


- Cannot create a thread group



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)

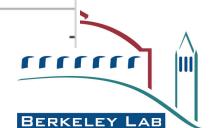


Breakpoints									
	Processes	Threads	File	Line	Function	Condition	Start After	Trigger Every	Stop After
<input checked="" type="checkbox"/>	All	all	memory_leak.f	29		val > 0.5	0	1	
<input checked="" type="checkbox"/>	All	all	memory_leak.f	39			0	1	
<input checked="" type="checkbox"/>	All	all	memory_leak.f	49			0	1	



U.S. DEPARTMENT OF
ENERGY

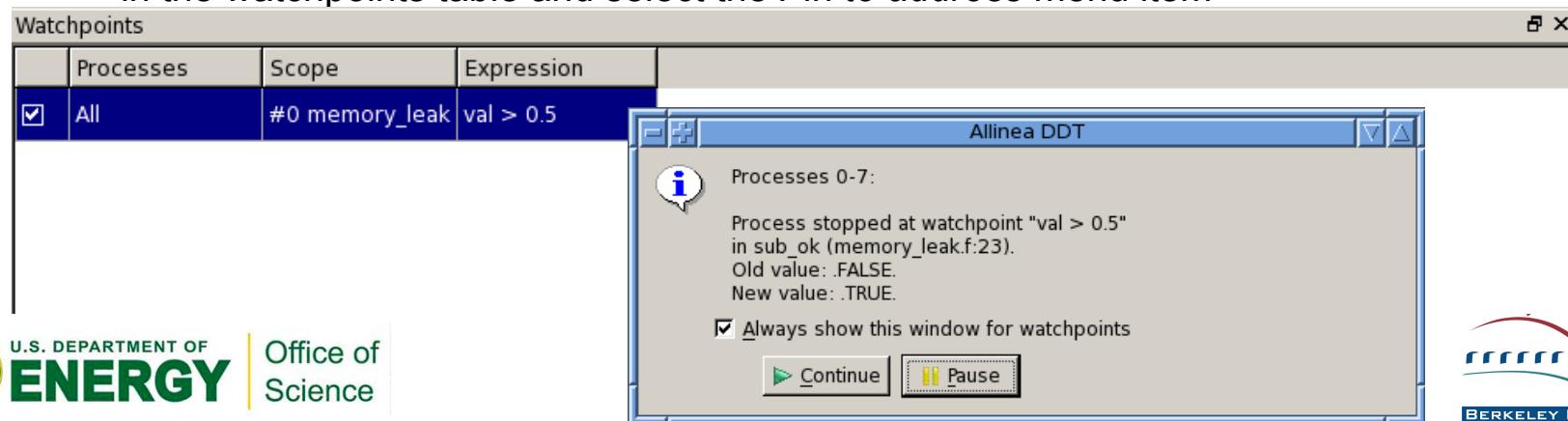
Office of
Science





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



U.S. DEPARTMENT OF
ENERGY

Office of
Science





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**

The screenshot shows two windows related to tracepoints. The top window is titled 'Tracepoints' and contains a table with the following data:

	Processes	Threads	File	Line	Function	Condition	Start After	Trigger Every	Stop After	Full p
<input checked="" type="checkbox"/>	All	all	memory_leak.f	23			0	1	-	/glob

The bottom window is titled 'Tracepoint Output' and contains a table with the following data:

Tracepoint	Processes	Values logged
memory_lea...	8, ranks 0-7	n: —— 1000000 val: —— 5.0e+05
memory_lea...	8, ranks 0-7	n: —— 1000000 val: —— 1.0e+06
memory_lea...	8, ranks 0-7	n: —— 1000000 val: —— 1.5e+06
memory_lea...	8, ranks 0-7	n: —— 1000000 val: —— 2.0e+06

At the bottom of the 'Tracepoint Output' window, there is a search bar labeled 'Only show lines containing:'.



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**

Locals	Current Line(s)	Current Stack
Current Line(s)		
Variable Name	Value	
<code>cosx</code>	1	
<code>me</code>	0	
<code>np</code>	-8	
<code>sinx</code>	0	
<code>x</code>	0	

Evaluate		
Expression	Value	
<code>cosx</code>	1	
<code>cosx+sinx</code>	1	
<code>sinx</code>	0	
<code>x</code>	0	

Sparklines



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

Expression	Value
└ c_array	
└ [0]	
└ [0]	0
└ [1]	1
└ [2]	2
└ [3]	3
└ [4]	4
└ [1]	
└ [2]	

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

Expression	Value
└ f_array	
└ [1]	
└ [1]	11
└ [2]	21
└ [3]	31
└ [2]	
└ [3]	
└ [4]	
└ [5]	

Fortran:
f_array(3,5)



U.S. DEPARTMENT OF
ENERGY

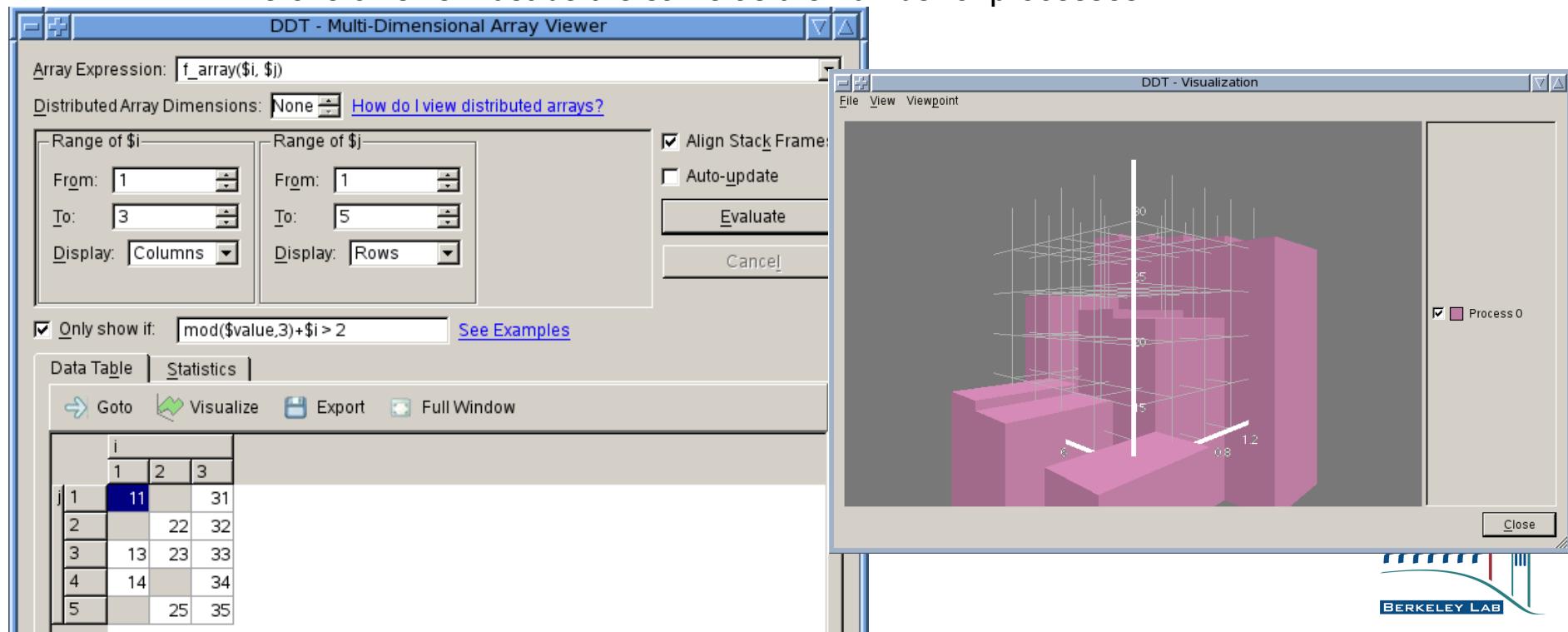
Office of
Science





MDA (Multi-dimensional Array)

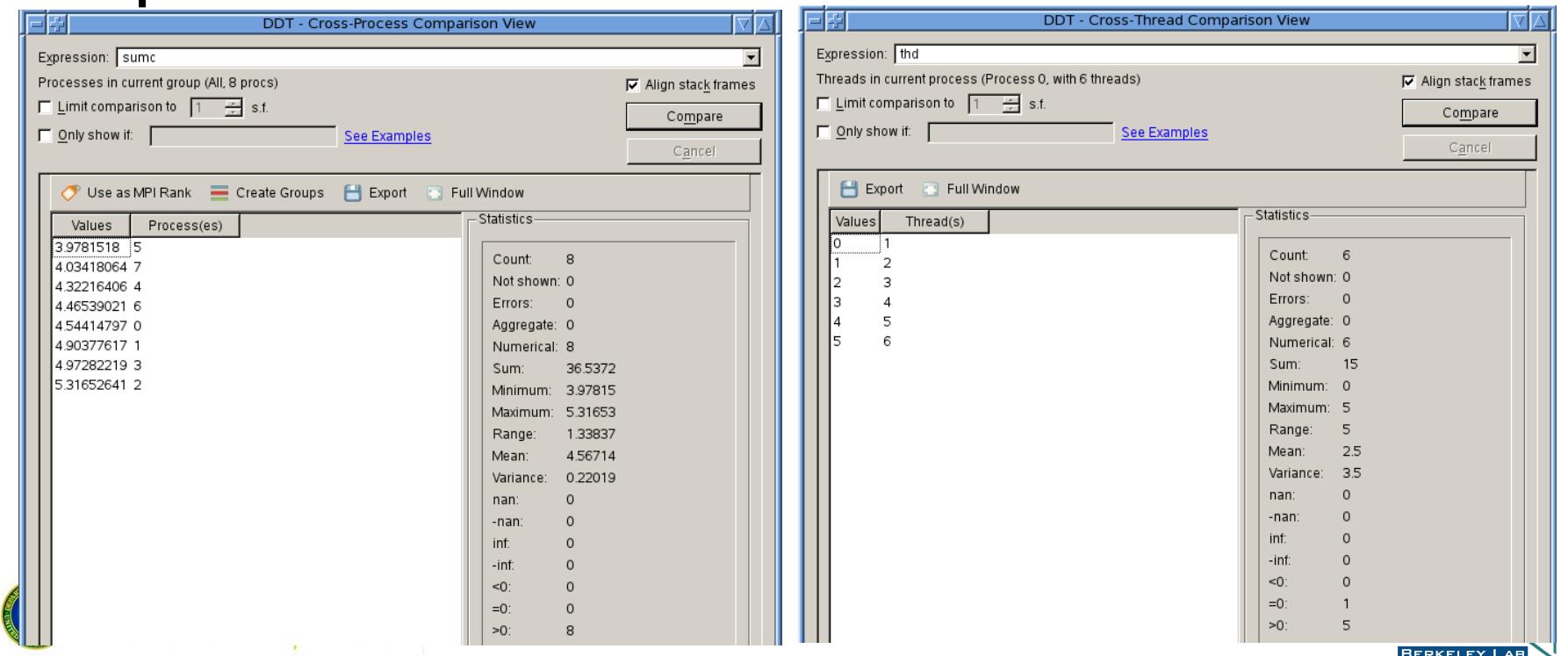
- Right click on an array and select ‘View Array (MDA)’
- Index order: $\$i, \j, \dots , same as what’s used in the code
- Can filter by value: use an expression using $\$value, \$i, \$j$ in the *Only show if* window
- Can visualize the array (showing only the filtered values)
- Can show statistics (for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

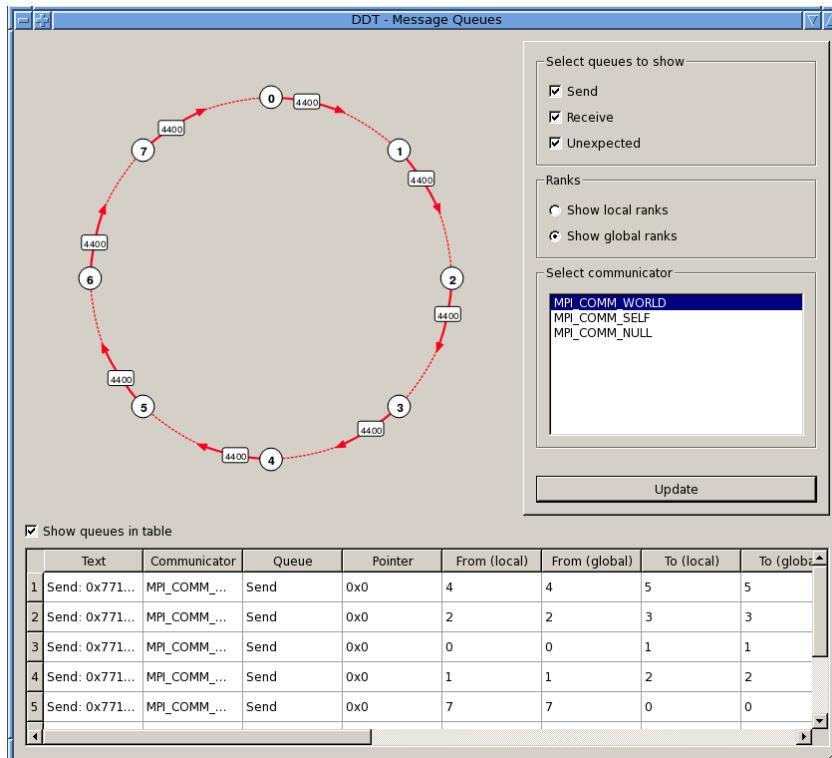
- **Examining status of the internal MPI message buffers when a program hangs**
 - Can be helpful in detecting deadlock
- **Message queue debugging only available on Carver**
- **Three queues can be examined**
 - Send Queue:
 - Calls to MPI send not yet completed
 - **Red arrow**
 - Receive Queue:
 - 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

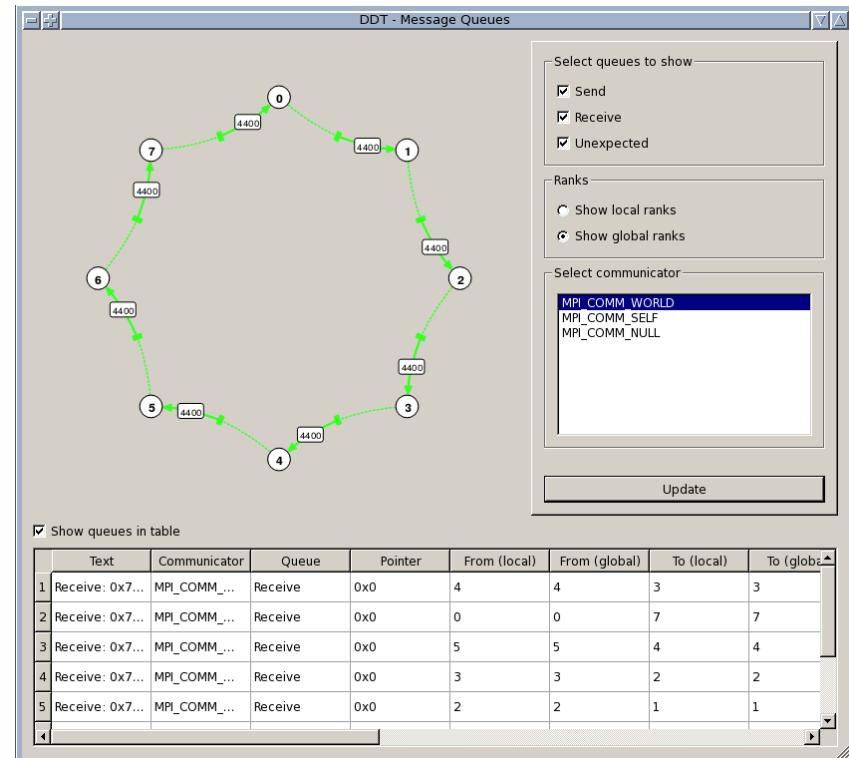
deadlock with a large message

```
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



U.S. DEPARTMENT OF
ENERGY

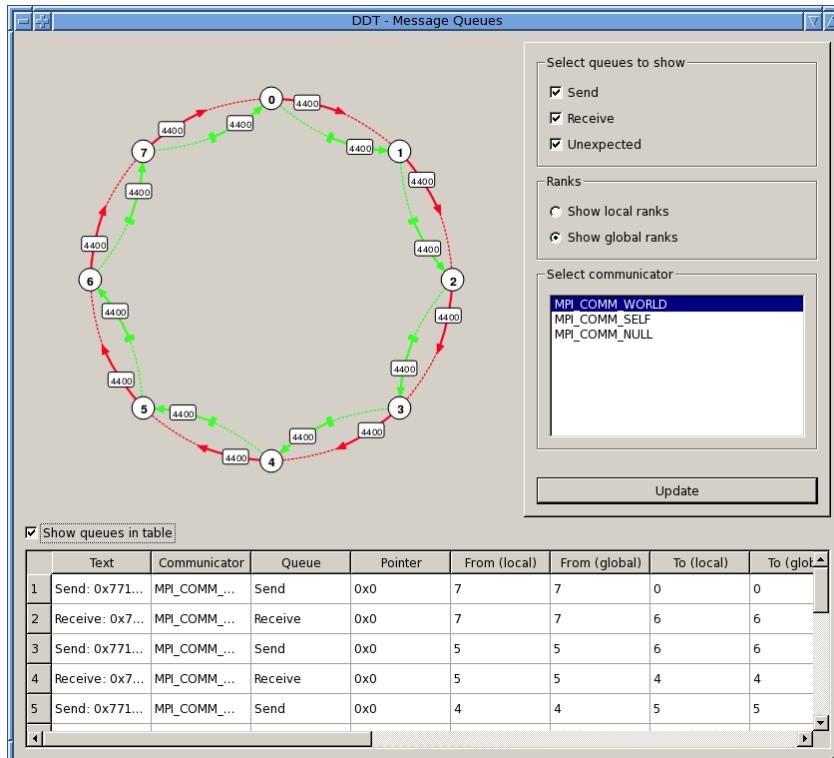
Office of
Science



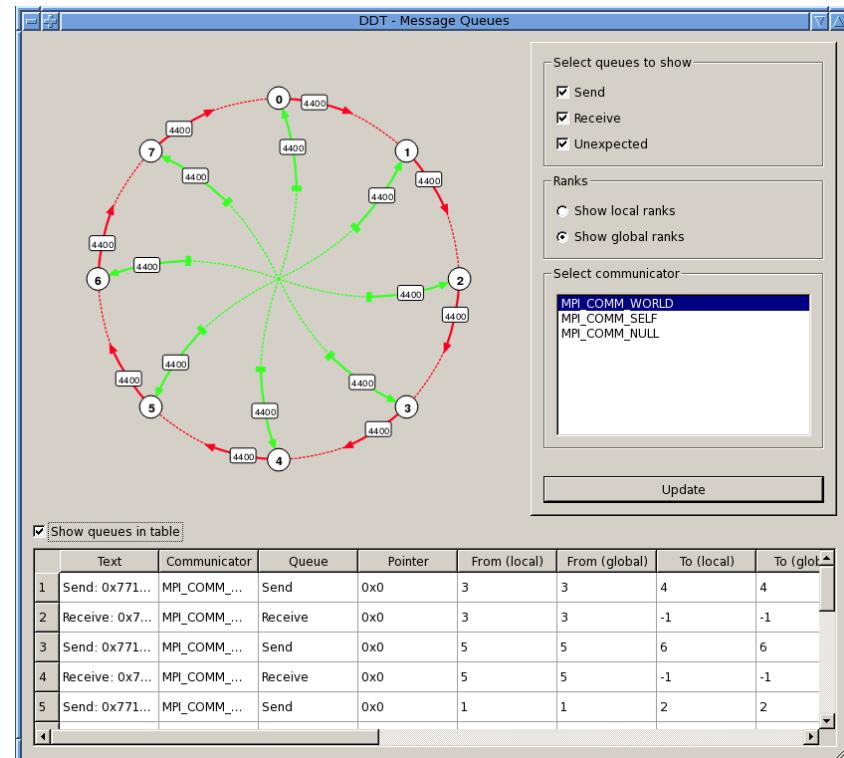
Message Queues Examples (cont'd)

```
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),ier)
call mpi_waiall(2,req,stat,ierr)
```

```
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_waiall(2,req,stat,ierr)
```



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

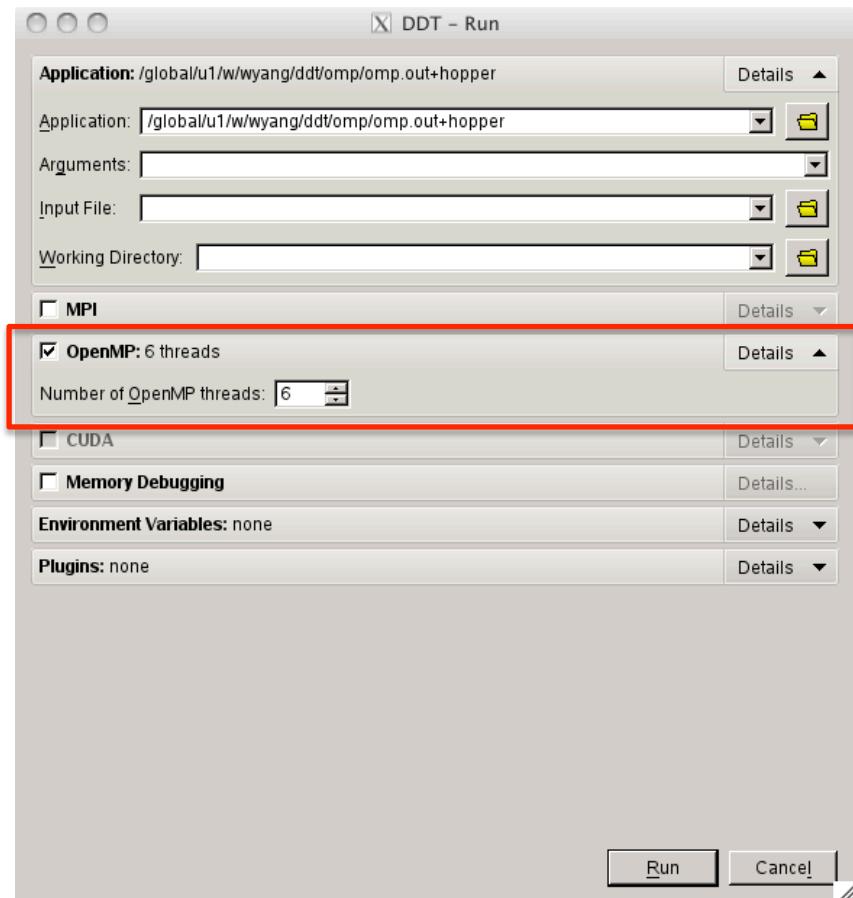


Receive queue messages look ambiguous because of use of MPI_ANY_SOURCE



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 you 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

The screenshot shows the DDT (Deja Vu Debugger) interface. On the left, there's a project tree for 'omp.f'. In the center, the code editor shows a Fortran program with a parallel region. A context menu is open over line 10, which contains the instruction `c(i) = cos(a(i)) * sin(b(i))`. The menu options include 'Add breakpoint', 'Add tracepoint (c, i, cos, a, sin, b)', 'Add tracepoint (c, i, cos, a, sin, b, n)', 'Delete breakpoint for process 0', 'Edit breakpoint', 'Run to here', 'Copy To Clipboard', 'Split view', 'Open file in editor', and 'Close'. The status bar at the bottom says 'Hover the mouse over to see thread IDs'.

Threads: 1 2 3 4 5 6

Focus on current: Process Thread Step Threads Together

omp.f

```
program omp
integer, parameter :: n = 12
real, allocatable :: a(:, b(:, c(:)
integer i
allocate (a(n), b(n), c(n)
call random_number(a)
call random_number(b)
 !$omp parallel do
 do i=1,n
   c(i) = cos(a(i)) * sin(b(i))
 end do
 !$omp end parallel do
 print *, sum(c)
 deallocate(a,b,c)
end
```

Add breakpoint
Add tracepoint (c, i, cos, a, sin, b)
Add tracepoint (c, i, cos, a, sin, b, n)
Delete breakpoint for process 0
Edit breakpoint
Run to here
Copy To Clipboard
Split view
Open file in editor
Close

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

Threads	Function
1	main
1	+_mp_slave
5	omp (omp.f:10)

Locals Current Line(s) Current Stack

Variable Name	Value
a	
b	
c	[1] 0 [2] 0 [3] 0 [4] 0 [5] 0.41047397 [6] 0.49938094 [7] 0.48871362 [8] 0 [9] 0.45540645 [10] 0 [11] 0 [12] 0 i 8

Evaluate

Expression	Value
c	[1] 0 [2] 0 [3] 0 [4] 0 [5] 0.41047397 [6] 0.49938094



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

The screenshot shows the Allinea DDT v3.1.2-20638 debugger interface. On the left, the 'Run' window displays the application path as /global/u1/w/wyang/ddt/omp/mpioomp.out+hopper, arguments as /global/u1/w/wyang/ddt/omp/mpioomp.out+hopper, and working directory as /global/u1/w/wyang. It includes sections for MPI settings (4 processes, Cray XT/XE/XK (MPI/shmem), checked) and OpenMP settings (6 threads, checked). A red box highlights these settings. The main window shows the source code of 'mpioomp.f' with MPI and OpenMP directives. The code initializes MPI, allocates arrays 'a', 'b', and 'c', generates random numbers for 'a' and 'b', and performs a parallel loop over 'i' from 1 to n, calculating 'c(i) = cos(a(i)) * sin(b(i))'. A callout box points to the MPI tasks and OpenMP threads, stating: '4 MPI tasks with 6 OpenMP threads each; Rank 0 and its thread 3 in focus'. The right side of the interface shows the 'Locals' and 'Current Line(s)' panes, and a bottom pane for evaluating expressions.

Application: /global/u1/w/wyang/ddt/omp/mpioomp.out+hopper

Arguments: /global/u1/w/wyang/ddt/omp/mpioomp.out+hopper

Input File:

Working Directory:

MPI: 4 processes, Cray XT/XE/XK (MPI/shmem)

Number of processes: 4

Implementation: Cray XT/XE/XK (MPI/shmem), no queue

OpenMP: 6 threads

Number of OpenMP threads: 6

CUDA

Memory Debugging

Environment Variables: none

Plugins: none

Allinea DDT v3.1.2-20638

Session Control Search View Help

Current Group: All Focus on current: Group Process Thread Step Threads Together

[All] 0 1 2 3

[Process 0's threads:] 1 2 3 4 5 6

mpiomp.f

```
program mpiomp
include 'mpif.h'
integer, parameter :: n = 12
real, allocatable :: a(:, ), b(:, ), c(:, )
integer i, me
call mpi_init(ierr)
call mpi_comm_rank(MPI_COMM_WORLD, me, ierr)
allocate (a(n), b(n), c(n))
call random_number(a)
call random_number(b)
!$omp parallel do
do i=1,n
    c(i) = cos(a(i)) * sin(b(i))
end do
!$omp end parallel do
print *, me, sum(c)
deallocate(a,b,c)
call mpi_finalize(ierr)
end
```

Add breakpoint for thread 0.3
Add tracepoint for thread 0.3 (c, i, cos, a, sin, b)
Add tracepoint for thread 0.3 (3)

Run to here

Copy To Clipboard
Split view
Open file in editor
Close

Evaluate
Expression Value

c

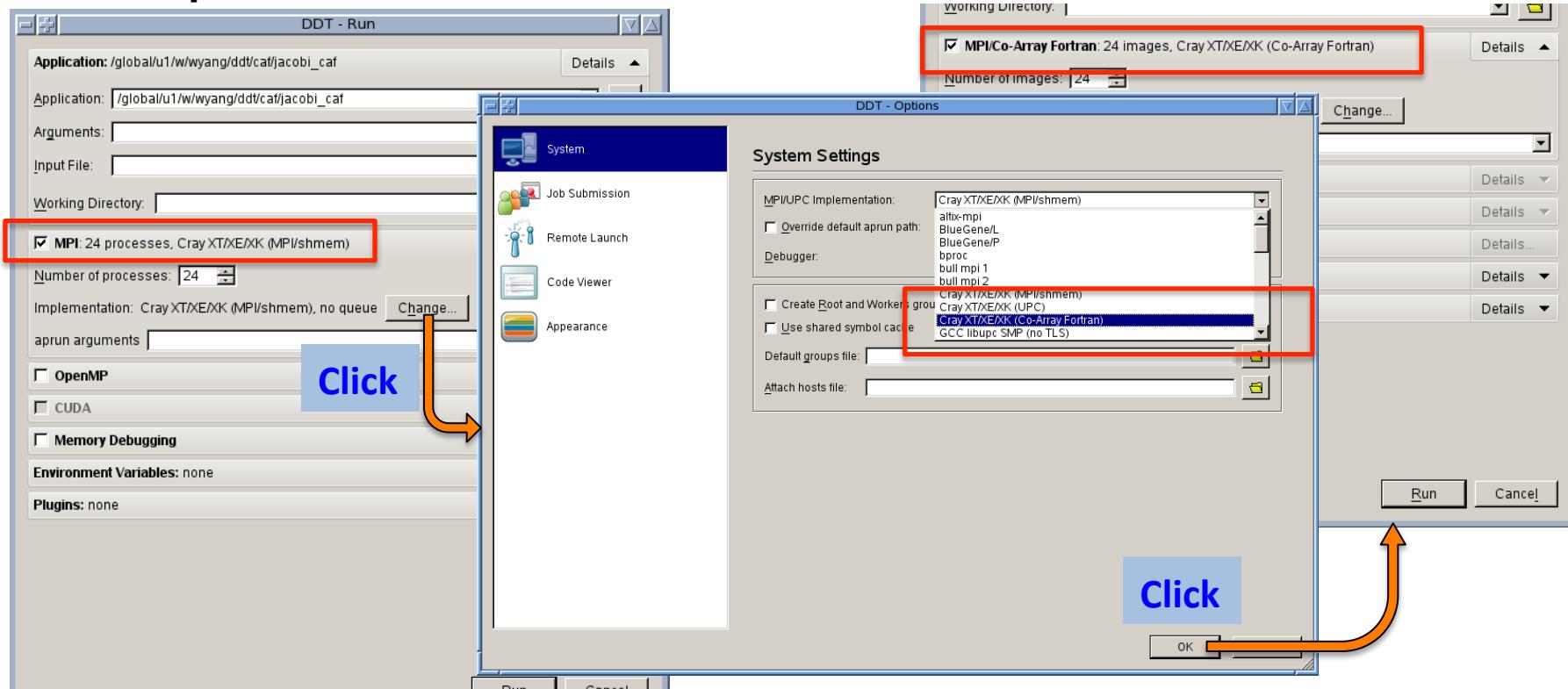
1	0
2	0
3	0
4	0
5	0
6	0
7	0.488713622

U.S. DEPARTMENT OF ENERGY Office of Science



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 can be used for coarrays – used automatically for static coarrays (but not for allocatable coarrays)



Debugging UPC Programs

- 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`)

The screenshot shows the Allinea DDT v3.1-20638 debugger interface. The main window displays the source code for `axb.c`:

```
10      a[i] = i / 3;
11      b[i] = i % 5;
12  }
13  upc_barrier;
14 }
15
16
17 int main() {
18     int i;
19
20     set_b_c();
21
22     upc_forall (i=0; i<N; i++) {
23         c[i] = a[i] + b[i];
24     }
25     upc_barrier;
26
27     if (MYTHREAD == 0) printf("sum1: %d\n", c[0] + c[N-1]);
28
29     return 0;
30 }
```

The code editor has tabs for `Input/Output`, `Breakpoints`, `Watchpoints`, `Stacks`, `Tracepoints`, and `Tracepoint Output`. The `Stacks` tab shows:

Stacks	UPC Thread	Native Thread	Function
1	1		<code>_dmappi_error_handler (dmapp_init.c:686)</code>
1	1		<code>_dmappi_error_handler (dmapp_init.c:698)</code>
2	2		<code>_dmappi_error_handler (dmapp_init.c:704)</code>
4	4		<code>main (axb.c:25)</code>

The `Evaluate` window on the right shows the values of arrays `a`, `b`, and `c`:

Expression	Value
<code>a</code>	
<code>b</code>	
<code>c</code>	
<code>[0]</code>	0
<code>[1]</code>	1
<code>[2]</code>	2
<code>[3]</code>	4
<code>[4]</code>	5
<code>[5]</code>	1
<code>[6]</code>	3

At the bottom left is the U.S. Energy Department logo, and at the bottom right is the Cray Lab logo.



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 Linked 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/snobs/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 -WI,--export-dynamic -TENV:frame_pointer=ON -funwind-tables -c prog.f % ftn -v -WI,--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} -WI,--allow-multiple-definition	% cc -g -c prog.c % cc -dynamict -o prog prog.o \ \${DDT_LINK_DMALLOC} -WI,--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 -WI,--export-dynamic -TENV:frame_pointer=ON \ -funwind-tables -c prog.f % ftn -dynamic \ -WI,--export-dynamic -TENV:frame_pointer=ON \ -funwind-tables -o prog prog.o \ \${DDT_LINK_DMALLOC} -WI,--allow-multiple-definition	% cc -g -c prog.c % cc -dynamic -o prog prog.o \ \${DDT_LINK_DMALLOC} -WI,--allow-multiple-definition

- ‘Preload the memory debugging library’ in the Memory Debugging Option window

- Before running ddt

- Load the same PrgEnv

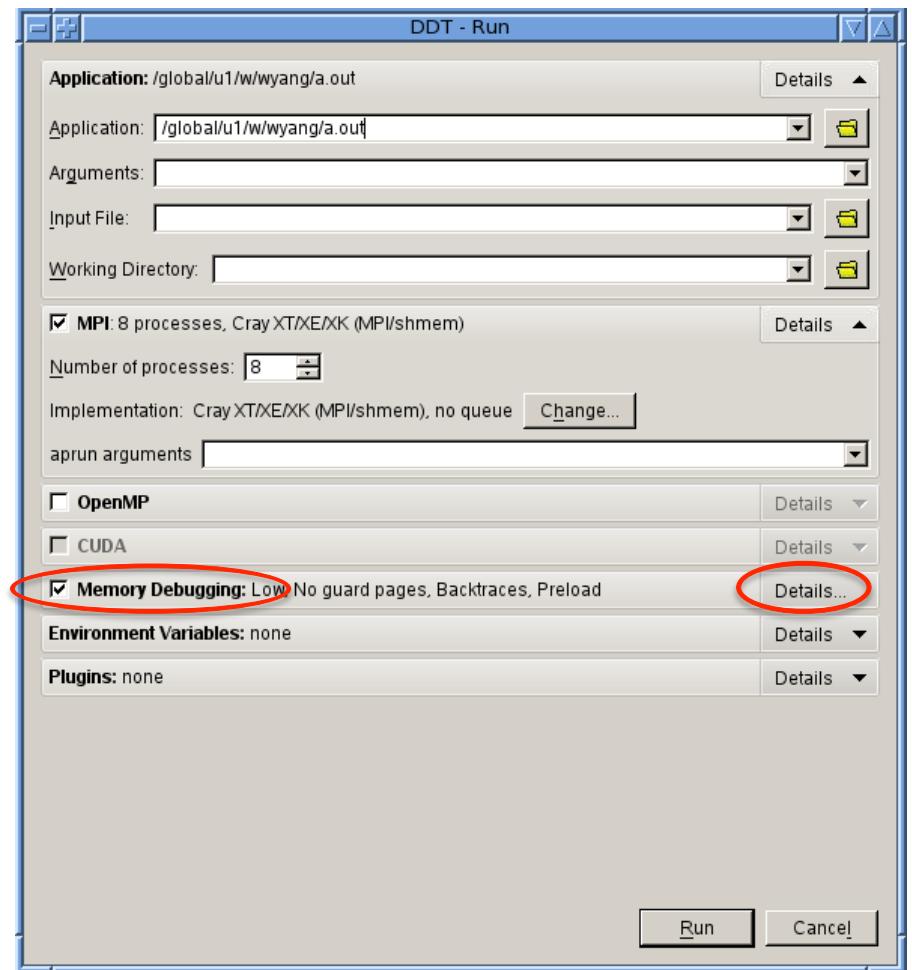
- Set CRAY_ROOTFS before running ddt

```
setenv CRAY_ROOTFS DSL      # for csh/tcsh shell  
export CRAY_ROOTFS=DSL       # for sh/ksh/bash shell
```



Starting Memory Debugging

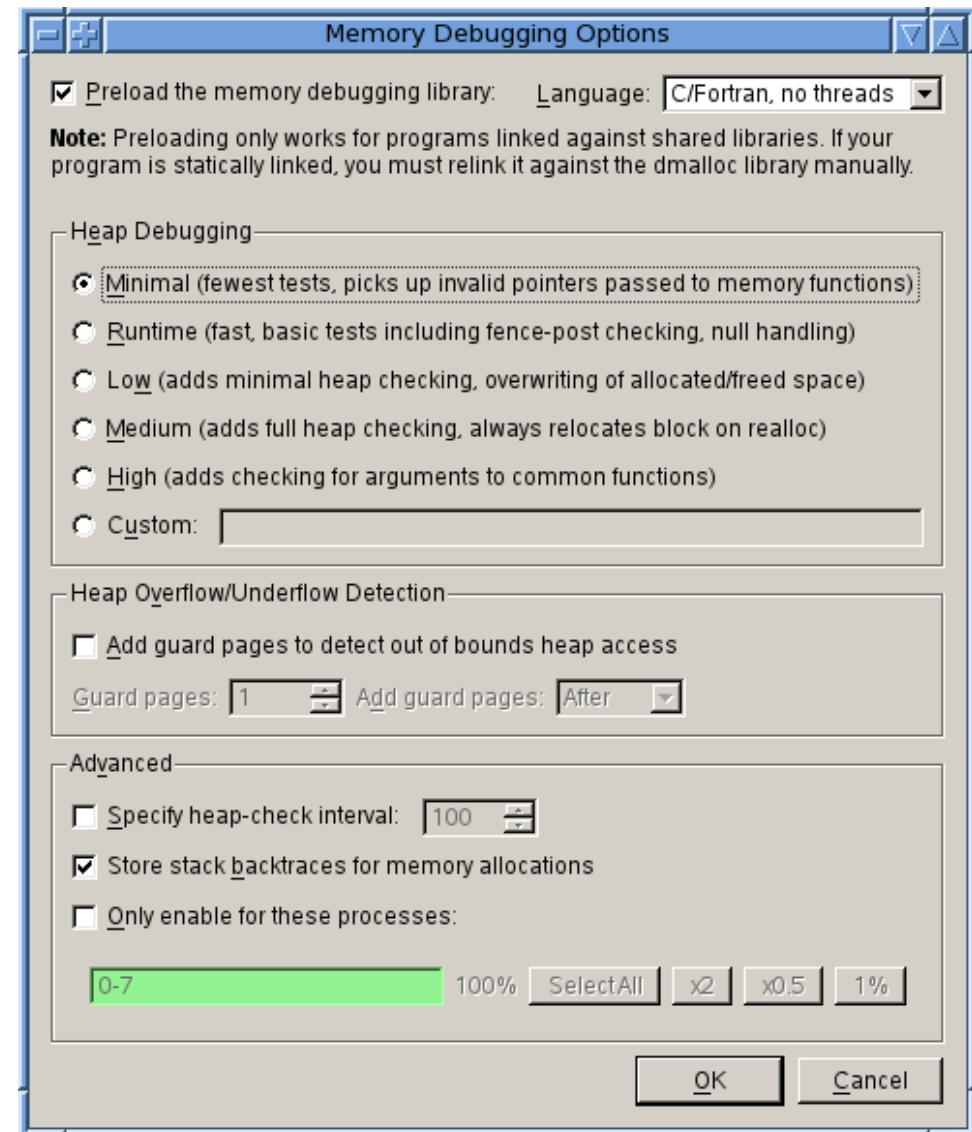
- Enable ‘Memory Debugging’ in the Run window
- Clicking on ‘Details’ opens the ‘Memory Debugging Options’ window for setting options (next slide)





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



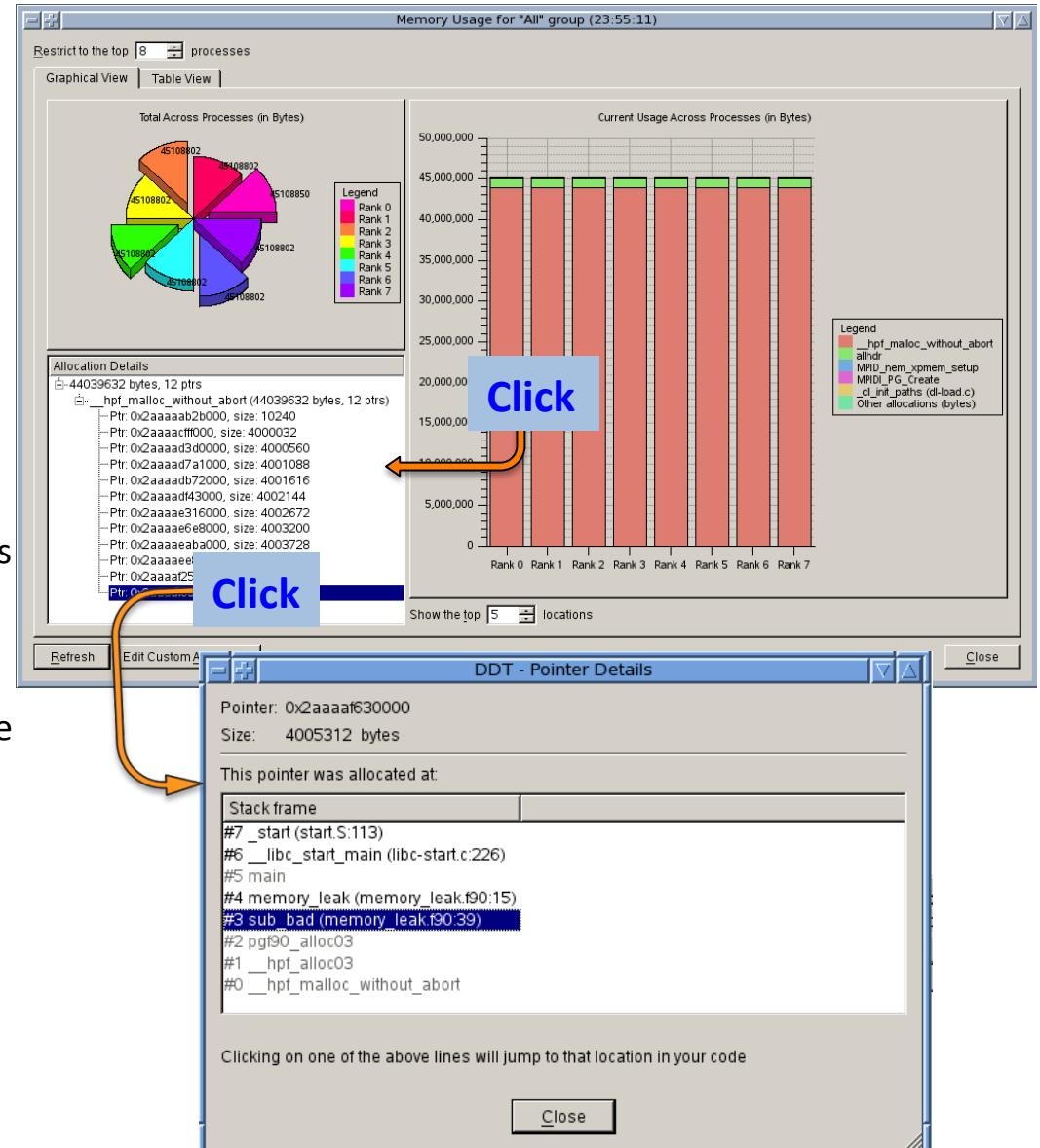
U.S. DEPARTMENT OF
ENERGY

Office of
Science



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



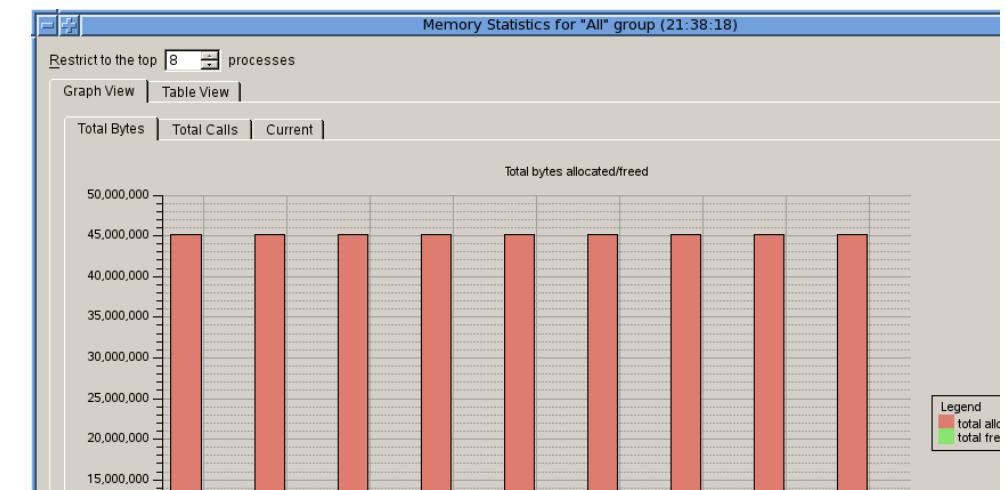
U.S. DEPARTMENT OF
ENERGY

Office of
Science



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 = 10000000
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
23
```

```
24      subroutine sub_ok(val,n)          no memory leak
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      subroutine sub_bad(val,n)         memory leak of
34      integer n
35      real val
36      real, pointer :: a(:)
37      allocate (a(n))
38      call random_number(a)
39      val = val + sum(a)
40      deallocate(a)                  ! not ok not to deallocate
41    end

42      subroutine sub_badx2(val,n)        memory leak of
43      integer n
44      real val
45      real, pointer :: a(:)
46      allocate (a(n))
47      call random_number(a)
48      val = val + sum(a)
49      allocate (a(n))                ! not ok to allocate again
50      call random_number(a)
51      val = val + sum(a)
52      deallocate(a)                  ! not ok not to deallocate
53    end
54
55
```

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



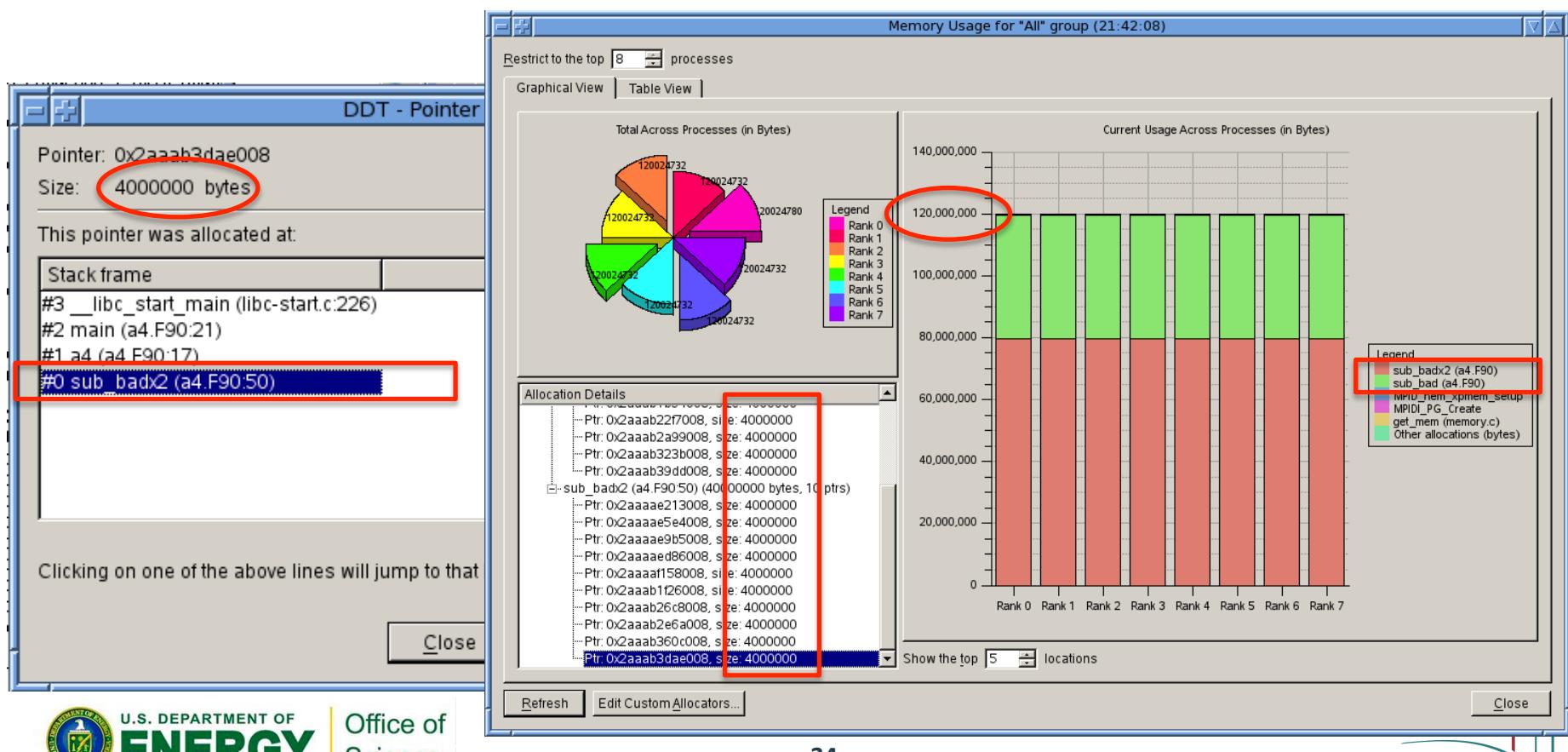
U.S. DEPARTMENT OF
ENERGY

Office of
Science



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



U.S. DEPARTMENT OF
ENERGY

Office of
Science



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, ...)

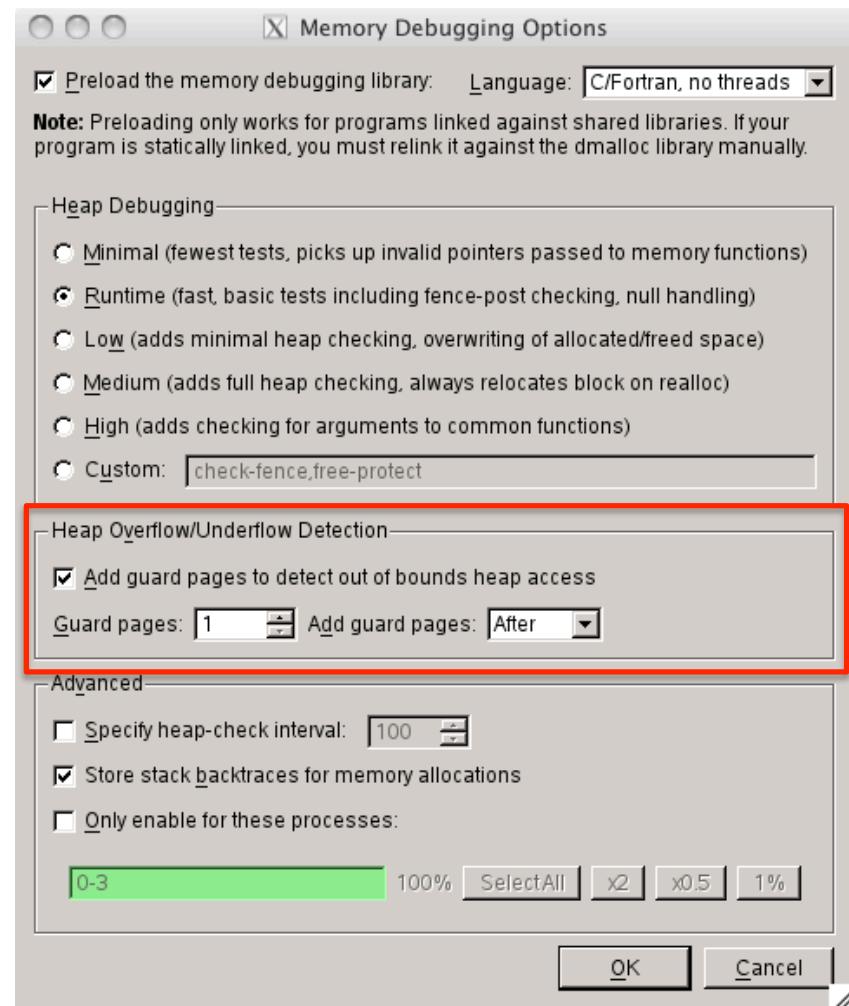
Location	Total Allocated bytes	Total Freed bytes	Current Memory Usage (TAB-TFB)	Total Allocation Calls	Total Free Calls
Before sub_ok loop	36,348	13,702	22,646	166	72
After 10 calls to sub_ok	40,036,348	40,013,702	22,646	176	82
After 10 calls to sub_bad	80,036,348	40,013,702	40,022,646	186	82
After 10 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=4KB) before or after (but not both) allocated blocks for detection
 - For reads and writes
- 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 extra small portion around an allocated block and DDT checks this area for corruption
 - Thus, checking only for writes
 - 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

The screenshot shows the Allinea DDT interface. On the left, the Project Files window displays a file named `f heap_overflow_underflow.f`. The code contains several lines of Fortran, including a warning message and a series of operations involving arrays `a` and `b` using the `cos` function. A specific line at the bottom is highlighted in blue: `b(n) = cos(a(n+ouf)) ! read overflow`. To the right, a larger window titled "Processes 0-3:" provides a detailed error report. It states: "Memory error detected in `heap_overflow_underflow` (`heap_overflow_underflow.f:14`). Process attempted to read or write beyond the end of heap memory it had allocated. This error will be suppressed for subsequent read/writes to this memory location. Tip: Use the stack list and the local variables to explore your program's current state and identify the source of the error." There is also a checkbox labeled "Suppress memory errors from this location in future". At the bottom of the interface, there are buttons for "Continue" and "Pause".



U.S. DEPARTMENT OF
ENERGY

Office of
Science



More Memory Debugging Example

- Deallocating the same memory block twice
- Not sure if it's DDT who stops the program in this example

The screenshot shows the Allinea DDT interface. On the left, the Project Files panel displays a file named `free_twice.f`. The code in the editor window is:

```
1 program free_twice
2 !... A buggy code prepared for a debugger tutorial by NERSC
3 include 'mpif.h'
4 integer, parameter :: n = 1024
5 real, allocatable :: a(:,), b(:)
6 integer i, ierr
7 call mpi_init(ierr)
8 allocate (a(n), b(n))
9 call random_number(a)
10 b = cos(a)
11 deallocate (a)
12 print *, sum(b)
13 deallocate (a,b) ! Oops..., deallocating 'a' again
14 call mpi_finalize(ierr)
15 end
```

A modal dialog box in the center-right says "DDT - all processes finished." It asks, "Every process in your program has terminated - would you like to restart this session from the beginning?" with "Yes" and "No" buttons. In the bottom-left corner of the DDT window, there is a red text overlay that says "Error message not from DDT?". The bottom-left also shows the output from `aprun`:

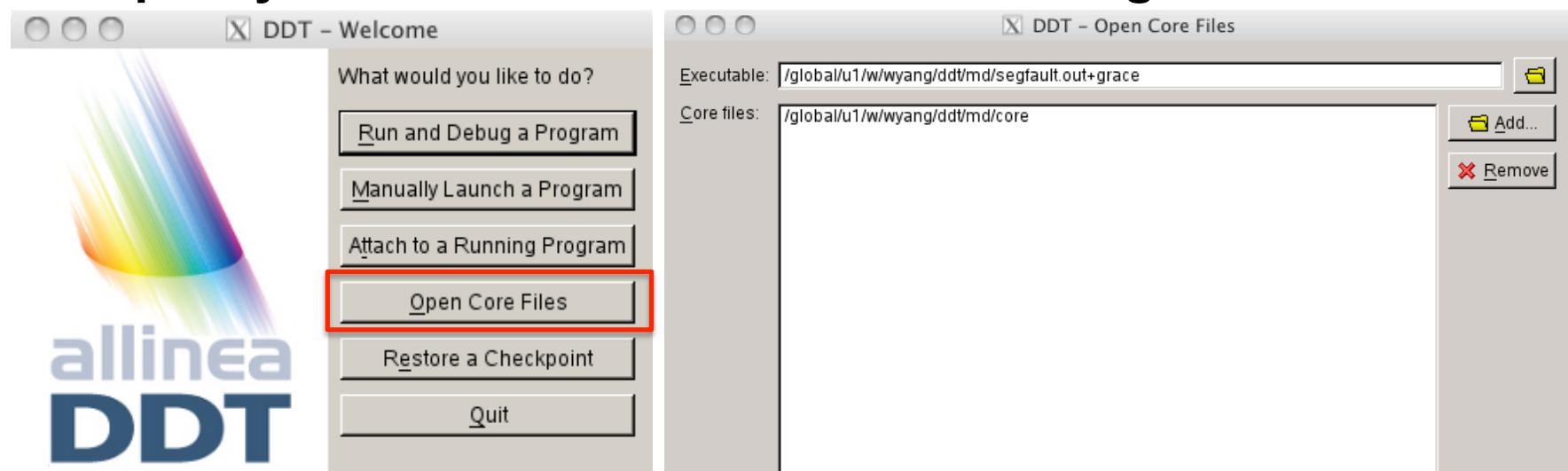
```
aprun: 862.7344
Other: 0: DEALLOCATE: memory at (nil) not allocated
Other: 0: DEALLOCATE: memory at (nil) not allocated
Other: 0: DEALLOCATE: memory at (nil) not allocated
```

Note: DDT can only send input to the aprun process with this MPI implementation



Opening Core Files

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



allinea
DDT

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

Allinea DDT v3.1-20638

Session Search View Help

Focus on current: Process Thread Step Threads Together

Threads: 1

Project Files | segfault.f

Search (Ctrl+K)

Project Files
Source Tree
Header Files
Source Files
segfault.f

1 program segfault
2 !... A buggy code prepared for a debugger tutorial by NERSC
3 integer, parameter :: n = 10
4 include 'mpif.h'
5 real, pointer :: a(:) => null()
6 real, pointer :: b(:) => null()
7 real, pointer :: c(:) => null()
8 integer me, i, ierr
9 call mpi_init(ierr)
10 call mpi_comm_rank(MPI_COMM_WORLD,me,ierr)
11 ! allocate(a(n), b(n), c(n)) ! Oops, forgot to allocate...
12 call sub(a,b,c,n)
13 print *, sum(c)
14 deallocate (a, b, c)
15 call mpi_finalize(ierr)
16 end
17
18 subroutine sub(a,b,c,n)
19 integer n
20 real a(n), b(n), c(n)
21 call random_number(a)
22 call random_number(b)
23 do i=1,n
24 c(i) = cos(a(i)) * sin(b(i))
25 end do
26 end

Locals Current Line(s) Current Stack

Variable Name Value

Type: none selected

Input/Output Stacks

Stacks

Function

segfault (segfault.f:12)
sub (segfault.f:21)
+pghpft_num

Stack backtrace when the code crashed

Evaluate

Expression	Value
a	<not allocated>
n	10

That was the problem!

Can evaluate expressions

NERSC Science

40

BERKELEY LAB



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 occurs
- 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
```

- Ignore the following warnings on Hopper

```
Warning: Configuration key "solib search path" in block "startup" is not recognised.
Warning: Configuration key "sys root" in block "startup" is not recognised.
```



U.S. DEPARTMENT OF
ENERGY

Office of
Science





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

Messages Tracepoints Output

Messages

[+] Expand All [-] Collapse All

#	Type	Time	Processes	Message
1	i	00:00.001	n/a	Launching program /global/u1/w/wyang/ddt/od/offline.out+carver.
2	i	00:05.946	0-3	Startup complete. ▶ Stacks
3	i	00:08.466	0-3	Process stopped at <u>breakpoint</u> in sub (offline.f:22). ▶ Stacks ▶ Stack for process 0 ▶ Local variables for process 0 (ranges shown for 0-3) i = 1 & j = 6
4	i	00:10.991	0-3	Process stopped at <u>breakpoint</u> in sub (offline.f:22). ▶ Stacks ▶ Stack for process 0 ▶ Local variables for process 0 (ranges shown for 0-3) i = 1 & j = 9
5	i	00:11.046	n/a	Every process in your program has terminated.

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
```

Click here for details



More Help?

- **User guide on each machine**
 - `$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**



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.**