
allinea

SCALE TO NEW HEIGHTS

Debugging with DDT at NERSC

*David Lecomber
CTO Allinea Software*

Contents

- **Overview of Allinea and DDT**
- **Using Allinea DDT to fix bugs at NERSC**
- **Hands on**

- **HPC tools company since 2001**
- **Core products**
 - **DDT** - Debugger for MPI, threaded/OpenMP and scalar applications
 - **OPT** - Optimizing and profiling tool for MPI and non-MPI applications
 - **DDTLite** - Plugin for Microsoft Visual Studio 2008
 - Adds parallel and multi-threaded components to user interface
 - Real parallel debugging for Windows!
 - Released September 22nd 2008

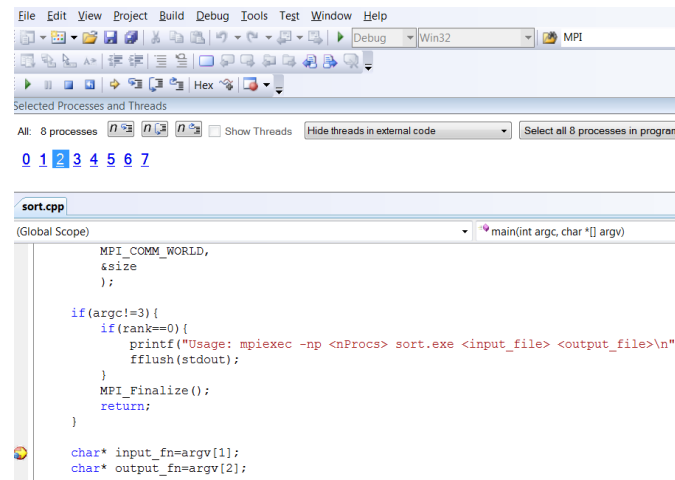
Partners



- Government
 - CEA, NERSC, IDRIS, BSC, ONERA, AWE, RAL, HLRS, CASPUR, CINECA, ORNL, NERSC, LLNL, Ifremer, Proudman, Plymouth Marine, BGS
- Universities
 - Sharcnet, Jülich, North West Grid, Vanderbilt, Penn State, TACC, IPGP, ETHZ, HLRS, LRZ, Dresden, Karlsruhe, ICHEC, Bristol, Nottingham, Glasgow, Edinburgh, Oxford, Tokyo, USATU, Arizona, UCL
- Aerospace research
 - DLR, EADS CCR, CIRA, MBDA, BAE Systems
- Commercial research
 - Synopsys, Airbus, Fujitsu (Japan & UK), CGGVeritas, Total, IFP, OHM, AVL, MTEM, Konica-Minolta

- **DDTLite for Visual Studio**

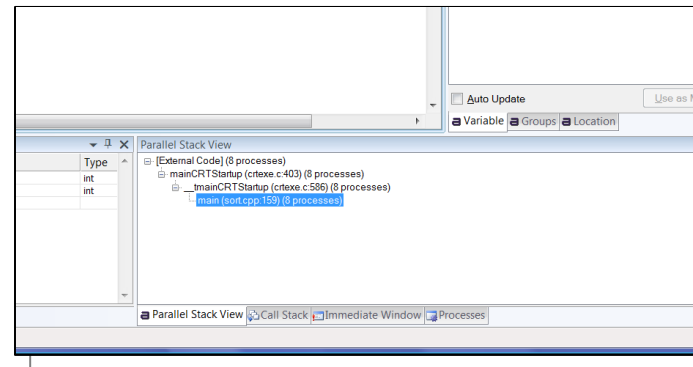
- Simplifying development on the Microsoft® platform
- Bringing features from DDT into Visual Studio®
- Take total control of all processes
- Makes Visual Studio® a true parallel debugger
- Released last week!



```
File Edit View Project Build Debug Tools Test Window Help
Debug Win32 MPI
Selected Processes and Threads
All: 8 processes
0 1 2 3 4 5 6 7
sort.cpp
(Global Scope) main(int argc, char *[] argv)
MPI_COMM_WORLD,
&size
);
if(argc!=3){
if(rank==0){
printf("Usage: mpiexec -np <nProcs> sort.exe <input_file> <output_file>\n");
fflush(stdout);
}
MPI_Finalize();
return;
}
char* input_fn=argv[1];
char* output_fn=argv[2];
```

Debugging with DDTLite

- **Group control**
 - Step or play multiple processes
- **Compare data over processes**
- **See all the process stacks**
- **C/C++/C#, Intel or PGI Fortran**



Using DDT

Using DDT to fix bugs - fast!

- **Overview of the features**
- **How to get started with DDT**
- **Memory Debugging with DDT**

Parallel Software is Complicated

- **Multithreaded, multiprocess code**
 - The usual issues: bugs, speed
 - ... now add communication, synchronization, race conditions, deadlock, scalability
- **Hardware / Software capability misaligned**
 - Significant gap between user requirements and application capabilities
 - Emphasis is now moved to software development
 - How do we address this issue?

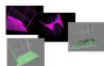
Some problems DDT helps with

- **For when you have a bug – if your application**
 - Crashes all the time
 - Click play and run until it crashes – DDT shows the point of segfault, abort or exit – before it has quit
 - Crashed yesterday
 - Load up the core file and look at where it crashed
 - Crashes sometimes
 - Possibly a memory error – turn on memory debugging and force the problem
 - And many other scenarios – not just crashes
- **It helps when maintaining and developing code**
 - When you need to know how it works to add a feature
 - Load DDT and just watch how your program runs

- **A mature, powerful and highly intuitive tool**
 - Traditional focus has been HPC
 - Developing new capabilities.....multicore, GP-GPU
- **Cross-platform support**
 - Linux, Solaris, AIX, Super-UX
 - GNU, Absoft, IBM, Intel, PGI, PathScale, Sun compilers
 - x86, x86-64, ia64, PowerPC, Cell, UltraSparc, NEC
 - Support for all major scheduling systems
 - Across all MPIs
 - OpenMP and Threads

allinea ddt

the distributed debugging tool



allinea
SCALE TO NEW HEIGHTS

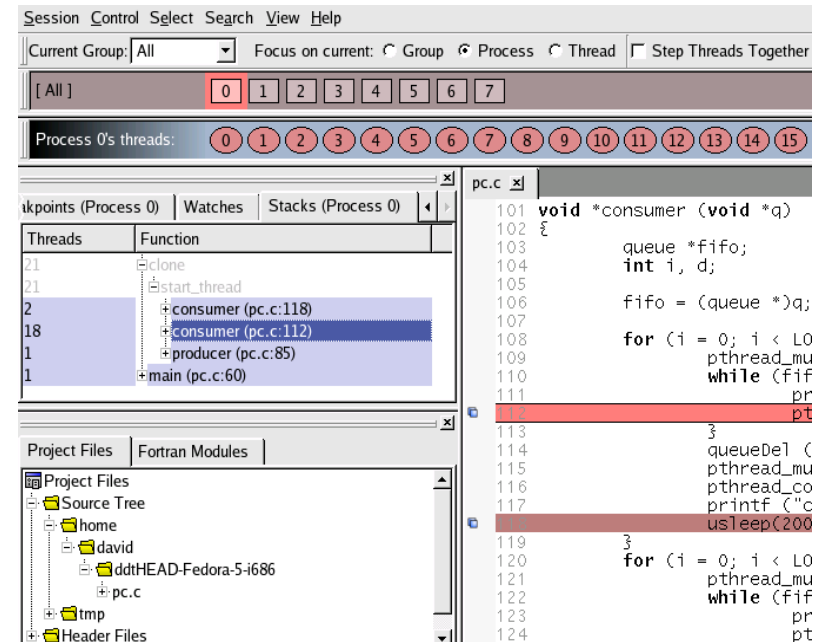
sales@allinea.com
support@allinea.com

The basic features

- **Automatically locates source files**
- **Step, play, pause**
- **Breakpoints and data watchpoints**
- **Fortran 90, 95 and 2003**
 - Modules, allocatable data, pointers and derived types
- **C, C++ support**
 - Structs, classes, pointers, STL, namespaces, virtual functions and templates
 - Catch exceptions at throw or at catch point

Features for multithreading

- Perform actions individually or collectively
- Breakpoints for all or per thread
- See all threads at a glance
 - Parallel stack view shows threads and groups common stacks



Features for parallel codes

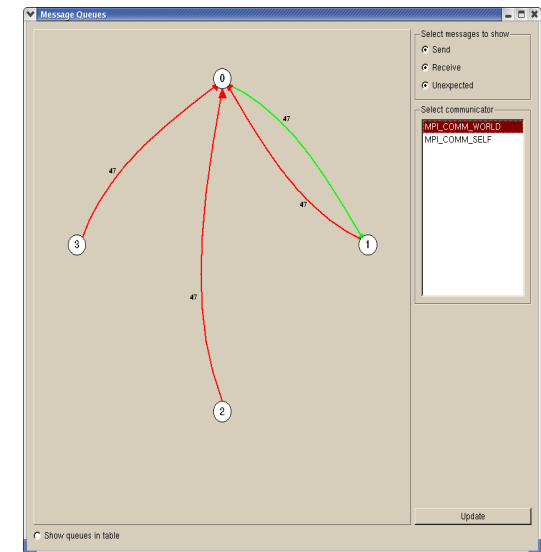
- Support for every MPI
- Control processes individually or by groups
- Visualize message queues

The screenshot shows the Allinea Distributed Debugging Tool interface. At the top, the title bar reads "Allinea Distributed Debugging Tool - [/home/matt/dd/examples/hello.c]". Below the title bar is a menu bar with "Session", "Select", "Search", "View", "Code", and "Help". A "Current Group" dropdown is set to "Crash". Below this is a control panel with buttons for play, pause, step, and other debugging actions. The main area displays a process control panel with four rows: "All" (0-15), "Workers" (1-15), "Crash" (3, 5, 8), and "Root" (0). Below this is a project tree showing "Project Files", "Header Files", "Source Files", and "hello.c". The code editor shows the following code:

```

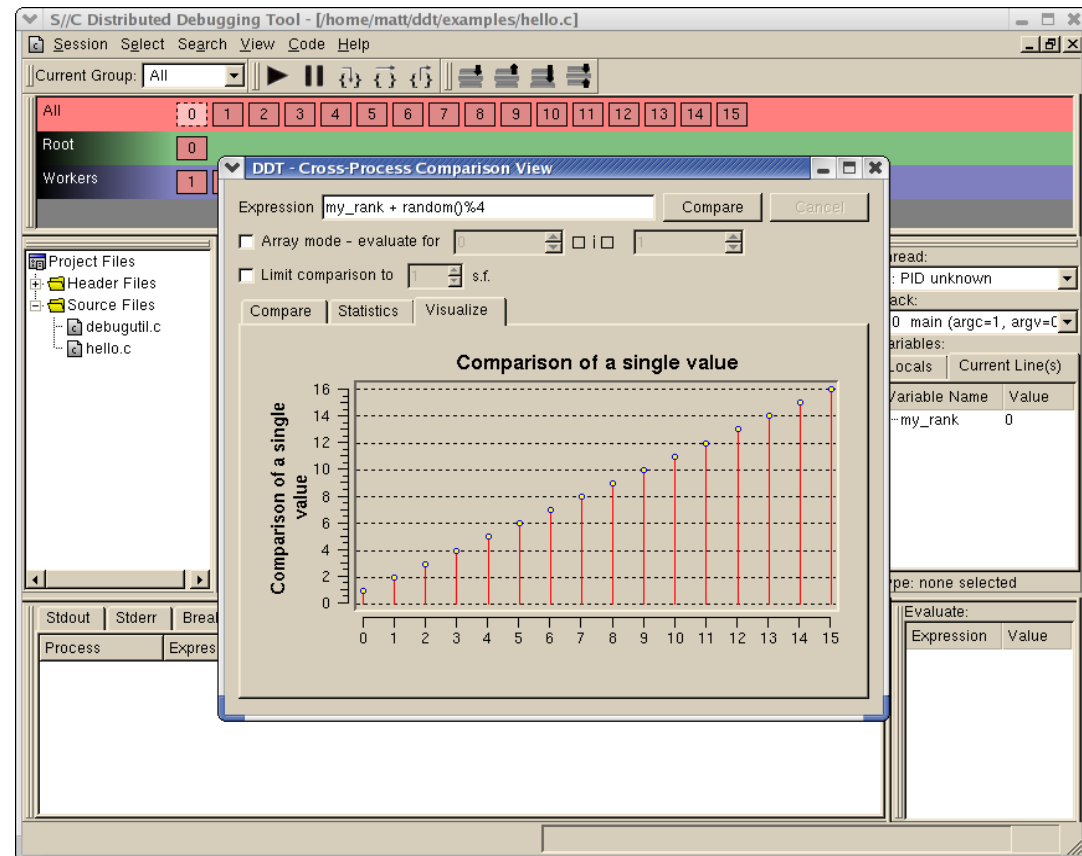
96 printf("I am running on %s as %d.\n", message, getpid());
97 printf("I have %d arguments.\n", argc);
98 printf("\tHow many did I say?\n");
99 printf("They are:\n");
100 for (i=0; i<argc; i++)
101     printf("%d\n", i, argv[i]);
102
103 Workers (3.0, 5.0, 8.0)
104 All (3.0, 5.0, 8.0)
105     {
106     e an environment too!\n");
107     e:\n");
108     Breakpoint for Crash _environ++;
109     printf("%s\n", *environ);

```



Viewing concurrent data

- Cross process/thread comparison



Viewing vast data

- **Visualize multidimensional data**
 - From 2D viewer to new multidimensional viewer
 - 3D OpenGL array viewer (stereo !)

DDT - Multi-Dimensional Array Viewer

Array Expression: `[array[$i][$j][$k][$l]]`

Range of \$i: From: 0 To: 24 Display: Rows

Range of \$j: From: 0 To: 24 Display: Rows

Range of \$k: From: 0 To: 24 Display: Rows

Range of \$l: From: 0 To: 24 Display: Columns

Aggregate Function: Sum

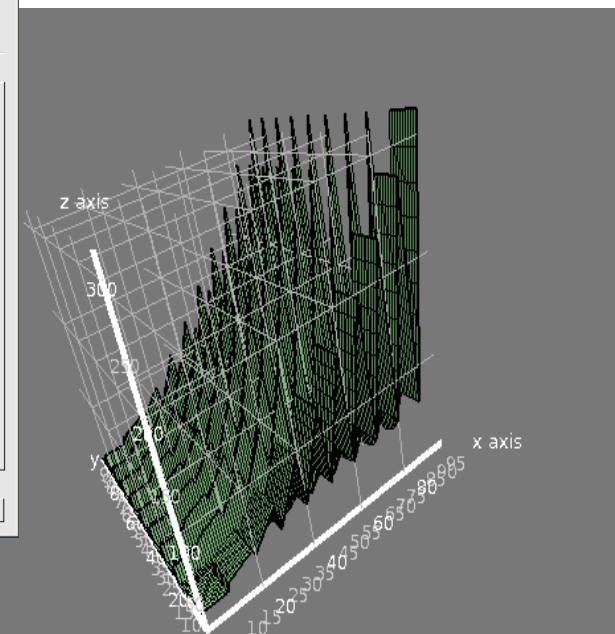
Filter: =

Data Table | Statistics

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
0	0	0	20	40	60	80	100	120	140	160	180	200	220	240	260	280	300	320	340	360	380	400
1	20	40	60	80	100	120	140	160	180	200	220	240	260	280	300	320	340	360	380	400	420	440
2	40	60	80	100	120	140	160	180	200	220	240	260	280	300	320	340	360	380	400	420	440	460
3	60	80	100	120	140	160	180	200	220	240	260	280	300	320	340	360	380	400	420	440	460	480
4	80	100	120	140	160	180	200	220	240	260	280	300	320	340	360	380	400	420	440	460	480	500
5	100	120	140	160	180	200	220	240	260	280	300	320	340	360	380	400	420	440	460	480	500	520
6	120	140	160	180	200	220	240	260	280	300	320	340	360	380	400	420	440	460	480	500	520	540
7	140	160	180	200	220	240	260	280	300	320	340	360	380	400	420	440	460	480	500	520	540	560
8	160	180	200	220	240	260	280	300	320	340	360	380	400	420	440	460	480	500	520	540	560	580
9	180	200	220	240	260	280	300	320	340	360	380	400	420	440	460	480	500	520	540	560	580	600
10	200	220	240	260	280	300	320	340	360	380	400	420	440	460	480	500	520	540	560	580	600	620
11	220	240	260	280	300	320	340	360	380	400	420	440	460	480	500	520	540	560	580	600	620	640
12	240	260	280	300	320	340	360	380	400	420	440	460	480	500	520	540	560	580	600	620	640	660
13	260	280	300	320	340	360	380	400	420	440	460	480	500	520	540	560	580	600	620	640	660	680
14	280	300	320	340	360	380	400	420	440	460	480	500	520	540	560	580	600	620	640	660	680	700
15	300	320	340	360	380	400	420	440	460	480	500	520	540	560	580	600	620	640	660	680	700	720

Expression "array[\$i][\$j][\$k][\$l]" evaluated for process 0 at 11:49.

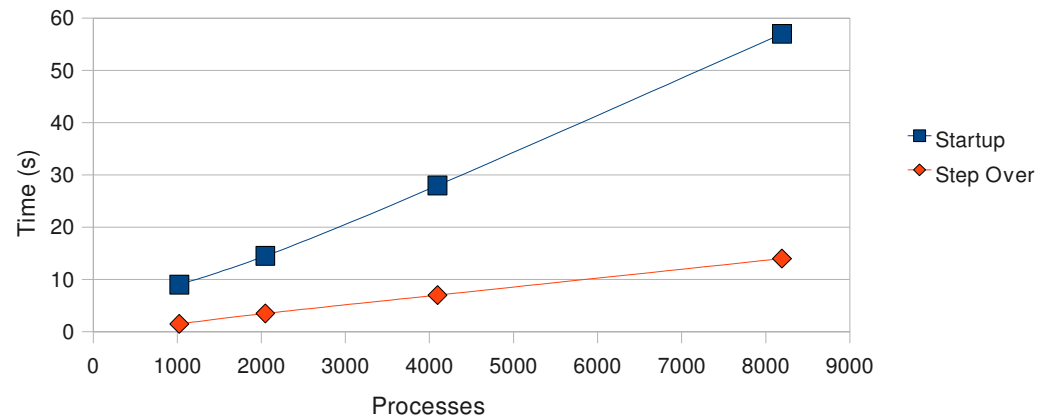
Visualize in 3D | Export to Spreadsheet... | Close



- **Increased thread and processor counts are coming**
 - Desktop multicore and GPUs - multithreading by default
 - 1,000+ core clusters everywhere
- **DDT allows control of 1,000s of processes**
 - An MPI application or a custom network of processes
 - Highly threaded codes GPUs, and (very-) multi-core ready
 - Or many single core Monte Carlo applications

DDT 2.4 Measured Timings

Dual Core 2.0 GHz 1GB RAM frontend



Finding rogues quickly

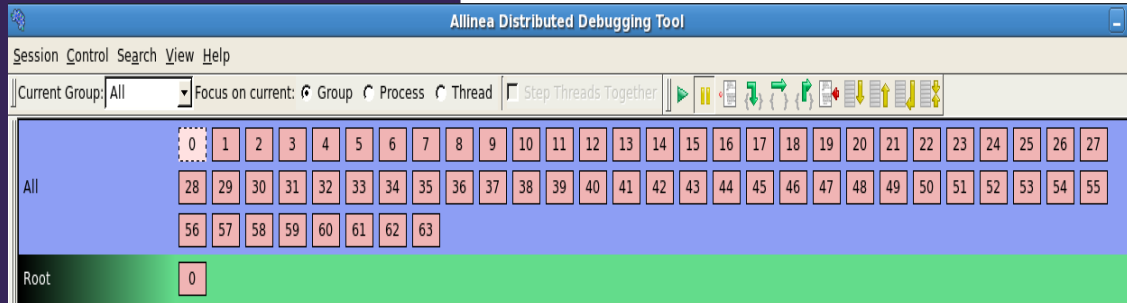
- **Parallel Variable View**
 - Find processes or threads with different data
- **Parallel Stack View**
 - Identifies classes of process and thread behaviour easily
 - Allows rapid grouping of processes
- **Integrated with process groups**

The screenshot displays a development environment with two main panels. The top panel shows a C++ source code file with a red highlight under a `while` loop. The bottom panel shows a 'Parallel Stack View' window with a tree structure of process and function calls.

```

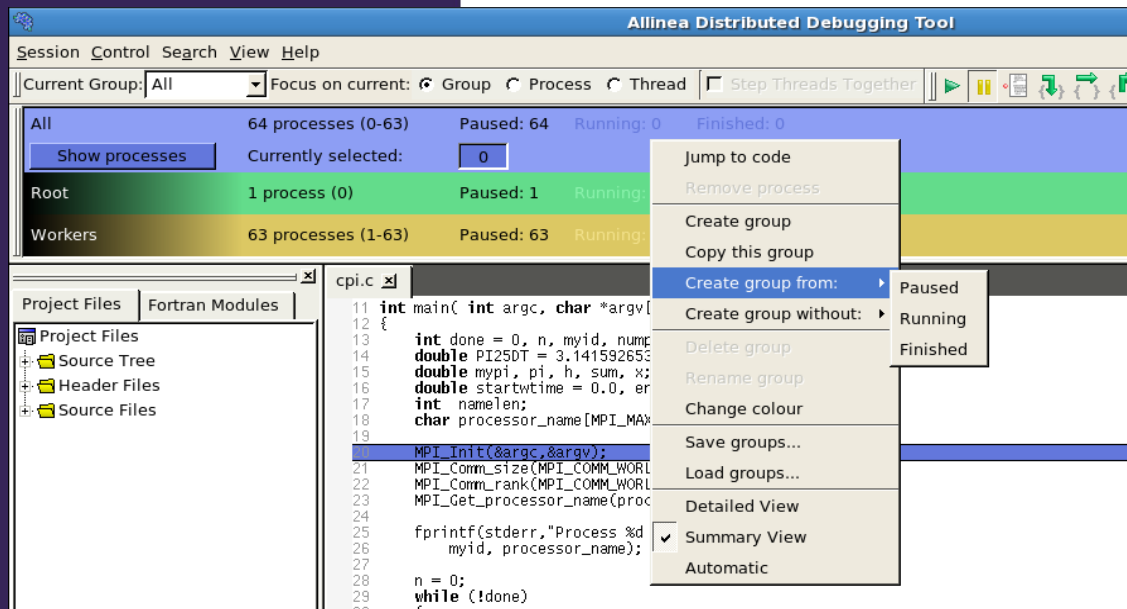
chunk.c | QMCJastrowParameters.cpp | heap.c | basic_string.h | malloc.c | QMCWalker
664 localTimers->getInitializationStopwatch()->start();
665
666 if( qmcCheckpoint && Input.Flags.use_available_checkpoints == 1 )
667     // There is a checkpoint file
668     readURL(qmcCheckpoint);
669
670 while(
671     // There is not a checkpoint file
672     !qmcCheckpoint.isInitialized(qmcCheckpoint));
673
674 localTimers->getInitializationStopwatch()->stop();
675 qmcCheckpoint.close();
676
677
678 void QMCManager::checkTerminationCriteria()
679 {
680     checkMaxStepsTerminationCriteria();
681     checkConvergenceBasedTerminationCriteria();
682 }
    
```

Procs	Function
16	main (QMCBeaver.cpp:35)
16	QMCBeaver (QMCBeaver.cpp:64)
8	QMCManager::initialize (QMCManager.cpp:58)
8	QMCManager::initialize (QMCManager.cpp:623)
1	QMCRun::randomlyInitializeWalkers (QMCRun.cpp:86)
1	QMCWalker::initializeWalkerPosition (QMCWalker.cpp:1071)
2	QMCRun::randomlyInitializeWalkers (QMCRun.cpp:85)
2	QMCWalker::initialize (QMCWalker.cpp:360)
2	QMCFunctions::initialize (QMCFunctions.cpp:50)
1	QMC Slater::allocate (Array2D.h:119)
1	operator
1	malloc (malloc.c:1092)
1	dmalloc_malloc (malloc.c:625)
1	QMC Slater::allocate (Array2D.h:116)
1	operator
5	QMCRun::randomlyInitializeWalkers (QMCRun.cpp:80)
5	QMCFunctions (Array3D.h:152)
8	QMCManager::initialize (QMCManager.cpp:31)



- **At lower parallelism**

- See state of each process
- Drag and drop to manage groups



- **At 100s or 1000s of processes**

- Summary view gives rapid overview and same level of control

How to get started at NERSC

- **Compile your code – with “-g” option**
- **Bring up the DDT GUI - “ddt”**
- **Select your program**
- **Click submit! DDT submits the job for you.**

- **Is it that easy?**

- **Not quite:**
 - Temporarily complicated until MPT update to 3.1
 - `module unload xt-mpt/3.0.2`
 - `module load xt-mpt/2.0.53b`
 - **Recompile your code**

- **Interactive sessions?**
 - Change “Session/Options/Job Submission” so that DDT does not submit for you.
 - “qsub -l” and wait, then launch DDT in the shell
 - Why would I do that?
 - Quickly and repeatedly try something: run 10 times in a 10 minute session instead of waiting in the queue 10 times
 - Why wouldn't I do that?
 - Easier not to – is a trade off – it's convenient if your program runs for a while
- **Using the queue differently with DDT**
 - Can change the default queue at click of button
 - Click “change” on launch page default is “debug”
 - Can use DDT up to 1024 processes
 - “regular” queue, not “debug” queue
 - But remember your budgeted hours....!!
 - Change the walltime in the same manner

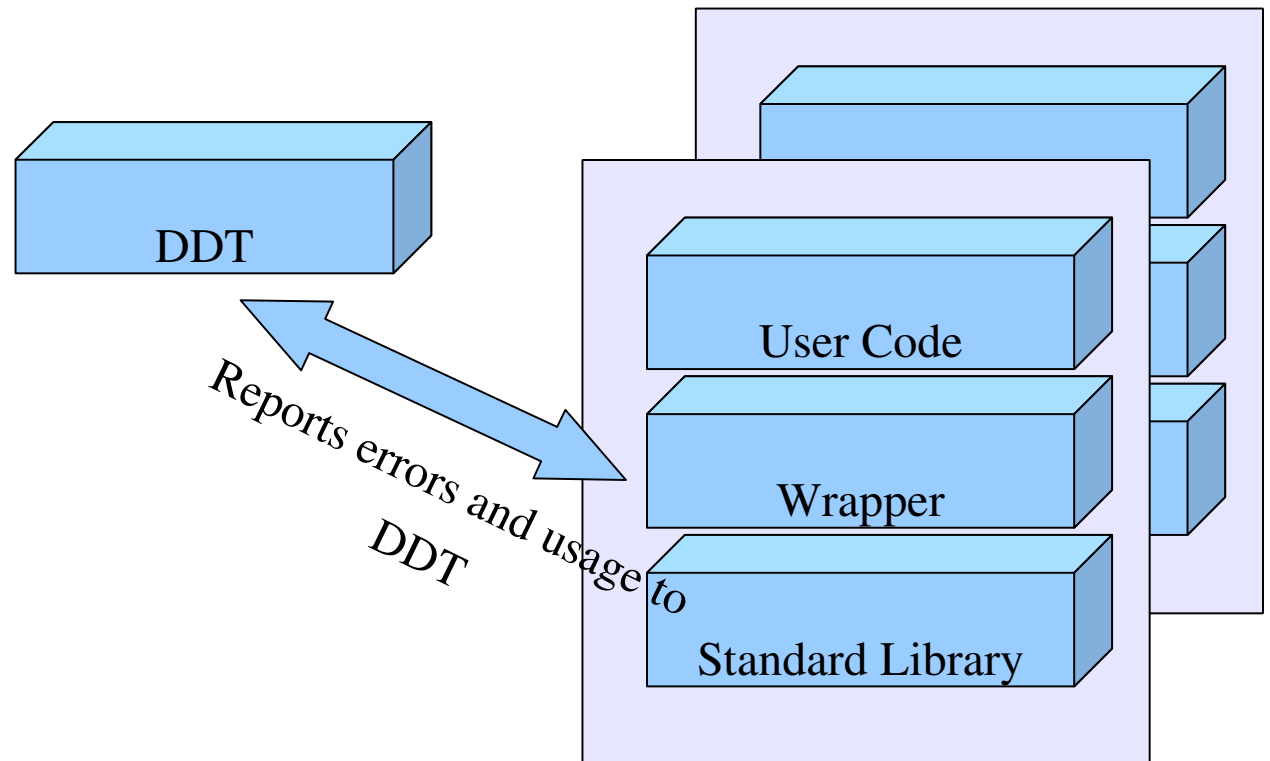
Interlude!

DDT demo

A Tour of Memory Debugging

Focus on Memory Debugging

- **DDT's memory debugging works with the Heap**
 - Memory allocated via allocate (F90), malloc (C), or new (C++)



Memory Debugging

Check your overall memory usage

trisol.f90

```

60 ! Check the solution
61 !
62 !
63 CALL CHECK (a,b,x,resnm)
    
```

Overall Memory Stats

Graph View | Table View

Total bytes allocated/freed

Processor	Total allocated bytes	Total freed bytes
P.0	830,000	20,000
P.1	480,000	20,000
P.2	480,000	20,000
P.3	480,000	20,000
P.4	480,000	20,000
P.5	480,000	20,000
P.6	480,000	20,000
P.7	480,000	20,000

Total number of memory allocation/deallocation calls

Processor	Allocation calls	Deallocation calls
P.0	200	95
P.1	120	25
P.2	120	25
P.3	120	25
P.4	120	25
P.5	120	25
P.6	120	25
P.7	120	25

Refresh Close

- **Common C coding error, increasingly in F90**
 - When memory is not released after use
 - Kills job if repeated too often, eg. calls to leaky function
 - First identify if there is a problem
 - Then find it..

```
void fun()  
{  
    Car *p;  
    p = new Car();  
    /*  
    ... do something  
    */  
    return;  
}
```

Checking for leaks

Locate where memory was allocated in depth

Check your current memory usage and where memory was allocated

The screenshot shows the Allinea Distributed Debugging Tool (DDT) interface. The main window is titled 'Current Memory Usage' and has two tabs: 'Graphical View' and 'Table View'. The 'Graphical View' contains two charts: a pie chart titled 'Total Across Processes (in Bytes)' and a bar chart titled 'Current Usage Across Processes (in Bytes)'. The pie chart shows memory usage broken down into segments of various colors, with values like 456989, 456989, 456989, 456989, 456989, 456989, 456989, and 456989. The bar chart shows memory usage across processes 0 through 7, with Process 0 having the highest usage. A 'DDT - Pointer Details' dialog box is open, showing a pointer at 0x41215000 with a size of 199360 bytes. The dialog also shows the stack frame where the pointer was allocated, with the top frame being 'main' at address 0x804d2e8. The dialog includes a 'Close' button and a note: 'Clicking on one of the above lines will jump to that location in your code'. The 'Allocation Details' section in the main window shows a list of pointers and their sizes: 'Ptr: 0x41215000, size: 199360', 'Ptr: 0x41246000, size: 3560', and 'Ptr: 0x41247000, size: 3560'. The 'Current Usage Across Processes (in Bytes)' bar chart has a legend with the following categories: 'Largest set (bytes)', '2nd largest set (bytes)', '3rd largest set (bytes)', '4th largest set (bytes)', '5th largest set (bytes)', and 'Other allocations (bytes)'. The bar chart shows memory usage across processes 0 through 7, with Process 0 having the highest usage.

- **Dangling Pointers**

- Pointer variables pointing to memory after it has been discarded
- Pointing to discarded memory now reissued to another role

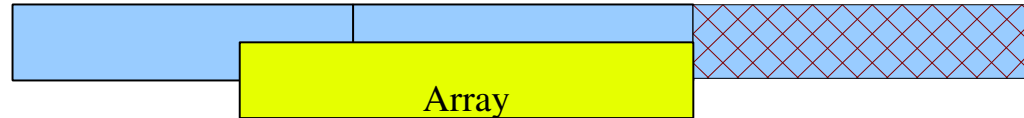
- **We can prevent both**

```
Car *p, *q;  
p = new Car();  
q = p;  
/*  
... do something  
*/  
delete p;  
  
/*  
... something else  
*/  
q->drive();
```

Beyond Bounds Reading

- **“Guard Pages”**

- Instant protection against read/write outside an array



- **DDT uses OS to lock the page**

- Above or below for protection (but not both)
- Or lock more pages.. for multi-dimensional arrays

- **.. and can still allow you to continue!**

- **“Fence post” checking**

- Periodic protection against write outside an array – at both ends

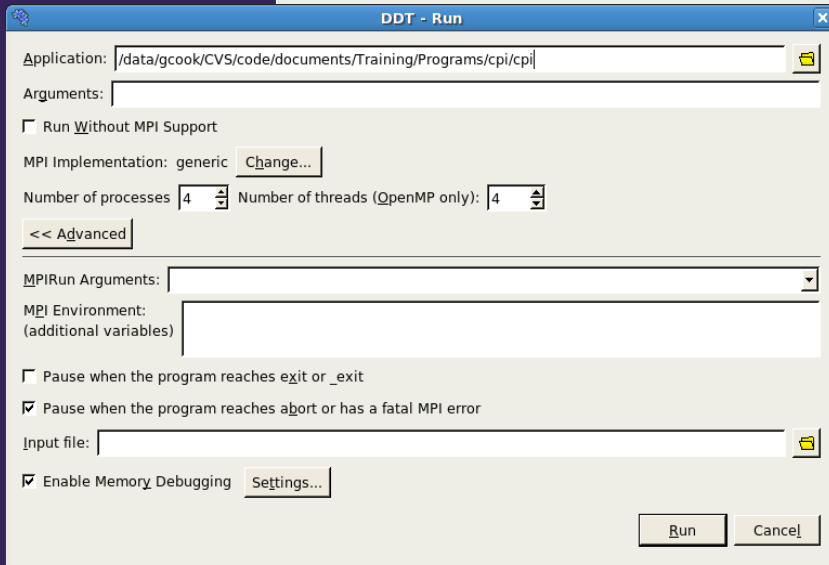
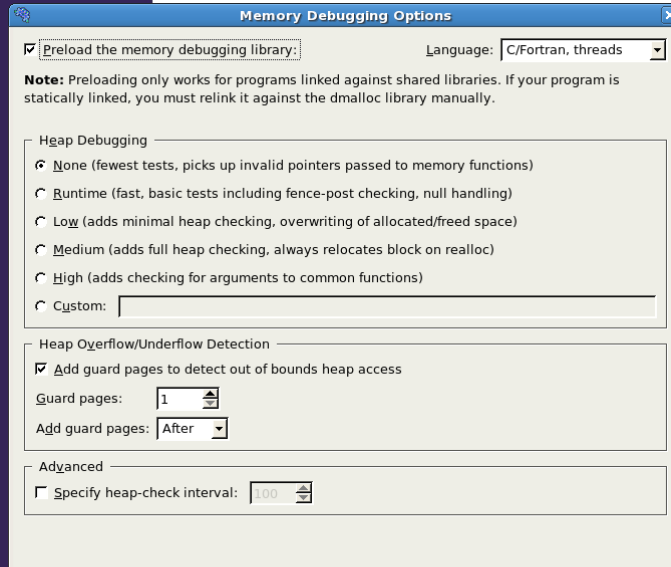
Memory Debugging

The screenshot shows the Allinea Distributed Debugging Tool interface. At the top, there is a menu bar with 'Session', 'Select', 'Search', 'View', and 'Help'. Below the menu bar is a toolbar with various icons for session control. The main area is divided into several panes:

- Process List:** Shows 'All' (red bar), 'Root' (green bar), and 'Workers' (blue bar). Each bar has a row of buttons numbered 0 through 7.
- Project Files:** A tree view on the left showing 'Header Files' and 'Source Files' with sub-items like 'check.f90', 'd1aruv.f', 'gendata.f90', 'matnm.f90', 'mod_trisol.f90', 'numoc.f', 'solve.f90', and 'trisol.f90'.
- Source Code:** A window showing the source code for 'trisol.f90'. The code includes comments like 'Check the solution' and 'Print the solution', and a 'CALL CHECK (a,b,x, resnm)' statement. A yellow warning icon is visible in the code editor.
- Stack/Variables:** A window on the right showing the current thread and stack frame. It lists variables like 'MATNRM' and 'anrm' with their values.
- Error Dialog:** A modal dialog box titled 'Distributed Debugging Tool' is displayed in the foreground. It contains the following text:
 - Processes 0-7:**
 - Program stopped in vprintf from /lib/libc.so.6 with signal SIGSEGV.**
 - Reason/Origin: address not mapped to object**
 - Your program will probably be terminated if you continue.**
 - You can use the stack controls to see what the process was doing at the time.**
 - Always show this window for signals
 - Buttons: **Continue** and **Pause**

Stop immediately on reads beyond array end and common errors

Using memory debugging



- **Easy to use**

- Tick check box to enable and use the default, or choose your own settings
- **Do not tick “Preload” - we use link explicitly on Cray**

- **Relink**

- Use “-Bstaticddt” compile option at NERSC

Q & A: Open Floor

- **Any questions?**