

# **Allinea DDT Training Day**

## **Reading Materials and Exercises**

*April 2014*



# Allinea DDT Training Course

## ***Objectives***

This training course contains the materials you will need to get started with Allinea DDT.

If you are in a taught Allinea DDT tutorial, the tutor will begin by showing you many of the Allinea DDT features as a quick overview – the exercises cover a lot of Allinea DDT, but it's nice to see a little tour up front of what to expect as the day progresses.

There are a series of walked through examples – each followed by a hands on exercise for you to try.

All of these materials will be left behind for you to use after the day is complete, so you can return to them any time. Please feel free to mail your tutor, or [support@allinea.com](mailto:support@allinea.com) afterwards if there is anything that you still want to know!

## ***Session 0 – Getting Started***

Allinea DDT will have been configured to work with your system already. Your tutor today will tell you what to do to get Allinea DDT started, if you do not already know.

The examples in this course all have a makefile, which should work on most systems.

If your command for MPI source compilation is “mpicc” or “mpif90” then to make an example, use “make” alone.

```
make
```

**NOTE:** if your MPI compilation command is different you will need to set the CC or F90 variables – the following will work on a Cray XT or XE system for example.

```
make CC=cc  
make F90=ftn
```

Some platforms may require additional flags to enable memory debugging during compilation – notably IBM AIX and Blue Gene plus the Cray XT and XK systems all use static linking. There is information in the userguide about how to link in memory debugging support on those systems – other systems do not require you to do anything.

## Session 1: Straightforward Crashes

First let's look at debugging crashes – the kinds of crashes or errors that happen repeatedly. These are often segmentation faults or aborts, or even exiting with an error code.

This form of bug is very common – and very easy to fix with a debugger, but much harder without one!

We'll all walk through one case using the `cstartmpi` example together. This is a messy, confusing C program, with some bugs.

Afterwards, there's another crash for you to solve on your own or with your neighbour.

### Walkthrough

First we will compile the application `cstartmpi`. There is a makefile for this in the `cstartmpi` directory.

```
cd cstartmpi
make
```

Run with 4 processes - it's ok

```
mpirun -np 4 ./cstartmpi.exe
```

Now try again with some arguments

```
mpirun -np 4 ./cstartmpi.exe some input arguments
```

The program will abort as there has been a problem:

```
rank 0 in job 52 tenku_60773 caused collective abort of all
ranks
```

The next step is to bring this up in Allinea DDT and find out what happened. The quickest way to start is to run Allinea DDT almost identically to the way you launched MPI.

```
ddt -start -np 4 ./cstartmpi.exe some input arguments
```

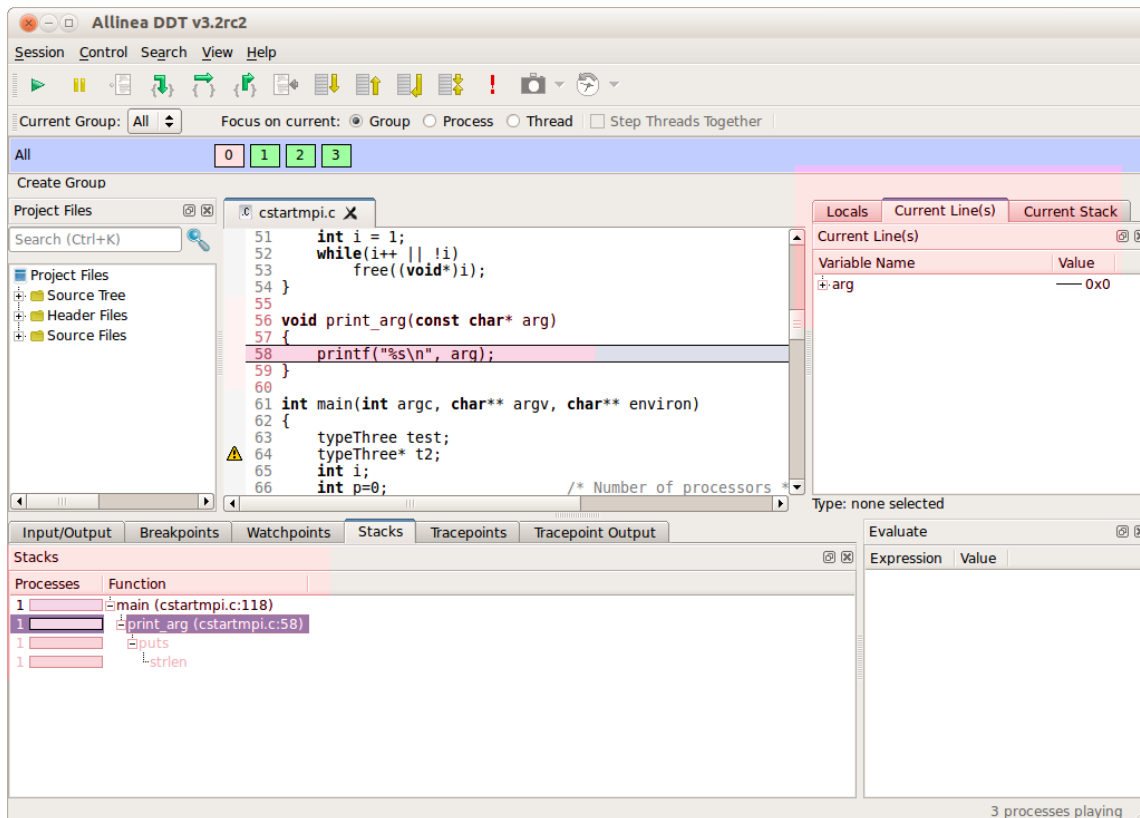
The Allinea DDT GUI will appear - and it will have started your program. You can see the source code, and there is a colour highlighted line. This is the current location that processes are at. Initially all processes are paused after `MPI_Init`.

At the top of Allinea DDT you will see a number of control buttons, a bit like a VCR (or PVR for the modern reader). If you hover the mouse over the control buttons, a tooltip will appear that gives the name of the button.

- Play – make the processes in the current group run until they are stopped.
- Pause – cause the processes in the current group to pause, allowing you to examine them.
- Add Breakpoint – adds a breakpoint at a line of code, or a function, that will cause processes to pause as soon as they reach that location.
- Step Into – will either step the current process group by a single line, or if the line involves a function call, it will step into the function instead.

- Step Over – will step the current process group by a single line.
- Step Out – will run the current process group to the end of their current function, and return to the calling location.

*Press play to run the program.  
Allinea DDT stops with an error message – indicating a segmentation fault.*



*Illustration 1: Allinea DDT display after dismissing the dialog*

The screenshot shows Allinea DDT after the dialog has been dismissed – we've colour highlighted the most important parts.

At the bottom of the GUI you can see the Stacks view (you may need to raise the tab by clicking on it to see it). This is tightly connected to the source code, also highlights in the screenshot, and shows where all the paused processes are: all the current function calls – higher points of the tree call the lower branches.

Often just looking at the variables at different points in the stack is enough to tell you why the program crashed.

In this case – you can see `arg` is the null pointer (`0x0`) which is invalid for its usage in the `printf` statement in the code. Hence, the program crashed because `print_arg` was called with the wrong thing.

*Click on the “main” directly above the print\_arg function in the Stack View.*  
*This takes you to main which lets you see where that arg value comes from.*  
*Now click on the “Locals” tab (on the right-hand side of the GUI) – you are seeing all the local variables.*  
*Click on the “Current Line” tab to simplify and show only the variables on that line.*  
*Click and drag between lines 113 and 118 in the source code to show all the variables in that region.*

You can now see y is clearly incorrect - there aren't that many arguments (argc).

To find why is it wrong, examine the line 117: x is being checked against argc but y is being incremented.

*Fix the for loop condition in your favourite editor to read “y < argc” then recompile and re-run - now it works!*

## Exercise

Our cstartmpi program has another bug: it runs fine for 4 processes, as we've just seen, but at larger numbers it segfaults again.

```
mpirun -np 5 ./cstartmpi.exe
```

```
rank 4 in job 60 tenku_60773 caused collective abort of  
all ranks
```

Now it's up to you to find out why – you can join with your neighbour at one computer to run the program with Allinea DDT and work out what's going wrong and whether you can fix it!

## Hints

- To start debugging with Allinea DDT

```
ddt -start -np 5 ./cstartmpi
```

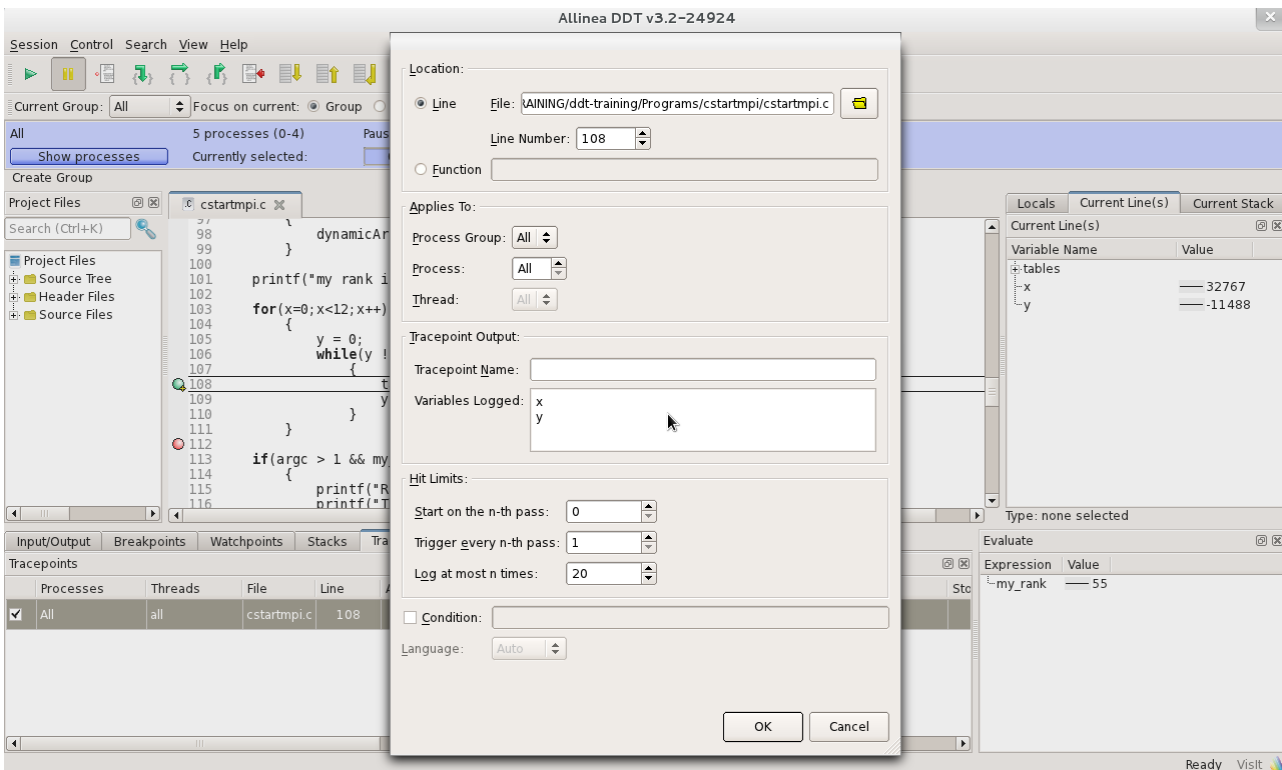
- Click 'Play' to run a program
- Use the stack view to see which sequence functions called each other in
- Click and drag to show variables from many lines in the current line view
- Why didn't the loop terminate? Why did it terminate for the other processes?

If you are tempted to use some “print” statements, it might be interesting for you to try the tracepoints. They can be useful for example:

*Go to the “Tracepoint” tab and right-click to “Add a tracepoint”.*  
*Add the tracepoint on line 108 on variables x and y for all the processes and for a few steps (see picture below).*  
*After the form is correctly filled in, click on add.*

*Then, create a breakpoint right after your tracepoint and run your program.*

*The output will be generated in the “Tracepoint output” tab.*



*Illustration 2: Add a tracepoint in Allinea DDT*

**NOTE:** Allinea DDT can also evaluate expressions. For instance, you can try to trace the following : “y<12”. You will then retrieve binary values (0 and 1).

## Walkthrough

We have previously seen how to get started with Allinea DDT in the interactive mode. As you know, Allinea DDT also integrates offline capabilities.

Using the help provided in the command line, find out how to get started in offline mode.

It is very easy to run Allinea DDT in a non-interactive mode and to create an HTML report. To do so, just start your job using the following command and create an HTML report :

```
ddt -offline report.html -n 4 ./cstartmpi.exe arg1 arg2
```

Allinea DDT will re-use the latest automatically saved session for that particular example. If you have created some tracepoints and breakpoints, they will appear in the report.

Now, open the report that has been generated :

```
firefox report.html
```

The report is divided into 3 parts that you can browse :

- messages : these are the different events that occurred (breakpoints, errors...)
- tracepoints : the values that have been traced during the run
- input/output : the standard printed values to stderr and stdout.

If you wish to use a specific session file, you can do the following.

```
Open your program with Allinea DDT in interactive mode as we have  
seen previously.  
Create breakpoints, tracepoints in your interactive session.  
Click on "Session" and "Save session". Name it "cstartmpi.ddt"  
Close Allinea DDT.
```

Now, start Allinea DDT in offline mode and use this session file.

```
ddt -offline report2.html -ddtsession cstartmpi.ddt -n 4  
./cstartmpi.exe arg1 arg2
```

The report that is now generated will include the information that you provided in your session file.

## Exercise

We have seen how to use offline debugging using automatically and manually saved sessions. Now, we want to provide some specific instructions to Allinea DDT in command line only.

In this exercise, you have to create a report containing a breakpoint and a tracepoint that have been defined in CLI only.

### Hints

- Get some help about the CLI syntax from Allinea DDT

```
ddt -help
```

- Check the key words : "-break-at" and "-trace-at".



## Session 2: Memory Errors

Memory errors often result in a segmentation fault but can be complex to track down – as a problem will not always be triggered – it depends often on the data set for example. Sometimes they simply cause subtly incorrect results – the worst outcome in a scientific application. Overwriting memory can also cause errors to be reported later in execution, or another problem is freeing memory twice.

Allinea DDT comes with built-in memory debugging, which we'll use in a moment.

We'll start with a step-by-step walkthrough. Then move on to an exercise again.

## Session 2: Walkthrough

In this example we will use the trisol code in the trisol directory.

```
cd trisol
make
```

Run it:

```
mpirun -np 4 ./trisol.exe
```

The application will – most probably – crash, although being a memory error you might be running on a system that does not trigger the problem.

Let's try it in Allinea DDT (with memory debugging disabled).

```
ddt -start -np 4 trisol.exe
```

Allinea DDT will start the code as expected.

*Click on 'Play'*

Depending on your luck, Allinea DDT may or may not be able to run the code through to completion successfully. You may see no error – in which case the code runs to the end, or you may find an error triggered.

This is often the case when a memory error is at work; but fear not, we just need to turn up the dial on Allinea DDT's memory debugging.

In order to enable memory debugging with your program, the application need to be linked with Allinea DDT memory debugging library. Although this is done automatically and transparently on most systems, some specificities apply when it comes to Cray systems. You need to manually modify the Makefile to link your application towards the appropriate library contained in /path/to/ddt/lib/64/.

If you are working on a Cray system, please first re-link your application.

*Find out what library you should use.  
Modify the Makefile accordingly.  
Relink your code*

Once this is done, let's start again.

*Click on the “Session” menu, and select “New Session/Run”.*

Before we start the new session, we need to configure the memory debugging settings for this run.

*Tick to enable memory debugging, and then click on “Details”.*

There are now some further options to choose from to get the right amount of debugging help. The higher levels use more memory and are slower than the lower levels.

*Ensure that the options to “Preload” the memory debugging library, and for Fortran/C are selected.*

*Choose “Thorough” for the level of checking. Leave “Guard Pages” unchecked.*

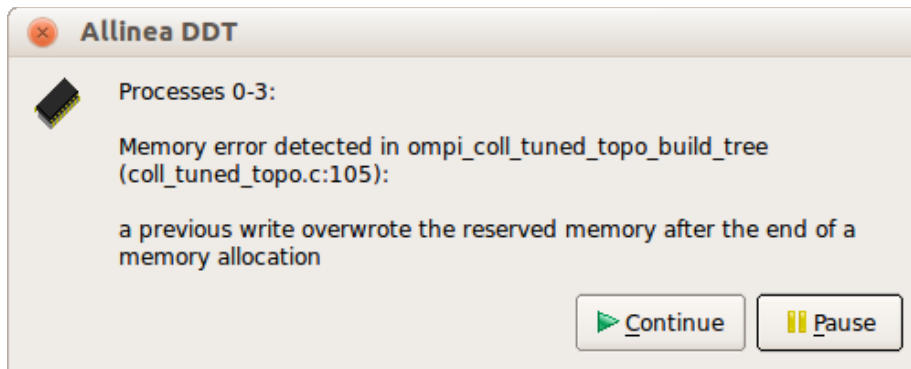
*Click “Ok” to confirm your settings.*

We're now ready to debug again.

*Click “Run” to start your code.*

*Click “Play” to run through to the end, or first error.*

You should now see a memory error appears.



*Illustration 3: Allinea DDT detects something has overwritten a value*

Allinea DDT explains that at some point, some reserved memory that Allinea DDT added beyond your array has been changed unexpectedly,

So, we have an array overflow error, then. But where?

To find out, we need to turn on an advanced feature: Guard Pages. As before let's get the settings ready and restart.

*From the Session menu, click New session/Run.*

*Change the Memory Debugging Settings and enable the Heap Overflow/Underflow detection option. A setting of “1” and “After” is appropriate for this example.*

*Now run the application as before and click “Play” once it has started.*

The Heap overflow/underflow detection adds an extra buffer beyond allocated memory (or before, but not both at the same time!) and then Allinea DDT can detect any writes **or reads** in this zone.

Allinea DDT should now detect a memory error – that a process tried to read/write after the end of the array!

*Click “Pause”*

Now we see the line the problem occurred on.

*Click on res in current line.*

The type of “res” is shown as size 4095, but we're trying to set element 4096. This didn't cause problems until the MPI\_Reduce, but we trapped it at the cause with Allinea DDT.

## Session 2: Exercise

Fix the bug in trisol around line 27 of check.f90 as follows:

```
DO k = 1, block_size
  res(k) = k
END DO
```

Run the program – it may seem to work and print out “Solution correct”, but there is actually still a particularly subtle memory bug still present.

Now run again in Allinea DDT, with the same memory settings as before: what happens? Allinea DDT reports there is a read after the end of the array in matnrm.f line 30.

### **Hint**

Examine variables k and then j2, can you solve the problem?

## Session 3: Incorrect Results

We're going to take a naive C matrix multiplication example and use it to look at some of Allinea DDT's graphing features.

This also tours some of the most important parts of debugging - breakpoints, watches, evaluation and array viewing - classic tools to really dig around in the code.

### Walkthrough

We will work with the matrix example – it is a simple C code to do matrix multiplication. **Note that in this example at this point, the bug is not important, we are only walking through the feature.** Don't worry, there is an exercise with a real bug in a minute!

```
cd Matrix
make
ddt -start ./matrix
```

Allinea DDT will look slightly different from the previous runs, as it is now debugging a scalar (non-MPI application).

Hunting down the source of invalid results is a much freer activity than tracking down a crash. You often want to explore bits of code in more detail. Allinea DDT includes several features to make this easier. One is 'run to here' - a short-cut to run the program until all processes reach a certain point.

*Right-click on line 29 and choose run to here.*

You'll notice we can explore arrays in the locals / current line, but there is a better way to look at larger arrays.

*Switch to the current line view, drag A in the source code into view.  
Now expand A.*

You can also keep an eye on an actual element within the array by using the Evaluate window.

*Drag and drop B[0][2] from the Current Line tab to the Evaluate tab  
(bottom right of the Allinea DDT screen by default).*

You can edit values as well with Allinea DDT.

*Right click on B[0][2] in the Evaluate tab and select Edit Value.  
Give B[0][2] a new value.*

The Evaluate tab lets you also enter arbitrary expressions – like  $B[0][2] + A[1][1]$  – in addition to simple values.

*Step into `init_array(B, 2)`.  
Step over a few times.*

Observe how the value of  $B[0][2]$  is updated.

There are however better ways to look at a whole array.

*Right-click on B in the source code (line 20).  
Select View Array.  
Enter B[\$i][\$j] as the expression and give i and j some bounds (they are inclusive bounds).*

Allinea DDT also lets you visualize arrays.

*Click on Visualize in 3D*

The 3D view shows the array in current state of being assigned to.

*Step out to finish the function `init_array`.  
Click Evaluate in the “Multi-Dimensional Array Viewer” again and then Visualize*

The array B is now being shown as it stands after it has been initialized.

*Use step over to show stepping to the next line without going into any functions*

Another great feature debuggers have is being able to stop as soon as a value of data changes. This is a feature that the hardware adds to help debugging.

*Raise the “Watches” tab at the bottom part of Allinea DDT.  
Right click and select “Add watch”.  
Enter the expression `C[4][4]`  
Press play.*

Allinea DDT will stop the program – immediately after the variable is update, which means the line after the line that changed the value is highlighted because highlighting shows the next instruction that will be executed.

## Exercise

Exit Allinea DDT and run the program from the command line.

As you can see, the program runs – but it runs incorrectly. Now it's up to you to find out why!

There are many ways to find the bug. The simplest is to start Allinea DDT and add a watch for `C[1][1]`. This shows that it changes to 1 in the `init` function and gives a hint that this is a bad thing (0 is expected).

Alternatively, just looking at the C array any time before or during calculation should suggest what's happened.

## Session 4: Memory Leaks

Many languages allow you to allocate memory yourself – this area of memory is known as the “heap”. Allocations on the stack (local variables) are automatically deallocated when they're no longer relevant, but allocations on the heap (pointers, allocatable arrays) are not.

If this happens in an iteration loop for example, then your program keeps on consuming more and more memory – this is a leak. Tracking down memory leaks on your own is awful, but with a memory debugger, it is easy.

We'll walk through two ways to look for memory leaks with Allinea DDT – a general one that (almost) always finds the problem, and a two-minute version that's usually good enough.

Then there's an exercise for you to work on!

### Walkthrough

#### 1. Compile and run the mandel example program

```
cd mandel
make
mpirun -np 4 ./mandel
```

The program runs fine – no major issue. However, under the hood there was a memory leak - we don't see it here because we're running at modest scales but it uses twice as much memory as it should. We'll use Allinea DDT to diagnose and fix this problem.

Open Allinea DDT without using -start:

```
ddt ./mandel
```

We want to change the memory debugging settings again before running the program. Click on run, advanced, settings.

This program makes a lot of memory allocation/deallocation calls, so to make sure it runs quickly we'll turn off the extra debug checking; we're only interested in leak detection anyway.

*Using the advanced button on the run dialog, go to the memory debugging settings.  
Change Heap Debugging to Fast and turn off Heap Overflow Detection .*

Once these changes are made, we're ready to start debugging again.

*Return to the run dialog, set the number of processes to 4.  
Click Run.*

There are lots of ways to debug a memory leak, but I'm going to show a simple technique that works well with most HPC codes. First, we find the main iteration loop (there's almost always one in HPC). Normally you know where it is. In this case, it's at line 64.

*Scroll to line 64, or use Search->Go to line and type in 64.*

What we want to do is run the loop a couple of times, check the memory usage, then run it a few more and check again. If the memory usage is growing, that's a strong sign of a leak. Anything that's being allocated every loop without being deallocated again is a potential problem.

We'll start by putting a breakpoint at line 64. Breakpoints cause processes to stop when they reach a location. You can also add conditions to this if you want – so that you only stop if a certain condition is true – but we don't need to do this today.

*Right-click on the line and choose "Add breakpoint for All".  
Now click "Play"*

A dialog tells you the program has stopped at the breakpoint.

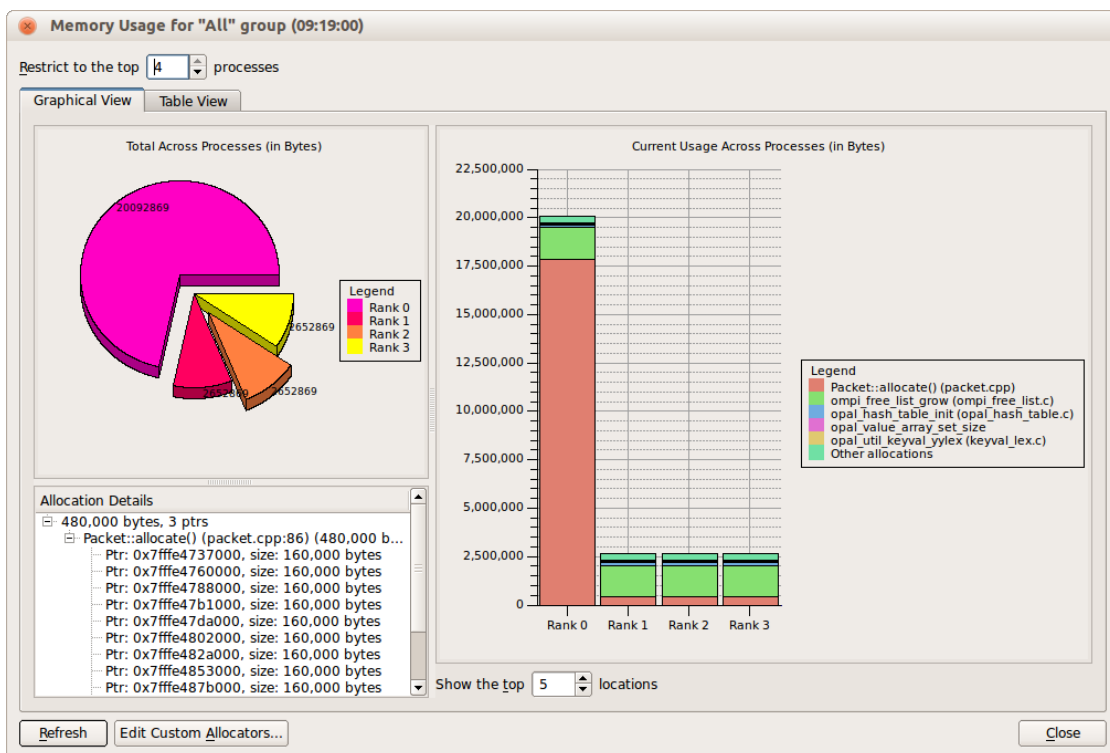
We're going to let the loop run a few times in case it does something special the first time.

*Click Continue a few times.  
Now click Pause.*

We will now look at the current memory usage.

*Select "View" and then "Current Memory Usage".*

After a few seconds to gather all the information, Allinea DDT now shows some charts describing the current memory usage. On the left, total memory used in each process is shown, and the view on the right does that too and more.



*Illustration 4: Current Memory Usage – Process 0 shows a leak*

Each process (0-3) has its own bar, the blocks represent lines on which memory was allocated. This is important, because lots of tiny allocations (such as one every iteration) are grouped together so that they stand out instantly.

Memory allocated in Packet::allocate() (reddish brown) is the most significant.

*Click on a block to see the collection of pointers allocated from this point.*

The allocation details pane on the left shows all the pointers Allinea DDT has seen allocated from that line of code. In this case, several different pointers. One is huge, the rest are of moderate size. This is just a baseline measurement, before we did much iteration.

*Move the memory usage window to one side and go back to Allinea DDT.*

Let's iterate a couple more times to see what changes.

*Click play / continue another 5-6 times, then pause.  
Click refresh in the usage window*

The total allocated has increased slightly; this suggests a memory leak.

*Click on a reddish brown bar to see which new allocations were made.*

We can see that several new regions of size 160k were allocated; one for each iteration. To solve the memory leak, we should find out what these are and why they are not being deallocated. Allinea DDT can help.

*Click on one of the size 160000 lines in the Allocation Details list.*

Now we're examining one specific memory allocation made during execution. The Pointer Details window shows the stack at the time it was allocated - which sequence of function calls resulted in the allocation. In this case main called strategy1, which called Packet's receive function, which called its allocate function.

Move the windows out of the way a bit and see what's going on.

*Click on each of the stack lines in turn and see as Allinea DDT shows that place in the code.*

To analyse, start from the bottom. Packet calls allocate() to assign memory to its iterations variable. Is this ever freed?

You can either scroll around the file and look, or - if you know some C++ - you might want to check the destructor. We can simply look for free() calls and see if iterations is freed at all.

*Click Search and then Find.  
Search for 'free' by pressing enter.*

No matches. The destructor is empty, too, which looks like a bug. Really the destructor should free the memory, if it has been allocated.

Now in the general case you may need to do this - look at individual iterations, but often you can short-cut the whole process by just running to the end of the program and looking at the memory usage. Because Allinea DDT gathers lots of little related pointers together, it's usually still easy to see the difference between a leak and normal usage.

*Remove the breakpoint by right-clicking on its line and choosing Delete breakpoint for All.*



*Ensure Allinea DDT stops in exit() by ticking the Control/Default Breakpoints options for abort and exit.  
Press Play*

Allinea DDT will stop the program just before it exits. When the window pops up, click 'Pause'. Now let's see what's left over after everything that should have been collected is collected

*Select View and then Current Memory Usage*

As you can see, there is a lot of wasted memory. Often it's worth starting by looking at the state at the end and only going into the iterations if it's not clear which parts are growing each iteration.

## Option - Session 5: Deadlock in MPI

Early versions of MPI programs often get into deadlock – things such as each process waiting for another, communicators are not matched up or tags not matched. In some cases, livelock happens too – where processes are communicating but not proceeding in any useful way.

Allinea DDT can inspect the message queues to show which processes are waiting and why (not supported on Cray systems), in addition, simply pausing processes is a good way of finding where processes are.

We'll start with a step-by-step walkthrough and then move on to an exercise again!

### Walkthrough

First we will build the cpi example.

```
cd cpi
make
```

Run it with 4 processes

```
mpirun -np 4 ./cpi
```

It works fine

```
.....
pi is approximately 3.1416009869231249, Error is
0.0000083333333318
wall clock time = 1.268749
```

Now we run with 10 processes and it also works fine.

```
mpirun -np 10 ./cpi
```

The next test is to try 8 processes.

```
mpirun -np 8 ./cpi
```

It locks up!

```
Process 7 on localhost
Process 5 on localhost
Process 6 on localhost
Process 3 on localhost
Process 2 on localhost
Process 0 on localhost
Process 1 on localhost
Process 4 on localhost
```

Press ctrl-c to abort, and let's try it under Allinea DDT.

```
ddt -start -np 8 ./cpi
```

When Allinea DDT returns with your code begin running the program

```
Press the play button.
```

After a while, the program has still not terminated – but the debugger has still not helped yet.

*Press the pause button.*

Examine the source code view and the stacks view. Both are showing that half of the processes are at one location and half at another. Half are in an MPI\_Barrier and the other half are in MPI\_Bcast.

The next challenge is to find out why this has happened.

Let's look at the loop with the barrier. Did every process execute it the same number of times?

*Open the View Menu, and select the Cross Process Comparison tool.  
Ask Allinea DDT to evaluate “ $i \leq n$ ” in this dialog.*

Sure enough, the “barrier” processes are still trying to loop and the rest have already exited.

How many times should each process execute this loop?

*Use the Cross Process Comparison to evaluate  $(n - (myid + 1))/numprocs$*

We see that processes 0-3 execute the loop one extra time. Possible solutions are to move the Barrier out of the loop to a place where it's executed the same number of times by every process, or to modify the loop to make sure all processes execute it the same number of times.

## Option - Session 6: Large-scale debugging

Debugging a 4 procs example or a 1024 procs example is just as easy with Allinea DDT. And it is time now for you to check that yourself!

Let's take the very first example we have been working on. If you have made some modifications and fixed the initial buggy code - this is great! Let's put those bugs back in place.

### Walkthrough

Just as the first time, we need to compile the application `cstartmpi` (with its bugs).

```
cd cstartmpi
make
```

Let's now run the program with 1024 processes :

```
mpirun -np 1024 ./cstartmpi.exe some input arguments
```

Just as before, the program will abort as there has been a problem:

```
rank 0 in job 52 tenku_60773 caused collective abort of all
ranks
```

The next step is to start the program at-scale at 1024 processes. You can place bets with your neighbour and guess how long Allinea DDT will need to start at this scale.

```
ddt -start -np 1024 ./cstartmpi.exe some input arguments
```

As you can see, it is very quick! And now, as easily as when you were working on a few processes earlier today, you can see the source code, the location of the processes, their status, etc.

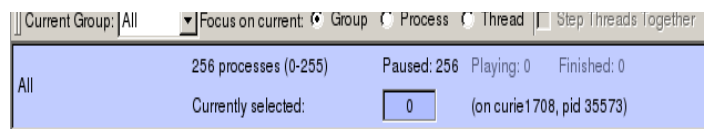


Illustration 5: Summarized view for the processes

```
Click on step and see how long Allinea DDT needs to go to the next line
and to update the variables.
```

Thanks to the scalable tree architecture of Allinea DDT, it is only a matter of milliseconds to step the code.

Now, let's just run the code and see how the bugs that we fixed earlier appear at a larger scale.

```
Check that the breakpoints and tracepoints that you created earlier are
still present.
Press play to run the program.
Allinea DDT stops with an error message – indicating segmentation faults
in different places in the code.
```

As you see, the tracepoints, the Parallel Stack View, the variables... are just as readable as they were previously at smaller scale. Allinea DDT gives you the information you need in any circumstance!

## **Summary**

We have seen a number of features in Allinea DDT that can help you to fix the really common types of problem that occur in everyday development.

This didn't cover everything that Allinea DDT can do for you, but it should give you the confidence to use Allinea DDT and try other features as you become more familiar with it.

For example, Allinea DDT's process groups are a great way of controlling subsets of processes – they're quick and easy to create and let you set breakpoints or step, say, with only a partial set of processes.

Another example is using the attaching feature to attach to a job that is already running.

The userguide gives a more comprehensive look at the features of Allinea DDT and you can get this to appear in Allinea DDT by pressing F1, a PDF version is also available in the doc subdirectory of Allinea DDT's installation.

If there are any questions you have, or problems with using Allinea DDT, please remember that [support@allinea.com](mailto:support@allinea.com) exists to ensure your debugging is successful! We always like to hear from you, as users like you help us to know what is important in our debugger.