DDT Training Day

Reading Materials and Exercises

December 2011
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 DDT tutorial, the tutor will begin by showing you many of the DDT features as a quick overview – the exercises cover a lot of 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 afterwards if there is anything that you still want to know!
Session 0 – Getting Started

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

The examples in this course have all been precompiled for convenience and are provided in a tarball – take a copy of this and extract it into your own account:

```bash
cp ~marko/Programs.tar.gz .
tar zxf Programs.tar.gz
```

DDT works great with both queue submission and interactive modes; today we'll just allocate interactive sessions for everybody:

```
Programs/start_interactive
...
module load ddt
cd Programs
```

None of the examples we'll look at in this short session require memory debugging, but if you want to try with your own codes later then you should be aware that AIX and the Cray XT systems both use static linking and require more involved link commands to get our memory debugging library loaded – there is information in the userguide about how to link in memory debugging support on those two systems.

Cray systems can also be configured to link dynamically, which enables zero-configuration memory debugging.
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 you can try to solve on your own to become more familiar with how DDT works.

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 DDT and find out what happened. The quickest way to start is to run DDT almost identically to the way you launched MPI.

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

The 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 DDT you will see a row of buttons that are used for controlling the execution of the program. As you hover the mouse over a 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.
DDT stops with an error message – indicating a segmentation fault.

The screenshot shows 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 highlighted 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 an invalid pointer (possibly even a null pointer, 0x0) which is breaking 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 in your favourite editor to read “y=0; 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 DDT and work out what's going wrong and whether you can fix it!

Hints

- To start debugging with 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?
**Session 2: 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.

DDT can inspect the message queues to show which processes are waiting and why, 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

```
... 
ipi 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 DDT.

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

When DDT returns with your code begin running the program

```
Press the play button.
```

After a while, the program has still not terminated – let's pause it to see what's going on.
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 DDT to evaluate “i <= 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 \( (n - (\text{myid} + 1))/\text{numprocs} \)

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.

Exercise

Let's look at a new program that also deadlocks. Compile and run the Loop example.

```bash
cd Loop
make
mpiexec -np 8 ./loop
```

It's supposed to pass a message around the loop, but it never finishes!

Kill it and debug it with DDT – try to find what the problem is.

Hints

- A small job is ample to find the cause

  ```bash
ddt -start -np 8 ./loop
  ```

- Investigate the odd process out; what should it have done?

- Think of the example as passing a token around a loop 'max' times. Where does the token start? Where does it stop? What should happen to it at the end?

- Look at the “received” variable in Cross Process Comparison tool, which is the number of times the token has been received.

For an example of MPI ambiguity, replace the BUFSIZE definition with a smaller quantity (~100 instead of 1024x1024) – on most MPIs an MPI_Send of small volumes of data is completed asynchronously! This means the code would terminate successfully, even though we know there is a bug, for smaller message payloads.
**Session 3: Incorrect Results - F90 Example**

The next example is for Fortran users – it has two bugs, both are left as an exercise for you.

The code we will use is the Array example.

```bash
cd Array
make
```

**Exercise**

The code is a simple convolution code. It is an MPI code, although it doesn't do any communication – and one or two processes is enough to show the problem.

A matrix B has a 3x3 so-called convolution matrix M applied to each cell. By this we mean that the new value in matrix C at cell (i,j) is the arithmetic sum of the products of each cell surrounding B(i,j), and B(i,j) itself – with a multiplication mask given by M.

Thus the 3x3 convolution matrix with all values zero except M(2,2), the central element, which is 1, is an identity matrix for convolution.

The developer of this code was kind enough to create and include a test case involving the identity convolution matrix, which applies it to an 5x5 array B – but something strange happens, the output is not the same as the input.

```bash
mpirun -np 8 ./array
```

The output snippets below should match values.

A real convoluted example code
Start of input b

```
1.0000000 0.0000000 0.0000000 0.0000000
0.0000000 ...
```

Output of the convolution of b

```
0.0000000 0.0000000 0.0000000 0.0000000
0.0000000 ...
```

However, they are different. Can you find out why, and fix the problem?

**Hint**

- Use watchpoints to help you determine when the new values are written. There are some cells which you know are wrong, watch one of those.
- Once you have found where things happen, double click on the line at the start of the relevant loop to set a breakpoint there (line 161 looks a good one for this).
- Restart the example and step through the offending loop to see what happens.

You will probably kick yourself when you see the answer, maybe a different font would help – try using the Session/Options/Appearance menu and change the code viewer font settings.
• Be really careful when you fix the problem so that you change the code in both loops – when you assign the elements of C to B, remember which elements you did not compute!

Exercise

The code may now run successfully, but there is still a bug.

This bug might, or might not, appear for you today – it's a memory bug that is triggered at random. The chance of it appearing is related to a number of things – a random number seed, and the machine page size (usually 4k for x86_64 Linux), and the current layout of the heap, the last of which is dependent on the compiler.

One run of this code might, on 1,000 cores, cause a segmentation fault on 10 random cores, which is exactly when you need a real debugger.

On our Open MPI x86_64 system, with the gfortran compiler, one run of 8 processes produced a single segmentation fault. On another run, none of the processes errored.

Disable memory debugging in DDT, and run through to the end on a few cores inside DDT.
Try this two or three times to see if you always get the same processes erroring.

A bug like this would be easier to fix if it happened every time? DDT's memory debugging gives exactly the help we need here.

Can you fix the problem?

Hints

• Using the underflow/overflow detection in the memory debugging settings, find where the program crashes.

• Edit the code to fix the problem, compile and re-run.

• You might notice that there are actually two similar bugs – re-run the code with the “below” protection instead of “above” (or vice-versa, if you started with below initially) and see if the problem shows again.

If you started with “above” guard pages, you might have been shown two different error messages at first – this is because some of the processes crashed before they got to the end of the loop – when they were below the array – which obviously happens during earlier iterations in the loop. Other processes were “lucky” and only crashed when DDT forced the problem to show with guard pages, at the last iterations of the loop. This just goes to show how random memory bugs can be!
**Bonus Session: GPU Debugging**

In this session we will take a look at debugging NVIDIA CUDA with DDT. Any system with CUDA toolkit and driver levels above version 3.1 will work for debugging, although as the systems are still evolving, the newer versions are generally more reliable or support more recent hardware.

The CUDA support is a natural addition to DDT, and fits well in the same way that multiple threads or MPI do. There are additional features to give you more detail about CUDA kernels and we will see these during this exercise.

**Walkthrough**

We will work with the prefix example. This computes the “prefix sum” of an array of integers. By this, we mean that in the output array, the element at position i is the sum of the elements in the input array, up to and including the position i.

Thankfully we don't need to know much about the algorithm here. It's a pretty awkward thing to do with a GPU, but is important for sorting algorithms, for example

```bash
cd Prefix
make
./prefix
```

There is a bit of checking at the end of the code, to test the output, and it fails.

```latex
... 124750 error at element 64
```

We now start DDT as normal, DDT will auto-detect that the code is CUDA.

```bash
ddt ./prefix
Click run to start the application.
Press Step Over twice to see DDT working, as normal, through the program.
```

Sometimes your program will call kernels from places you were not expecting, so it's good to know that breakpoints still work. You set breakpoints by double clicking on a line of source code. There is also a special breakpoint that will stop DDT any time that a kernel is about to start: this is the stop on launch feature.

```latex
Select the Control/Default Breakpoints menu item and ensure that the Stop on CUDA kernel launch feature is enabled.
```

We first start by looking at the CUDA initialization code.

```latex
Right click on the word cudasummer in line 193, and select “View source”.
```

Now we will execute a little further, until after the device has been set up ready for the kernel.

```latex
Right click on line 143 and select “run to here”.
```

You can now see values in the Current Line window for devIn and devOut, the device memory...
locations allocated in the previous lines. These are not ordinary pointers, they're device pointers, so we can't look at their targets until we're inside the device.

This program dumps some output about the device before it starts the kernel. Let's read it.

Click on the Input/Output tab
Everything looks normal so bring the Stacks tab back to the top.

Let's continue until we hit a kernel.

Press Play.

DDT returns control in the zarro function. Let's look and see what's changed in the interface.

The most important things to notice are the (at the top) CUDA thread selector – you can use this to set a particular CUDA thread that you want to examine and control. At the bottom you can see the Stacks view now has an extra column – giving the GPU thread count. You might also notice the “K1” at the top by the thread selector – this means “Kernel 1”. We treat kernels like a regular CPU thread in DDT.

We might want to see what happens when we step – let's advance a single line through the kernel.

Click Step Over.

You should see the colour highlighted line in the source code split to become two lines.

Hover the mouse on each line.

This tells you how many threads – and which ones – are at that particular line of code.
Down in the Stacks view – you can see that 480 threads are at one line, and 32 at another.

You can change to a different thread by entering new values in the thread selector.

Enter \textless 0,0\textgreater \textless 2,0,0\textgreater in the thread selector boxes.

Click “Go”

Notice how x has changed.

You can also switch threads by clicking on a branch in the Stacks view. DDT will change to a thread that is on that line.

Click on the line 90 branch

You will notice that threadIdx.x, for example, is changed.

Press Step Over again.

Now 64 GPU threads are on line 92. This is because CUDA forces 32 threads to step together – a warp. This is the smallest unit of execution that the debugger can control.

Press Play again

DDT will now begin the main kernel (prefixsumblock). Let's check the input is right, because we know the output is wrong somewhere!

Click on the “in” in the Current Line view.

At the bottom you should see the type of “in” - it is a “@global int* @parameter”. This means a pointer to integers that are located in the device global (ie. GPU memory), given as a parameter to the function. If you do the same for “x” – and will see it is in a register.

Right click on “in” in the locals tab.
Select “View Array”.
Enter “in[$i]” as the expression.
Set lower and upper bounds of 0 and 100 respectively.
Click “Evaluate”.

DDT is now examining the device memory.

Scroll through the input to the end.

The input looks sensible.

Repeat the above for the “out” variable.

Scroll down beyond the 64\textsuperscript{th} row.

What do you notice? Garbage in, garbage out – this must be the problem!

Thankfully, with the stop on launch feature, we know that the only kernel that did anything to the data is the zarro one. We'll go back to the zarro code.

Type zarro in the search box, which is above the project files list.
Notice that the assignment didn't properly zero the array! We should have had, inside the if statement, the following code.

```c
data[x] = 0
```

Edit the code, and re-run. The example now works.

**Exercise**

There is still at least one more bug remaining. It's a memory bug – so it won't crash every time but it might cause trouble later.

(Return to the session starting dialog, Click Advanced and enable CUDA memory debugging.)

Once DDT returns, we will remove the breakpoint on every kernel launch, we don't need this right now.

(Select the Control/Default Breakpoints menu. Disable CUDA stop on launch. Press “Play”)

DDT will return a short while later with an error message.

The error message is an exception triggered by CUDA - it means we are reading or writing outside of valid memory.

(Click Pause)

The exact thread and exact location of the problem is now identified. DDT has selected thread (32,0,0) as the first one that had problems. Remember that threads in the other warps could be executing at the same time, it has to pick one, which might not be the first in your mental model of how the GPU progresses!

What can you see? Do you know what has happened?

**Hints**

It helps to know a little bit about what's going on in the code.

Firstly, we compute the prefix sum of blocks of 64 elements in the array, in parallel, independently.

Then, if there is more than one block, we need to “correct” the results – adding the prefix sums of the end points of the preceding blocks. To put this another way, if the block size is 64, then the 3rd block of “sums” must include the last sum of the 1st block, and the last sum of the 2nd block, so that the sum of the first 128 elements is included in the sum for every element of the third block.

This is what is happening in gathersumends – it is collecting up the last-elements of each block – so that we can make the correction (via a recursion prefix sum, later in the code).

- Use the stack view to go to the CPU code and see where kernel size for gathersumends is defined.
- Consider how many blocks there are – ie. how many end elements are there to gather?
• Look at the size of the arrays sent to gathersumends (the devEnds pointer) by examining the definitions.
Summary

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

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

For example, 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 DDT and you can get this to appear in DDT by pressing F1, a PDF version is also available in the doc subdirectory of DDT's installation or from http://content.allinea.com/downloads/userguide.pdf

A more extensive set of training examples from one of our full-day workshops can be downloaded from http://content.allinea.com/downloads/ddt_training.tar.gz

If there are any questions you have, or problems with using DDT, please remember that 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.