DDT User Guide

Barcelona Supercomputing Center

July 29, 2019

Contents

1 Introduction to debugging with DDT

Debugging programs that run on MPI can be fairly cumbersome without the right tools, so we have provided our systems with the DDT program.

DDT is a debugger initially developed by Allinea, now property of ARM. The debugger is specifically designed to be used in HPC environments, as its purpose is to keep track of the state of the program in every MPI node/task it uses.

With DDT you can (but not limited to):

- Interactively track and debug program crashes that may occur on certain nodes.
- Track memory related problems in your programs.
- Use offline (non-interactive) debugging for long running jobs.
- Get more information about crashes.

We'll begin explaining how to set up your environment and job scripts for a simple debugging session.

2 Basic interactive debugging with DDT

To debug with DDT using an interactive session (as if it was a typical debugger), you need to do some things: you need to compile your program with a debugging flag and then modify your job script so your program is launched with the option to connect to the debugger (note that this is only one of the ways you use DDT).

2.1 Compiling your program

To compile your program for debugging purposes, you need to add the following flags to your compiler:

- -g (enabling executable debugging)
- -O0 (do not apply optimizations)

For example, the compiling line would be rewritten in the following manner:

```
$ mpicc application.c -o application.exe  $ mpicc  -O0 -g application.c -o application.exe
```

2.2 Modifying your job script

Your job script needs to be modified so it can launch DDT when the job enters execution in the queue. To do that, you need to specify that you want to connect with the DDT debugger when the job is launched, loading the DDT module and adding these parameters to the line that launches the program:

```
module load DDT
mpirun ./application.exe  ddt --connect mpirun ./application.exe
```

2.3 Launching the job script

Finally, to launch the job script with the debugger you need to load the DDT module, start the program in background mode and then launch the modified job script:

```
$ module load DDT (if not already loaded)
$ ddt &
$ sbatch your_modified_jobscript
```

```
#!/bin/bash

#SBATCH --job-name=armddt
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --time=00:30:00
#SBATCH --output=armddt-%j.log
#SBATCH --error=armddt-%j.err
#SBATCH --qos=debug

module load impi
module load DDT

ddt --connect mpirun ./mmult1_c.exe 1024
```

Figure 1: Example of a job script

We have provided a capture of a real modified job script as an example:

The DDT main screen will appear, but you have to wait until the job enters execution. To do interactive debugging, we strongly recommend using the debug queue as it normally has a shorter waiting time, but remember that you will have limited resources.

When the job enters execution, you will be prompted with the option to accept the incoming connection. It will give you some options before loading the debugger, mainly so it can know if you use OpenMP, CUDA or some sort of memory debugging.

Once the desired options are selected and the program is loaded, you will see the main GUI for the debugger.

2.4 Quick look of common utilities and general usage

DDT largely operates the same way than most classical debuggers for serial applications, with the distinct difference that it can effectively track the state of the execution of every MPI process involved. We've made a general legend of the different utilities present on the main GUI:

1. General debugging actions.
2. Process selector. You can also focus on single processes or threads of a process.
3. Project tree.
4. Code of the selected process.
5. Variable and stack monitoring window.
6. Input/Output and general tracking utility window.
7. Evaluate window (used to view values for arbitrary expressions and global variables).

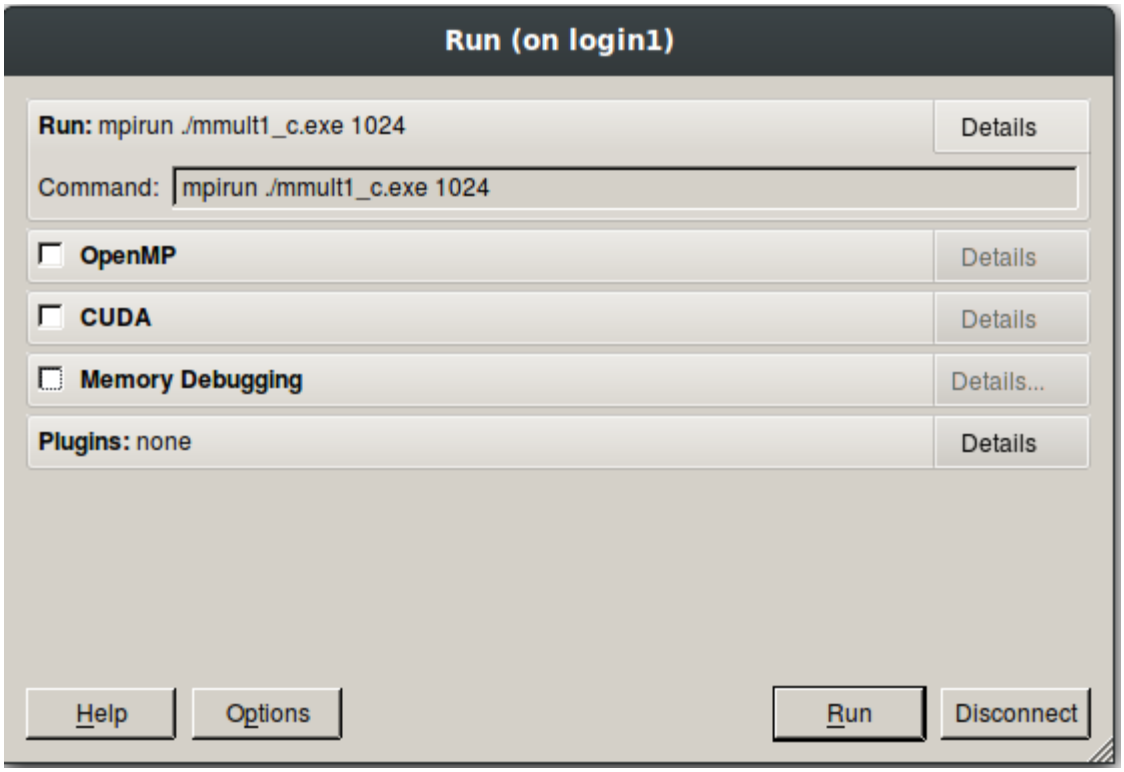


Figure 2: Debugging options

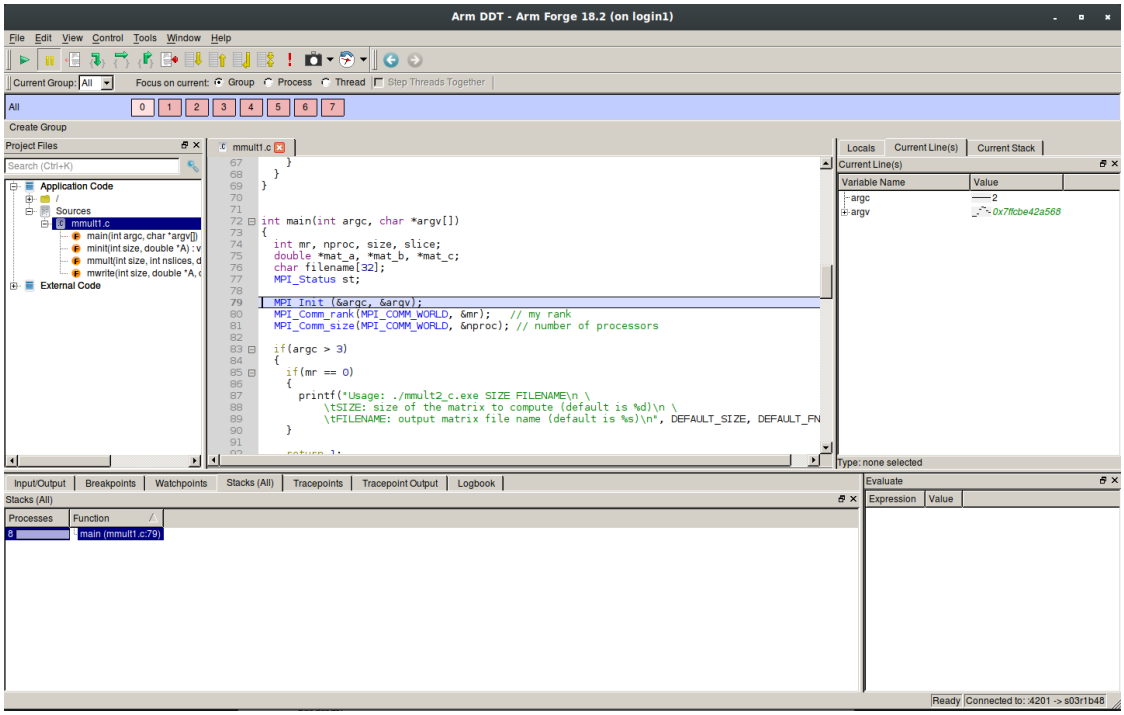


Figure 3: Main GUI

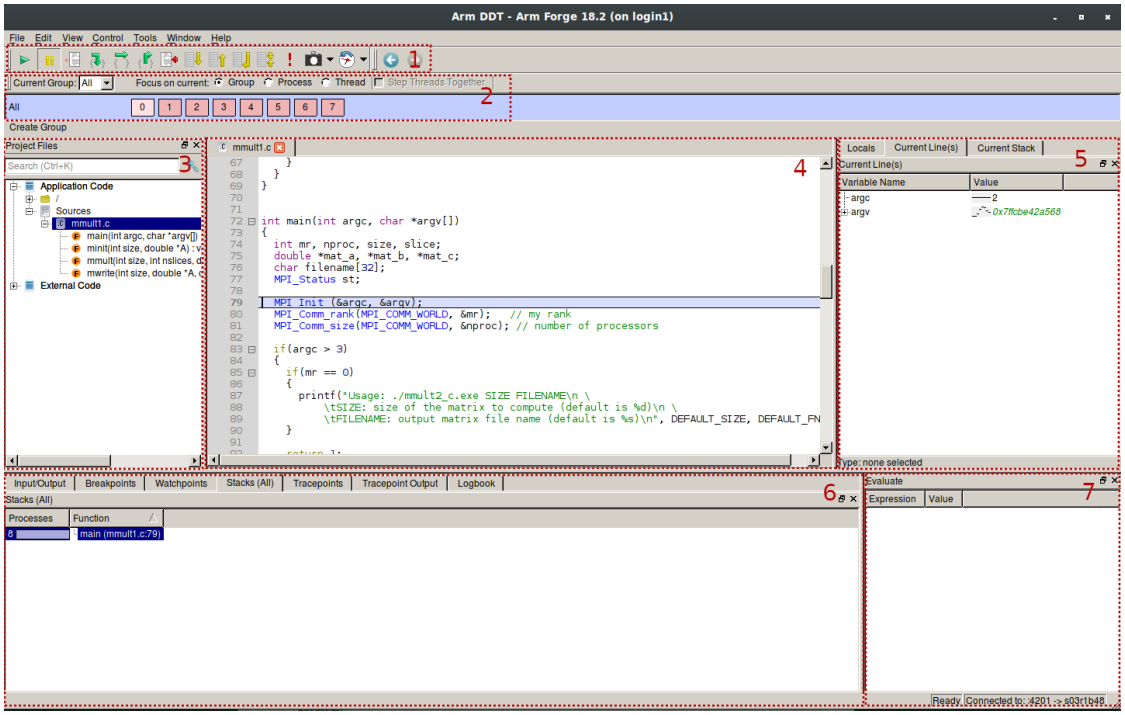


Figure 4: Utility legend

Outside the process selector, everything is like a normal debugger and is used in a similar way.

3 Offline debugging

As we know, jobs can take a while to complete or even get into an execution state, so an interactive debugging session may not be the best solution if we expect them to take some time. DDT offers the possibility of offline debugging, allowing us to come back whenever the execution finishes. The execution will generate a file (either a .html or .txt) where you can check the parameters of the execution and the problems that it may have encountered.

To do it, you need to follow the next steps.

3.1 Compiling the application for debugging

For this step, you have to compile the application applyting the same changes we did in the previous chapter:

```
$ mpicc application.c -o application.exe  $ mpicc -O0 -g application.c -o application.exe
```

3.2 Modifying your job script

Make sure that your script loads the ddt module:

```
module load DDT
```

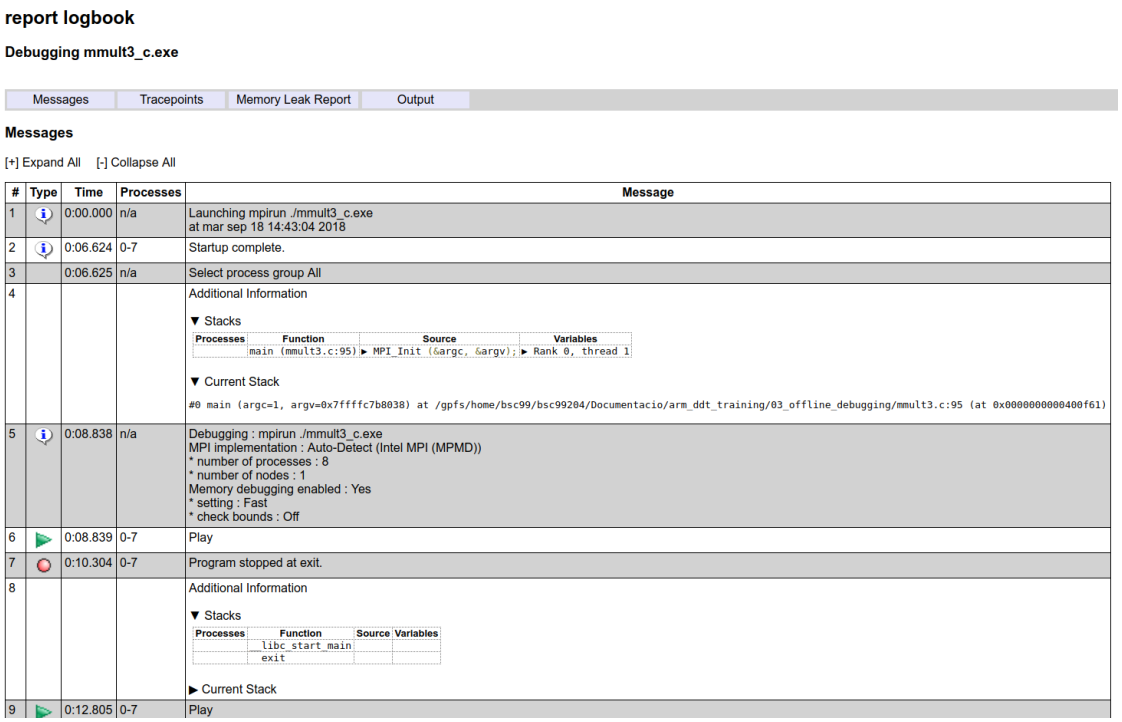
And now, modify your launching adding ddt and your desired flags. Note that you have the option to choose between generating a .txt or a .html. We will generate a .html in this example:

```
ddt --offline --output=report.html  mpirun ./your_application.exe
```

3.3 Launching the job

To launch the job, you just need to launch it as if you were launching it normally. Once the execution finishes, the report file will be generated. If it was a .txt, you can check it on the login node itself. The HTML version is more user-friendly and interactive, but needs a web browser to display it, so you will need to transfer it to your local machine.

Here's an example of a report:



report logbook

Debugging mmult3_c.exe

Messages | Tracepoints | Memory Leak Report | Output

Messages

[+] Expand All [-] Collapse All

#	Type	Time	Processes	Message
1		0:00.000	n/a	Launching mpirun ./mmult3_c.exe at mar sep 18 14:43:04 2018
2		0:06.624	0-7	Startup complete.
3		0:06.625	n/a	Select process group All
4				Additional Information ▼ Stacks Processes | Function | Source | Variables main (mmult3.c:95) ▶ MPI_Init (&argc, &argv); ▶ Rank 0, thread 1 ▼ Current Stack #0 main (argc=1, argv=0x7ffffc7b8038) at /gpfs/home/bsc99/bsc99204/Documentacio/arm_ddt_training/03_offline_debugging/mmult3.c:95 (at 0x0000000000400f61)
5		0:08.838	n/a	Debugging : mpirun ./mmult3_c.exe MPI implementation : Auto-Detect (Intel MPI (MPMD)) * number of processes : 8 * number of nodes : 1 Memory debugging enabled : Yes * setting : Fast * check bounds : Off
6		0:08.839	0-7	Play
7		0:10.304	0-7	Program stopped at exit.
8				Additional Information ▼ Stacks Processes | Function | Source | Variables __libc_start_main exit ▶ Current Stack
9		0:12.805	0-7	Play

Figure 5: HTML offline report

It may have caught your eye that there's a "Memory Leak Report" tab. DDT allows memory debugging with different granularities, which can be really helpful. Let's talk more about that in the following chapter.

4 Enabling memory debugging with DDT

DDT can track down memory related issues like invalid pointers, abnormal memory allocation, memory leaks and more. You can enable memory debugging using two different methods, one for interactive debugging and the other for offline debugging.

4.1 Interactive debugging

You don't have to modify anything for this. When your job requests a connection to DDT, you can check the "Memory Debugging" (which can be seen in Fig. 2) option with the desired parameters.

4.2 Offline debugging

For offline debugging you will need to add a simple flag to the execution line inside your job script. Using the line we used for the offline debugging chapter as an example, add this flag:

```
$ ddt --offline --mem-debug --output=report.html  mpirun ./your_application.exe
```

With this, you should be able to have memory-related information inside your report.

4.3 Static Linking

There's an exception to the previous instructions, and that is when your program has been statically linked. If your program is statically linked then you must explicitly link the memory debugging library with your program in order to use the Memory Debugging feature in Arm DDT. To link with the memory debugging library, you must add the appropriate flags from the table below at the very beginning of the link command. This ensures that all instances of allocators, in both user code and libraries, are wrapped. Any definition of a memory allocator preceding the memory debugging link flags can cause partial wrapping, and unexpected runtime errors.

The required linking flags are the following:

```
LFLAGS = -L/apps/DDT/18.0.1/lib/64 -zmuldefs -Wl,--undefined=malloc,-undefined=_ZdaPv -
    ldmallocthcxx
```

5 Example of a debugging, step by step

To end this manual, we will provide you a code and we will debug it using DDT. You can follow the same procedures that we will show by yourself. You can get the source code here (copy it to your home folder and extract it):

```
(Assuming youre inside your home folder)
$ cp /apps/DDT/SRC/DDT_example.tar.gz ~
$ tar xvf DDT_example.tar.gz
```

Inside the generated folder you will see some source code files (one in C and the other one in Fortran, we'll use the C version), a job script and a makefile alongside a solutions folder.

5.1 Compiling

Our job is to find and fix what is wrong with the source code, so the first step will be compiling our application using our makefile (feel free to check the contents). This makefile has an option to add the required compiling flags for debugging, so we'll take advantage of it:

```
$ make DEBUG=1
```

This will generate the required executable files for when we launch our job script.

5.2 Adapting the job script

The job script provided is functional as it is, but we will be doing an interactive debugging session, so you could be waiting for a while. To alleviate that, we will be using the debug queue, which shouldn't have too many waiting jobs. To achieve that, add this line to your job script:

```
#SBATCH --qos=debug
```

We're almost ready to launch it!

5.3 Launching the program and the debugger

First we need to load our DDT module:

```
$ module load DDT
```

Once we've done this, we can launch DDT as a background process:

```
$ ddt &
```

As you read before, the DDT window will appear, but ignore it for now. Now it's time to launch our job script:

```
$ sbatch job.sub
```

It may take a while, but eventually your job will enter execution and DDT will prompt you with a little window telling you there's an incoming connection. Accept it. In the next window you don't need to check any box, just press "Run".

5.4 Locating the issue with the debugger

First of all, lets talk a bit about the program we are launching. It's a matrix multiplication implemented with MPI, following this algorithm:

1. Master initializes matrices A, B and C.
2. Master slices the matrices A and C, sends them to slaves.
3. Master and slaves perform the multiplication.
4. Slaves send their results back to master.
5. Master writes the result matrix C in an output file.

Here you have a diagram showing the data distribution:

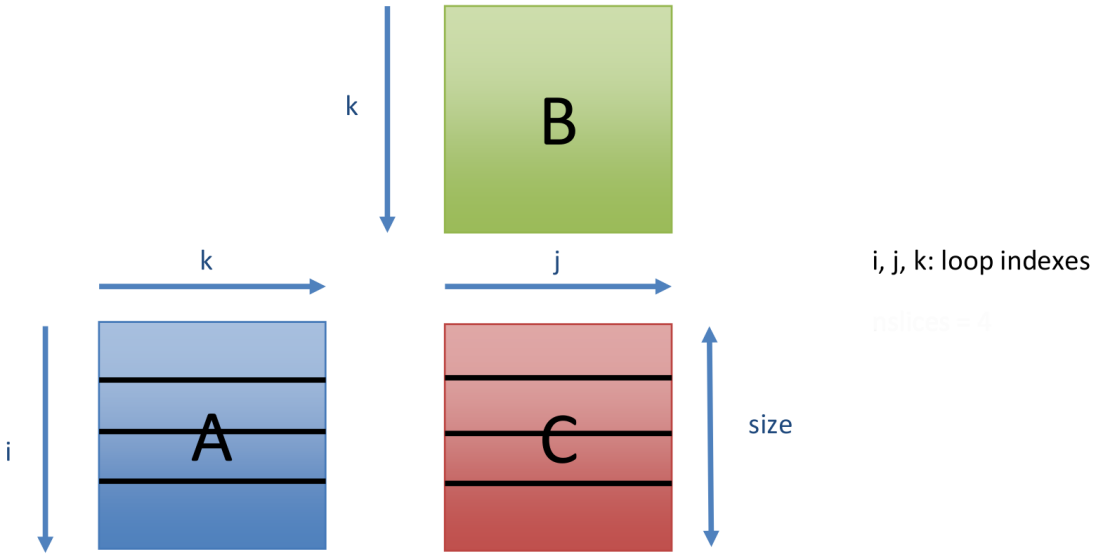


Figure 6: Data distribution

Reading the code you can see the detailed implementation. To see if the program works, we can just execute it without any break point. Let's do that:

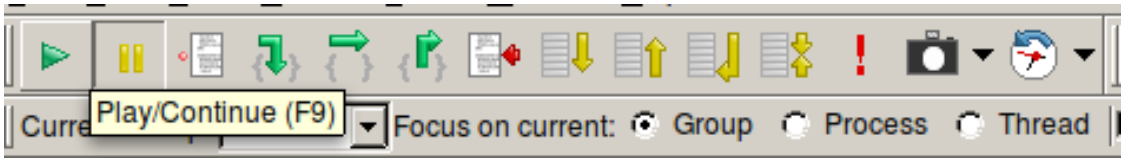


Figure 7: Program flow control bar

If everything works as expected (which is, that it isn't really working), we should see that DDT prompts us with a notification that our program received a signal (SIGFPE, arithmetic exception) and stopped.

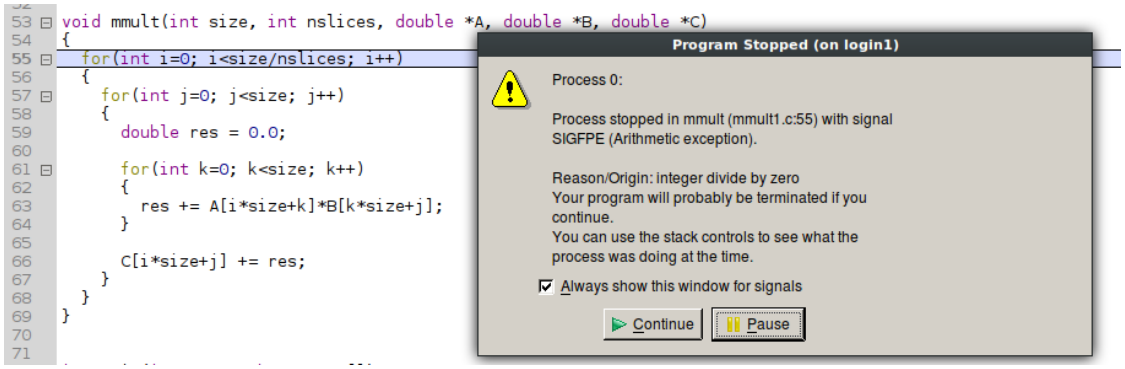


Figure 8: Program crash

DDT will give us some hints. The first one is the nature of the problem, in this case an integer division by zero. Not only that, it also tells (and shows) the line of code that launched the error. We can deduce that there's something wrong with the operation *"size/nslices"*.

Using the window to our right, we can check the values of all variables affected by the current line of code, and we can see that the problem resides in the variable "*nslices*", having 0 as its value.

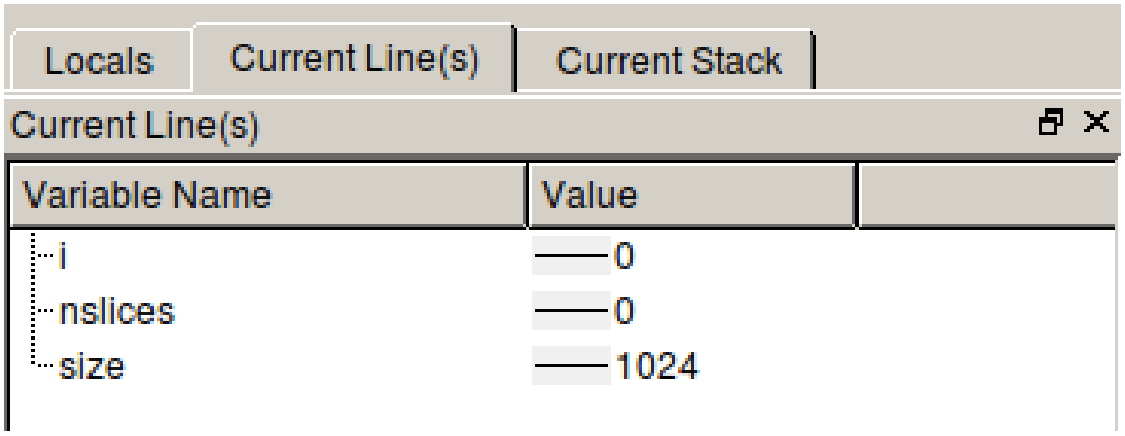


Figure 9: Variable values

The variable "*nslices*" is a parameter given to the function "*mmult*" and it's not changed anywhere inside it. That means that the value provided to our function is incorrect and we should check how the function was called. Looking through the code, we locate it:

```
153
154     printf("%d: Processing...\n", mr);
155
156     mmult(size, mr, mat_a, mat_b, mat_c);
157
158     if(mr == 0)
159     {
160       printf("%d: Receiving result matrix...\n", mr);
```

Figure 10: mmult call

We can see the arguments that this call provided. Specifically, we're interested in the "*mr*" variable, which in theory should be the one defining the number of slices used to divide the partition the data of the matrices.

Inspecting the code, we can see that the "*mr*" variable is **not** what we thought it was. Why? Because we can see that in reality is the variable that holds the identifier of our MPI rank. Our conclusion is that the error is just putting a wrong variable as a function parameter.

```
78
79    MPI_Init (&argc, &argv);
80    MPI_Comm_rank(MPI_COMM_WORLD, &mr);    // my rank
81    MPI_Comm_size(MPI_COMM_WORLD, &nproc); // number of processors
82
```

Figure 11: Getting the process rank

This explains why only process 0 is the only that gives us this problem, as it will be the only one where "*mr*" equals zero. We also know that the right variable is defined in the code, so we only need to find it and put it as the argument inside the "*mmult*" call.

5.5 Fixing the issue

Knowing that this program distributes the data into N slices of the matrices (one for each process), we can use the variable "*nproc*" shown above for that purpose. The only thing left to do is to apply the change to the function call:

```
mmult(size, mr, mat_a, mat_b, mat_c);  mmult(size, nproc, mat_a, mat_b, mat_c);
```

And with this, the program should work now. Let's recompile it and launch it again following the same steps we did for the first version, compiler and all. Once DDT is up and running, we can directly click the continue button. This time, DDT shouldn't give us any problems and the execution should end normally, as shown see here:

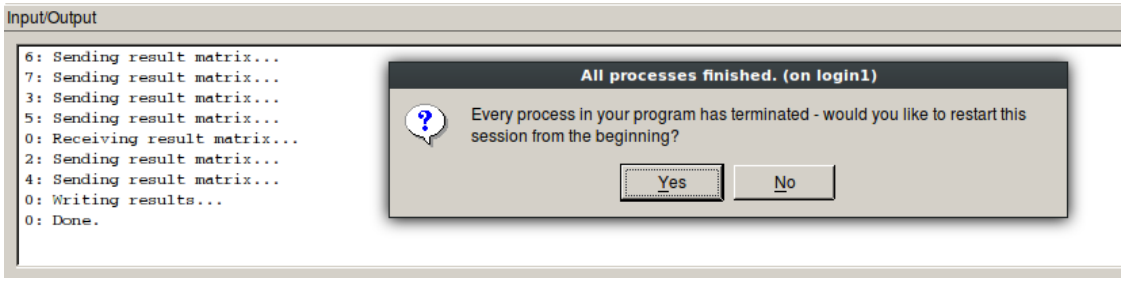


Figure 12: Program termination without problems

And this is it. We've debugged our first application! Although it is a rather simple application and fix, it's a good exercise to grasp the methodology to use with DDT. We hope you find it useful in future debugging sessions.

6 Where can I know more?

If you need more information about DDT and how to use it, check the reference manual:

DDT Documentation[1]

[1] https://developer.arm.com/docs/101136/latest/ddt