University of Hamburg
Department Informatics
Business Informatics, UHH

Debugging

Seminar paper

Seminar: Efficient Programming

Philip Kohlmann
Matr.Nr. 6901410
Philip.Kohlmann@informatik.uni-hamburg.de
Supervisor: Eugene Betke

21.02.2018

Abstract

The following paper gives a wide overview over the field of debugging. It covers basic topics such as dynamic debugging and debugging data formats and explains why debugging in general is important and how to use certain debugging techniques efficiently. Also a variety of different debugging methods for single process applications as well as for parallel programs are being presented, such as the GNU-Debugger, Valgrind and the Distributed Debugging Tool.
This paper is of special interest for beginners in the field of programming as well as for people, who want to learn about the basics of modern debugging.

Contents

1 Background Information

"Debugging refers to the process of finding and fixing Errors or problems within a computer program, which prevent a computer software or a system from working properly"[1]

There are several reasons, why debugging is very important: Most of all to make finding bugs and errors in a program easier and to help resolving these. Also debugging can be used as a tool to understand programs and processes better. In the upcoming sections I will explain with examples how Debugging can be helpful for especially those two reasons.

2 Debugging Data Format - DWARF

A debugging data format is a tool for the storage of debugging information for a compiled computer program. This information can be activated with certain commands while compiling. For example -g, -gdwarf2, -gdwarf3. This debugging information needs to be activated if a debugger shall be able to debug a program and can help to localize problem areas in a program or to find the stacktrace in case an error occurs.

DWARF is the more recent and most commonly used format for saving debugging information. The debugging information is saved in sections of the object file. In general this information resembles a relation between the executable program and the source code and can be imagined as a mapping between those two. [11] Figure 1 shows a part of the debugging information, which is called linenumber mapping. When a program using DWARF as debugging format is being debugged, this mapping lets the debugger know which address in the executable program resembles what line number in the source code. This can for example be useful when the program crashes and the debugger wants to let the programmer know in what particular line of the source code the error occurred.[8]

```
CU: /home/eliben/eli/eliben-code/debugger/tracedprog2.c:
File name           Line number    Starting address
tracedprog2.c                 5           0x8048604
tracedprog2.c                 6           0x804860a
tracedprog2.c                 9           0x8048613
tracedprog2.c                10           0x804861c
tracedprog2.c                 9           0x8048630
tracedprog2.c                11           0x804863c
tracedprog2.c                15           0x804863e
tracedprog2.c                16           0x8048647
tracedprog2.c                17           0x8048653
tracedprog2.c                18           0x8048658
```

Figure 1: Linenumber mapping in DWARF - https://eli.thegreenplace.net/2011/02/07/how-debuggers-work-part-3-debugging-information

3 Debuggingtools and Debuggingmethods

3.1 Debuggingprocess

The first step of debugging usually is trying to reproduce the problem. This can be a difficult task sometimes, especially when working with parallel processes or in case uncommon errors occur. After the error has been reproduced, it can be of help to simplify the program. For example could just a few lines of the code be enough to reproduce the error. In that case it safes a lot of time to just run the few lines instead of the whole code all the time. After simplifying the test case, the programmer can use a debugger to inspect the program and find the origin of the errors in order to resolve them.[3]

3.2 Dynamic Debugging

"*Dynamic debugging means the debugging of a program while it is running. Optional a dynamic debugger provides a console for interactions with the system.*" [4]

Dynamic debugging is the most important debugging technique and is used to find runtime errors that occur in a program. Those errors can for example be null pointer exceptions when a reference in the source code is set to a wrong location or arithmetic exceptions.[2] Tools of the dynamic debugging are the control of the program flow with breakpoints and the usage of single-step functionality.[9] Additionally dynamic debuggers offer ways to put out the values of variables or the program stack, while the program is running. In the following sections two debugging tools, that can optimize programs in different ways, are being presented.

3.3 GNU Debugger (GDB)

The GNU Debugger is a popular debugging tool for Windows and Unix systems. It offers various functionalities to track and change the execution of computer programs like setting breakpoints, single-step functionality and reading the value of variables. The GNU Debugger does not provide a graphical interface and is therefore used over the terminal. The figures below illustrate how the debugging of a program with the DNU Debugger works.[5]

```
int inverse(int divisor) {
    int a = 1 / divisor;
    return a;
}

void print_inverse() {
    for( int i = -2; i < 2; ++i) {
        printf("inverse of %d is %d\n", i, inverse(i));
    }
}

int main(int argc, char ** argv) {
    print_inverse();
    return 0;
}
```

Figure 2: Codeexample 1

Figure 2 shows the code example, which will be debugged using the GDB Debugger. The program uses a for-loop to print out the current index together with its inverse on the console with the index running from -2 to 1. The output of the program is seen in figure 3 below.

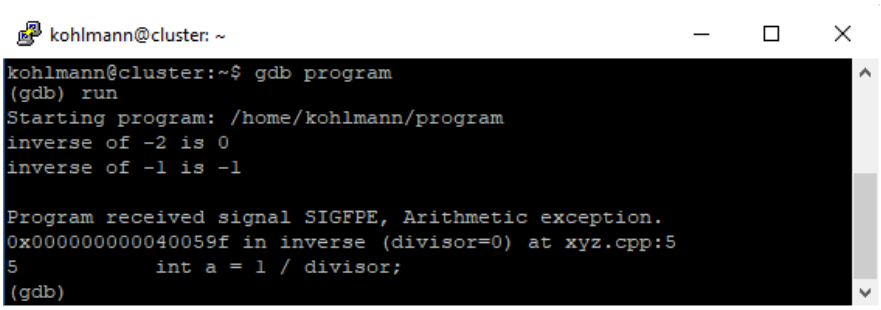

Figure 3: Console output after running the program with GDB

As shown above, the program crashes because of an arithmetic exception. The GNU Debugger now offers different functionalities to locate the origin of the error. Figure 4 illustrates how to find the cause of the error with using backtraces and reading out values of variables.

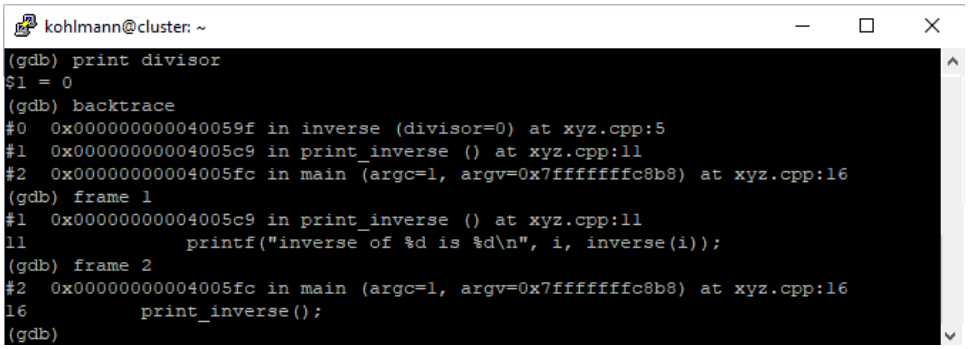

Figure 4: GDB - Print, backtrace and frames

The ”print” command is used to print out the value of a specific variable. In the example above the variable named divisor is specified, since it was responsible for the crash. The terminal now shows that at the moment of the crash the variable divisor had the value of 0. If we take a look at the source code now, we will see that in the inverse() function 1 is being divided by the divisor variable to get the inverse of the current index. At index 0 the program will crash because it is forbidden to divide by 0. To get further information about the program the ”backtrace” command can be used. It displays the program stack with the corresponding frames. To get more detailed information about the functions listed in the stack, the ”frame” command can be used to jump into the respective frames to show their source code. Figure 4 illustrated how the origin of the error could be found using print and backtrace commands. Another way for finding the error is the usage of breakpoints and single-step functionality as shown in figure 5.

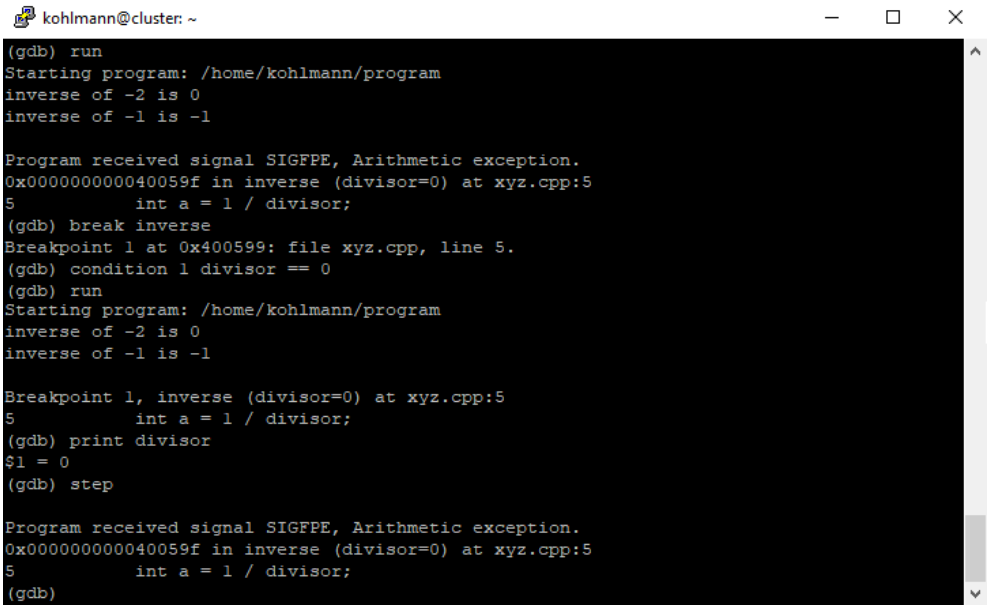

Figure 5: GDB - Breakpoints, conditions and single steps

The "break" command can be used to set breakpoints at specific locations in the program. The location can either be specified with the respective line in the source code or as shown in figure 5 with the actual name of the function. Afterwards conditions for the breakpoints can be set using the "condition" operator. This can for example be very useful when there is an error containing a for-loop, because it is more efficient to stop only at the problematic index of the for-loop instead of stopping at every single iteration. In the example above the condition will make the debugger stop at the breakpoint only, if the divisor variable is equal to 0. After running the program again it stops at the breakpoint and the programmer is now able to read out values of specific variables again or to use the single-step functionality to manually navigate through the program using the "step" command. In this example going one step further leads to the program crashing again because the breakpoint was set right before the action, that would divide with 0.

3.4 Valgrind

Valgrind is a toolbox for debugging, profiling and the dynamic error analysis of programs. Those tools are able to localize a variety of usually difficult to find errors like for example the usage of not initialized variables. Furthermore those tools can analyze performance problems very well.[7] The most important tool of Valgrind is Memcheck. It is able to find errors like usage of not initialized memory, read- and write-access on approved memory and most importantly memory leaks. Memory leaks are errors in the memory management of a program, which lead to memory being reserved but never used or freed. Those errors can cause huge performance issues for the computer. In the worst case those memory leaks lead to the program not being able to reserve any new memory and the program crashes.[6] The following figures illustrate how the Memcheck tool can help with finding those memory leaks.

```
int main(int argc, char ** argv) {
    char *p = (char*) malloc(sizeof(char) * 100);
    return 0;
}
```

Figure 6: Codeexample 2 with memory leack

Figure 6 shows the source code for this example. This program uses the malloc() function to reserve exactly 100 bytes of memory and then terminates after. Now the Memcheck tool from Valgrind is used to debug the program. As seen in figure 7 the 100 bytes, that were reserved by the program were neither freed nor used and therefore the tool detects that memory leak

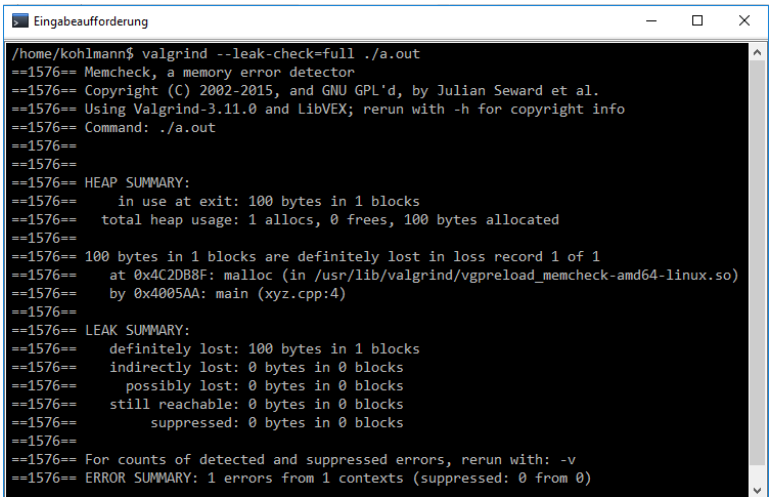

Figure 7: Console output when running Memcheck with memory leaks

To avoid those leaks the reserved memory must either be used or freed before the program terminates. This can for example be achieved by using the free() function on the pointer that points to the reserved memory as shown in figure 8.

```
int main(int argc, char ** argv) {
    char *p = (char*) malloc(sizeof(char) * 100);
    free(p);
    return 0;
}
```

Figure 8: Codeexample 2 without memory leak

Running Memcheck on the edited source code in figure 8 we get the console output as seen in figure 9. The reserved memory is now being freed and the tool does not detect a memory leak anymore.

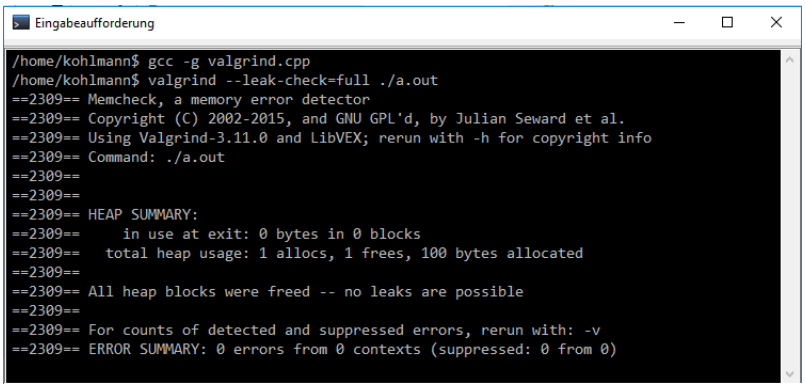

Figure 9: Console output when running Memcheck successful

3.5 Debugging of parallel programs

In the previous sections the paper presented methods for the debugging of single process applications. In this section a way, to debug parallel programs will be presented. The debugging of parallel programs is in particular important to fully understand the processes and the flow of the program in general, because it can sometimes be very hard to understand what exactly is happening in the program when many processes are running at the same time. The Debugger that will be used in the example is the Distributed Debugging Tool (DDT). It is able to debug single-process applications as well as parallel programs. DDT offers a variety of features while this paper will focus on those important for the analysis of parallel programs. In contrast to the two tools shown before, the Distributed Debugging Tools offers a graphical interface for interaction.

```c
int main() {
  int tid;
  #pragma omp parallel
  {
    tid = omp_get_thread_num();
    if(tid == PARENT_TID)
      printf("Parent: %d threads running\n", omp_get_num_threads());
    else
      printf("Thread# %d: Hello world\n", tid);
  }
  return EXIT_SUCCESS;
}
```

Figure 10: Hello World OpenMP - https://gist.github.com/romeroyonatan/6a380e189cee4b1290700cbc5259c837

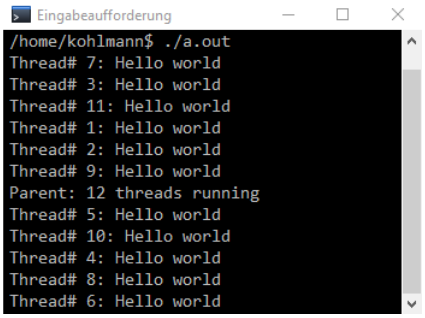

Figure 11: Example console output

Figure 10 shows the program that will be debugged in this example. It is a "Hello World"-program that runs on multiple threads. The program checks if the respective thread is the parent thread or not. The parent thread will write on the terminal how many threads are running in total. Every other thread writes its number on the terminal together with the string "Hello World". An example output of the program is shown in figure 11.

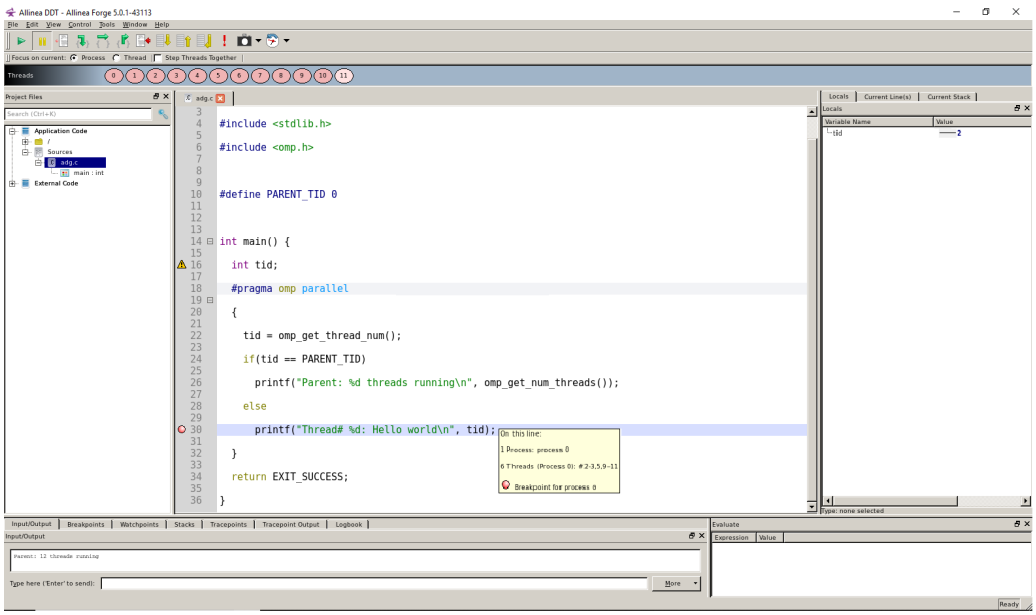

Figure 12: DDT graphical interface for interaction

Figure 12 show the DDT interface. Since DDT is a dynamic debugger the interface has got the typical features of a dynamic debugger. In the top left corner the program can be stopped or single-steps can be made. On the right side the values of the lcoal variables of the respective thread are being shown and at the bottom the programs output and other things like the stack or breakpoints can be seen. In this example the program already stopped at the breakpoint in line 30 and there are currently 12 threads running. Looking at the programs output it can be seen that the parent thread has already run though and made its output on the console. Hovering over certain lines in the source code can help with identifying where every thread is in the program. In the example above. The yellow box shows that there are currently six threads at the line of the breakpoint ready to print out their respective output on the terminal. DDT now offers a handful of features for debugging and analyzing all the threads in the program. It is for example possible to lock at the stack of each thread to compare them.

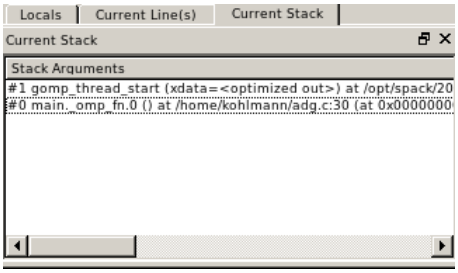

Figure 13: Stack of thread number 11

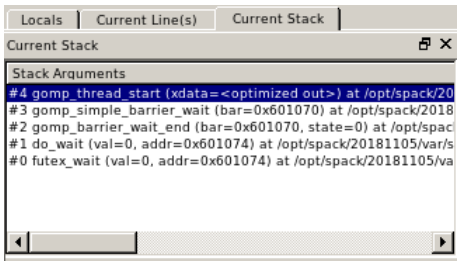

Figure 14: Stack of thread number 4

In figure 13 the current stack of thread number 4 can be seen and figure 14 shows the stack of thread number 11. Comparing both stacks it is easy to notice, that in contrast to thread number 11, thread 4 has not entered the "main()" function yet. That means, thread number 11 has already been started and will print its output on the console in the

next step, while thread 4 is still waiting to be started and will not print its output before that. Looking back at figure 12, the information provided by the "Current Stack View" matches the information given by the yellow box. It also shows, that thread number 11 is currently waiting to call the "printf()" function, while thread number 4 hasn't made it to that line in the program yet. Furthermore DDT offers another tool to not only look at the stack of single threads, but to look at the stack of the whole program as well as seen in figure 15.

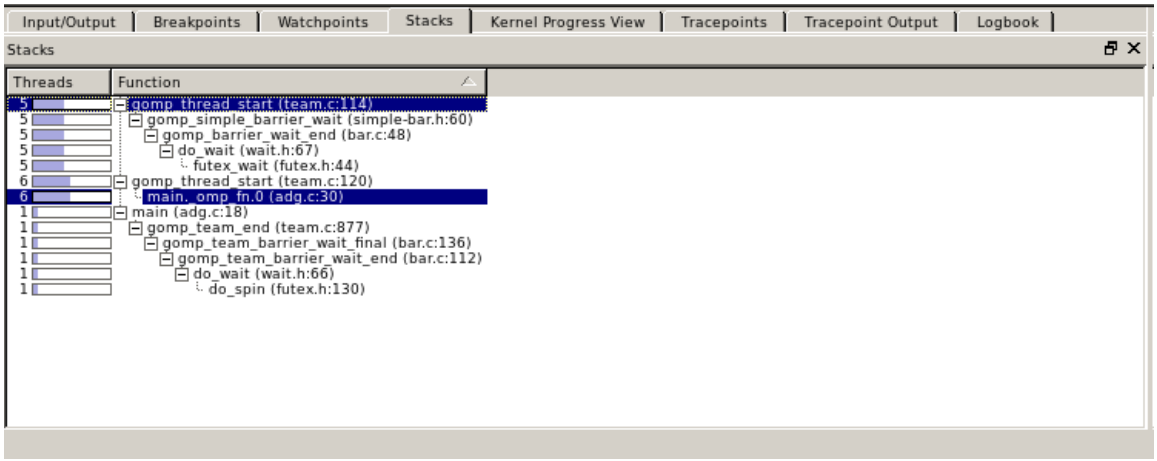

Figure 15: Stack of the entire program

This tool shows that right now 5 threads are in the same state as thread 11, since they got the same stack when compared with figure 14 and 6 threads are in the same state as thread 4, since they got the same stack when compared with figure 13 too.Additionally there is one more thread that has a different stack when compared to the other threads. The "gomp_team_end()" function signals that this thread has already run though and is waiting to be ended. In this example it's easy to say that this thread has to be the parent thread, because it is the only thread, which already printed its output on the console.

There are two more features that can help with debugging parallel programs, which this paper will present with the first one being the "Cross-Thread Comparison View" as seen in figure 16. This tools is able to compare certain expressions like variables, that appear in multiple threads, between those threads. In the example above each thread has a variable called "tid". As shown in figure 16 the variable tid has the value 2 in thread two, three, five, nine, ten and eleven and no value in all other threads. That information matches with all the information we got from the earlier figures. All threads that have a value currently assigned to the "tid" variable, have already been started and will print their output on the console within the next step. In contrast to that, all other threads, that have no value currently assigned to the "tid" variable, are either not started yet or did already end like the parent thread.

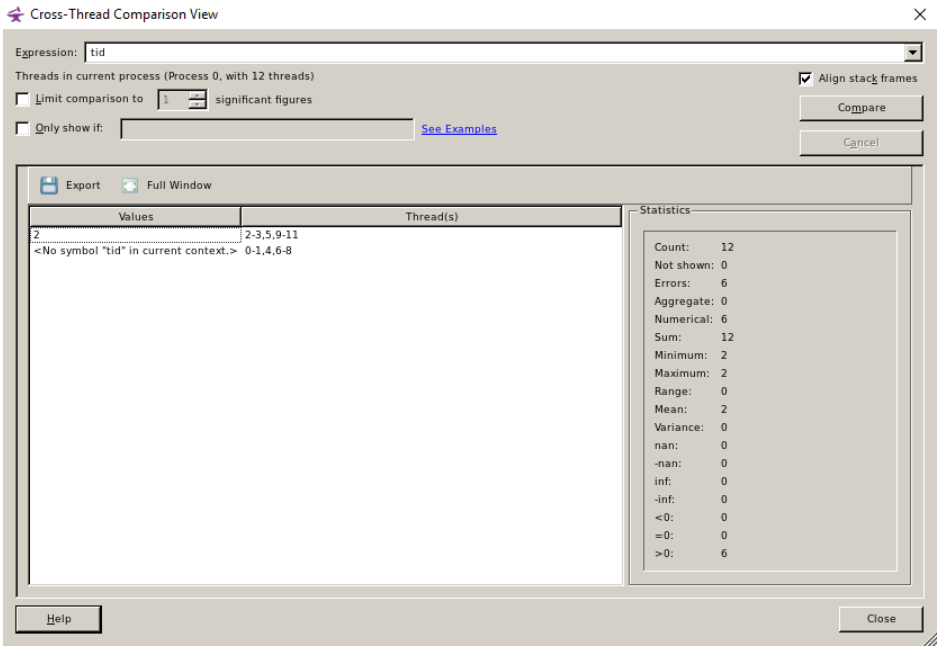

Figure 16: Cross-Thread Comparison View

The last tool this paper will present is the "Message Queues" feature. The example shown in figure 17 has no relation to the codeexample used from figure 12 to figure 16.

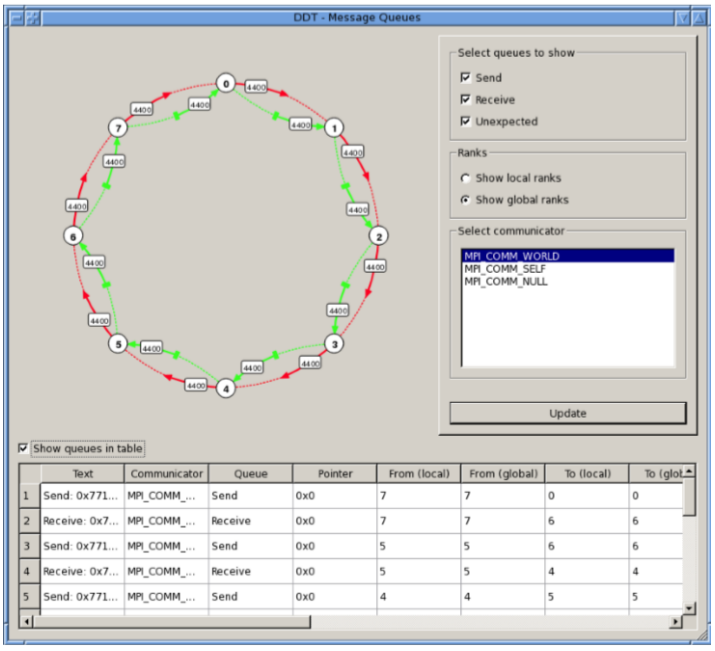

Figure 17: Message Queues - Woo-Sun Yang Debugging with DDT - https://www.nersc.gov/assets/Uploads/Debugging-with-DDT.pdf

In this figure a program is used , that uses a total of 8 threads, which send and receive information from each other. The "Message Queues" tool visualizes those information

flows as a graph and as a table, which makes it really easy to understand the communication between those threads. This tool can in particular be very useful to to detect deadlocks. For example: If the reason for a program freezing all the time is that at some point every thread is waiting to receive information but no thread is sending information, this feature can help with finding the problematic thread(s) without having to dive deep into the source code.[10]

4 Conclusion

In conclusion is it safe to say, that debugging in general is always useful. Especially when dealing with an error that appears only at runtime, since it will not be detected by compiler warnings and is therefore difficult to locate when not debugged dynamically. But even when not dealing with runtime errors, debugging is almost always recommended. It helps with finding errors and resolving those. As this paper showcased, there are many different methods and tools for debugging, which means that there is almost always a tool or method, that can help with the respective problem. Furthermore the usage of those methods and tools is usually easy to learn since there are a lot of documentations and tutorials to be found on the internet. On the other hand there may sometimes be occasions where especially dynamic debugging might not help at all and it would just be a huge loss of time to debug the program for multiple hours. That is why even though there are not really any downsides to debugging, it is recommended to take some minutes and think about where the problem could be and how to resolve it by yourself first before actually starting to debug the program.

References

[1] Debugging. Article on www.wikipedia.org. https://en.wikipedia.org/wiki/Debugging.

[2] Debugging common errors. Article on www.landofcode.com. http://www.landofcode.com/debugging-tutorials/debugging-common-errors.php.

[3] Definition debugging. Article on Economictimes. https://economictimes.indiatimes.com/definition/debugging.

[4] Definition dynamisches debugging. Article on www.igi-global.com. https://www.igi-global.com/dictionary/framework-based-debugging-for-embedded-systems/44531.

[5] Gnu-debugger. Article on www.wikipedia.org. https://de.wikipedia.org/wiki/GNU_Debugger.

[6] Speicherleck. Article on www.wikipedia.org. https://de.wikipedia.org/wiki/Speicherleck.

[7] Valgrind. Article on www.wikipedia.org. https://de.wikipedia.org/wiki/Valgrind.

[8] Eli Benderskyl. How debuggers work: Part 3 - debugging information. Article on www.eli.thegreenplace.net, 2011. https://eli.thegreenplace.net/2011/02/07/how-debuggers-work-part-3-debugging-information.

[9] Markus Dausch. Das debugging dilemma. Published as thesis.

[10] UMCoECAC. Ddt parallel debugging for mpi. Video on www.youtube.com, 2011. https://www.youtube.com/watch?v=Pf4yT3ugT-4.

[11] Adarsh Thampan Suchitra Venugopal. Debugging formats dwarf and stab. Article on www.ibm.com, 2011. https://www.ibm.com/developerworks/library/os-debugging/index.html.