Debugging on MareNostrum: from GDB to DDT

Óscar Hernández (oscar.hernandez@bsc.es)
Cristian Morales (cristian.morales@bsc.es)

PATC Systems Workshop: Programming MareNostrum4
Index

• Introduction

• Debugging basics

• GDB

• DDT
Introduction to Debugging
Motivation

Using debugging tools allows us to…

- Understand application behavior
- Control functionality
- Inspect variable values
- Quickly test different parameters and settings
When to consider a debugger

When a program behaves not as expected

- Try to find the problem in the source code
- Enrich your code with printf’s
- Use a debugger to see internals

Use cases

- Code correction
- Code exploration and learning
- Runtime operation study
- Memory utilization
- Serial, parallel, MPI, threaded (OpenMP, pthreads), GPGPU
Before debugging starts

Environment should be checked…

- **Increase ulimits (`ulimit -a`)**
  - `-s` Stack size (crucial for Fortran and OpenMP)
  - `-t` CPU time
  - `-v` Address space and others
  - `-c` Core file size (crucial for debugging on core files)

- **Remove all objects and intermediate files**
  - Rebuild with debugging info ON (`-g`)
  - optimization OFF (`-O0`)
Basic tools

- Breakpoints, Barrier points, Watch points
- Program control, Src/Asm single-stepping, restart, detach
- Source, stack trace, stack frame (args, locals, registers)
- View/edit data
- Thread control
- Signal sending
- Core file debugging
Gdb

- “GNU Debugger”
- A debugger for several languages, including C and C++
- It allows to inspect the program at certain points during execution
- Errors like segmentation faults may be easier to find with the help of gdb
Gdb’s interactive shell

GDB has an interactive shell, which allows users to:

- Set **breakpoints**
  
  $(gdb) break file1.c:6

- Set **watchpoints**
  
  $(gdb) watch my_var

- **Control execution** workflow
  
  $(gdb) start, continue, step…

- Recall history with the arrow keys, autocomplete with TAB
- Enter without argument repeats last command
- And much more…

- TIP: To get help about a command or more information, use the “help” command, with or without any argument:
  
  $(gdb) help [command]
Breakpoints

Simple ways to set breakpoints are:

- **Breakpoints at specific lines**
  $\texttt{(gdb) break file1.c:6}$

- **Breakpoints at specific functions** (func. My\_function)
  $\texttt{(gdb) break My\_function}$

Once breakpoints are set, control execution with:

- **Start**
  - Starts program execution until a breakpoint (b) is found

- **Continue**
  - Continues a paused execution until the next b

- **Step/Next**
  - Goes to the next line of your execution (more fine grained)
Watchpoint & print

- **watchpoints** act on variables & are defined by:

  
  $(gdb)$ watch my_var

  execution is paused whenever a watched variable’s value is modified.

- **Print** is used to print the value of a variable at a given moment.

  
  $(gdb)$ print my_var

  also, allows to **change** the value of a variable

  
  $(gdb)$ print my_var=15
Useful commands

- **backtrace**
  - produces a stack trace of the function calls that lead to a segfault
- **where**
  - same as backtrace but while program in execution
- **finish**
  - runs until the current function is finished
- **delete**
  - deletes a specified breakpoint
- **info breakpoints**
  - information about all declared breakpoints
Usage compilation

- $ gdb program core
- (gdb) run <arguments> [r <args>]
- (gdb) next [n]  // step over function calls
- (gdb) step [s]  // step into function calls
- (gdb) continue [n]

- (gdb) break [b]
  - (gdb) break main
  - (gdb) break 42
  - (gdb) factorial.c break 12
- (gdb) disable
- (gdb) info breakpoints [i b]

- (gdb) print $VAR [p $VAR]
  - (gdb) print factorial = 1 [p factorial=1]
- (gdb) info variables [i var]
- (gdb) info locals

Quick reference:
http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf
Running an application

janko@ubook:~/patc/debug/0-factorial$ gdb ./factorial
GNU gdb (Ubuntu 7.9-1ubuntu1) 7.9
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
Find the GDB manual and other documentation resources online at:
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./factorial...done.
(gdb) r
Starting program: /home/janko/patc/debug/0-factorial/factorial
Enter a positive integer: 4
4! = 786408
[Inferior 1 (process 4029) exited normally]
(gdb)
4! = 720

#include <stdio.h>
#include "common.h"

(gdb) l
int main(void) {
    // Factorials are only defined for positive numbers and zero
    int num;
    do {
        printf("Enter a positive integer: ");
        num =GetInt();
    } while (num < 0);

    int factorial;
    (gdb) break 18
Breakpoint 1 at 0x4007e5: file factorial.c, line 18.
(gdb) r
Starting program: /home/janko/patc/debug/0-factorial/factorial
Enter a positive integer: 4

Breakpoint 1, main () at factorial.c:19
    while (num < 0);
(gdb)
Reaching a breakpoint

Breakpoint 1, main () at factorial.c:19
19 while (num < 0);
(gdb) print num
$1 = 4
(gdb) info local
num = 4
factorial = 32767
i = 0
(gdb) i b
Num Type Disp Enb Address What
1 breakpoint keep y 0x00000000004007e5 in main at factorial.c:18
breakpoint already hit 1 time
(gdb)  
Further information

(gdb) info address factorial
Symbol "factorial" is a complex DWARF expression:
   0: DW_OP_fbreg -28
.
(gdb) p factorial
$2 = 32767
(gdb) show env
LC_PAPER=es_ES.UTF-8
XDG_VTNR=7
ORBIT_SOCKETDIR=/tmp/orbit-janko
KDE_MULTIHEAD=false
SSH_AGENT_PID=1434
LC_ADDRESS=es_ES.UTF-8
XDG_SESSION_ID=2
TERMINATOR_UUID=urn:uuid:8b3a35b4-e5ea-43fa-b195-7df2fda31d4c
LC_MONETARY=es_ES.UTF-8
GPG_AGENT_INFO=/tmp/gpg-mgupuL/S.gpg-agent:1435:1
TERM=xterm
SHELL=/bin/bash
Changing variables

Starting program: /home/janko/patc/debug/0-factorial/factorial
Enter a positive integer: 5

Breakpoint 1, main () at factorial.c:19
19 while (num < 0);
(gdb) info locals
num = 5
factorial = 32767
i = 0
(gdb) p num
$3 = 5
(gdb) p num=15
$4 = 15
(gdb) c
Continuing.
15! = 881371136
[Inferior 1 (process 4088) exited normally]
(gdb)
GDB in HPC

PROS
- Usually available everywhere
- Simple to manage with
- No interface needed

CONS
- Difficult to debug MPI
- No control of MPI messages
- Lack of interface complicates parallel codes
DDT
Introduction

• Arm (ex-Allinea) DDT is the debugger for C/C++, or Fortran
• Debugs parallel and threaded applications
• Runs on CPUs, GPUs and Intel Xeon Phi
• Extra features:
  • Powerful intuitive graphical interface
  • Automatic detection of memory bugs
  • Divergent behavior and lightning-fast performance
DDT cheat sheet

- Load the environment module (on MareNostrum4)
  
  $$\text{module load DDT/18.0.1}$$

- Prepare the code
  
  $$\text{mpiicc -O0 -g myapp.c -o myapp.exe}$$

- Start Arm DDT in interactive mode
  
  $$\text{ddt mpirun -n 8 ./myapp.exe arg1 arg2}$$
  
  Or use the reverse connect mechanism

- On the login node:
  
  $$\text{ddt &}$$

- or using the remote client:
  
  Which needs to edit the job script to run the following command and submit:

  $$\text{ddt --connect mpirun -n 8 ./myapp.exe arg1 arg2}$$
Thank you!
support@bsc.es