

Basics on debugging parallel codes

Luca Tornatore - I.N.A.F. 

“Foundation of HPC” course



DATA SCIENCE &
SCIENTIFIC COMPUTING
2022-2023 @ Università di Trieste



Debugging in parallel

The problem is much more complex: the fundamental additional challenge is the simultaneous execution.

Shared memory paradigm: OpenMP, pthreads, ...

- Multiple threads running
- Shared vs private memory regions
- Race conditions

Message-passing paradigm: MPI

- Multiple independent processes (+ possible multithread)
- Communication
- Deadlocks



Let's have a try:

```
:~$ gcc -g -o my_threadprog my_threadprog.c -lpthread
```

```
:~$ gdb ./my_threadprog
```



Multi-threads capability of **gdb**

```
~$ gdb ./my_threadprog
Reading symbols from my_threadprog...done.

(gdb) r

(gdb) Starting program: my_threadprog

[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Creating thread 0
[New Thread 0x7ffff77ef700 (LWP 24144)]
Thread #0 says: " Hello World! "
Creating thread 1
[New Thread 0x7ffff6fee700 (LWP 24145)]
Thread #1 says: " Hello World! "
Creating thread 2
[New Thread 0x7ffff7fff700 (LWP 24146)]
Thread #2 says: " Hello World! "
Creating thread 3
[New Thread 0x7ffff65d6700 (LWP 24147)]
Thread #3 says: " Hello World! "
[Thread 0x7ffff7fff700 (LWP 24146) exited]
[Thread 0x7ffff6fee700 (LWP 24145) exited]
[Thread 0x7ffff77ef700 (LWP 24144) exited]
[Thread 0x7ffff7fac700 (LWP 24140) exited]
[Inferior 1 (process 24140) exited normally]
(gdb) Quit
(gdb)
```



It is necessary to explicitly set up **gdb** for multi-thread debugging

```
(gdb) set pagination off
```

```
(gdb) set scheduler-locking on
```

```
(gdb) set non-stop [on|off]
```



In **all-stop** mode, whenever the execution stops, *all* the threads stop (wherever they are).

Whenever you restart the execution, *all* the threads re-start:

however, **gdb** can not single-step all the threads in the steplock.

Some threads may execute several instructions even if you single-stepped the thread under focus with *step* or *next* commands.

non-stop mode means that when you stop a thread, all the other ones continue running until they finish or they reach some breakpoint that you pre-defined



Multi-threads capability of **gdb**

Change focus to *thread_no*

Shows info on active threads.
* tags the active thread

(gdb) thread *thread_no*

(gdb) info threads

(gdb) thread apply [*thread_no*] [*all*] *args*

(gdb) break <...> thread *thread_no*

Insert a break into a list
of threads

Apply a command to a list
of threads



| Some hint on the workflow

- ▶ If possible, write a code natively parallel but able to run in serial, which means with 1 MPI task or 1 thread
- ▶ Profile, debug and optimize that code in serial first
- ▶ If multi-threaded, test & debug thread sync / races with 1 MPI task



| Some hint on the workflow

- ▶ Deal with communications, synchronization and race/deadlock conditions on a *small* number of MPI tasks
- ▶ Profile, debug and optimize communications on a *small* number of MPI tasks
- ▶ Finally, try the full-size run
unfortunately, some times bugs or improper design issues arise only with large number of processes or threads



It is still possible to use **gdb** directly, called from mpirun:

```
:~$ mpirun -np <NP> -e gdb ./program
```

(However, depending on your system that may not work properly)



| gdb and MPI

The simplest way to use **gdb** with a parallel program is :

```
:~$ mpirun -np <NP> xterm -hold -e gdb ./program
```

Which launches <NP> xterm windows with running gdb processes in which you can run each parallel process

```
(gdb) run <arg_1> <arg_2> ... <arg_n>
```

or

```
... xterm -hold -e gdb -args ./program <arg_1> ...
```



| gdb and MPI

```
:~$ mpirun -np <NP> xterm -hold -e gdb --args ./program <arg_1> ... <arg_2>
```

On a HPC facility, normally you do that while running an *interactive session*..
..and in several occasion *this* will not work, because HPC environments are hostile to X for several reasons (remember to connect with `-X` or `-Y` switch of ssh).



| gdb and MPI

Another not so handy possibility is to open as many connections as processes on different terminals on your local machine, and *attach* **`gdb`** to the already running MPI processes

```
:~$ mpirun -np <NP> ./program
```

Followed by:

```
:~$ gdb -p <PID_of_MPI_task_n>
```

For each MPI task you want to follow.



There are still 2 issues

1. *Where* to run **gdb**, if **xterm** is not available and you do not want to use it in multi-thread mode ?

You may consider using **screen** (practical example in few minutes)

2. *How* the MPI tasks should be convinced to wait for **gdb** to step in ?

→ *Next slides*



Note

A possible issue for attaching **gdb** to a running process is that you may not have the capability to do that on a Linux system.

Look in the file:
`/proc/sys/kernel/yama/ptrace_scope`

```
0 ("classic ptrace permissions")
    No additional restrictions on operations that perform
    PTRACE_MODE_ATTACH checks (beyond those imposed by the
    commoncap and other LSMs).

    The use of PTRACE_TRACEME is unchanged.

1 ("restricted ptrace") [default value]
    When performing an operation that requires a
    PTRACE_MODE_ATTACH check, the calling process must either have
    the CAP_SYS_PTRACE capability in the user namespace of the
    target process or it must have a predefined relationship with
    the target process. By default, the predefined relationship
    is that the target process must be a descendant of the caller.

    A target process can employ the prctl(2) PR_SET_PTRACER
    operation to declare an additional PID that is allowed to
    perform PTRACE_MODE_ATTACH operations on the target. See the
    kernel source file Documentation/admin-guide/LSM/Yama.rst (or
    Documentation/security/Yama.txt before Linux 4.13) for further
    details.

    The use of PTRACE_TRACEME is unchanged.

2 ("admin-only attach")
    Only processes with the CAP_SYS_PTRACE capability in the user
    namespace of the target process may perform PTRACE_MODE_ATTACH
    operations or trace children that employ PTRACE_TRACEME.

3 ("no attach")
    No process may perform PTRACE_MODE_ATTACH operations or trace
    children that employ PTRACE_TRACEME.
```




Note

Solutions:

1. Get the capability
2. As root type:
`echo 0 > /proc/sys/kernel/yama/ptrace_scope`
3. Set the `kernel.yama.ptrace_scope` variable in the file `/etc/sysctl.d/10-ptrace.conf` to 0

The last solution turns off the security measure permanently, it is not a good idea (at least on a facility)

```
0 ("classic ptrace permissions")
    No additional restrictions on operations that perform
    PTRACE_MODE_ATTACH checks (beyond those imposed by the
    commoncap and other LSMs).

    The use of PTRACE_TRACEME is unchanged.

1 ("restricted ptrace") [default value]
    When performing an operation that requires a
    PTRACE_MODE_ATTACH check, the calling process must either have
    the CAP_SYS_PTRACE capability in the user namespace of the
    target process or it must have a predefined relationship with
    the target process. By default, the predefined relationship
    is that the target process must be a descendant of the caller.

    A target process can employ the prctl(2) PR_SET_PTRACER
    operation to declare an additional PID that is allowed to
    perform PTRACE_MODE_ATTACH operations on the target. See the
    kernel source file Documentation/admin-guide/LSM/Yama.rst (or
    Documentation/security/Yama.txt before Linux 4.13) for further
    details.

    The use of PTRACE_TRACEME is unchanged.

2 ("admin-only attach")
    Only processes with the CAP_SYS_PTRACE capability in the user
    namespace of the target process may perform PTRACE_MODE_ATTACH
    operations or trace children that employ PTRACE_TRACEME.

3 ("no attach")
    No process may perform PTRACE_MODE_ATTACH operations or trace
    children that employ PTRACE_TRACEME.
```



| **gdb** and MPI

We are left with the problem of *attaching* the **gdb** to a running process (or several running processes).

There is a classical trick, that requires to insert some small additional code in your program

```
int wait = 1
```

```
while(wait)  
    sleep(1);
```

The MPI processes will wait indefinitely until the value of wait does not change..
*which you can do from inside **gdb** attached to each process.*



| gdb and MPI

Let's say that your MPI program starts with:

```
int main(int argc, char **argv)
{
    int Me, Size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &Me);
    MPI_Comm_size(MPI_COMM_WORLD, &Size);

    ...
}
```

and that you insert the following code snippets right after it →



gdb and MPI

compiled only if DEBUGGER is defined

```
#ifdef DEBUGGER
    int wait = 1;
    pid_t my_pid;
    char my_host_name[200];

    gethostname(my_host_name, 200);
    my_pid = getpid();
```

```
    for(int i = 0; i < Size; i++)
    {
        if(i == Me)
            printf("task with PID %d on host %s is waiting\n",
                  my_pid, my_host_name);
        MPI_Barrier(MPI_COMM_WORLD);
    }
```

```
    while ( wait )
        sleep(1);
```

} wait is 1, so each process is just spinning

```
    MPI_Barrier(MPI_COMM_WORLD);
#endif
```

} Once MPI procs exit the previous while they are not rushing away

Each process print the
message, forced to follow
rank-order



| gdb and MPI

If you do not want to recompile with the “DEBUGGER” compile.time option:

```
char *env_ptr;  
if( ( env_ptr = getenv("DEBUG_THIS")) != NULL) &&  
    ( strncasecmp(env_ptr, "YES", 3) == 0 )  
{  
    int    wait = 1;  
    pid_t my_pid;  
    char   my_host_name[200];  
  
    gethostname(my_host_name, 200);  
    my_pid = getpid();  
  
    < ... >  
  
    while ( wait )  
        sleep(1);  
  
    MPI_Barrier(MPI_COMM_WORLD);  
}
```

Now this code is always there.

It becomes active only when you define the environment variable “DEBUG_THIS” as being “YES”, which you can do at the shell prompt right before calling mpirun.



| gdb and MPI

Even more flexibility:

```
char *env_ptr;
if( (env_ptr = getenv("DEBUG_THIS")) != NULL) &&
    ( strncasecmp(env_ptr, "YES", 3) == 0) )
{
    int    get_through = Me;
    pid_t my_pid;
    char   my_host_name[200];

    gethostname(my_host_name, 200);
    my_pid = getpid();

    < ... >

    while ( !get_through )
        sleep(1);

    MPI_Barrier(MPI_COMM_WORLD);
}
```

get_through is set to the MPI rank of the process, i.e. it is > 0 for all the procs but for the 0th one.

In other words, all the MPI tasks but the 0th will wait at the subsequent barrier.

This way, you can avoid to manually change **get_through** for *all* the tasks and unlock only the 0th



| gdb and MPI



A handy alternative to is to use **screen** on a single term.
screen is a utility you may have to install by yourself (or ask the sys admin to do that):
check your preferred packaging system or whatever you use to install apps (on HPC facilities you shall compile and install in your home, however).
Basic commands:

- | | | |
|---------------------|----------------|--------------------------------|
| <code>screen</code> | | start the screen |
| <code>ctrl+a</code> | <code>?</code> | get help on commands |
| | <code>"</code> | get the list of active windows |
| | <code>c</code> | create a new window |
| | <code>A</code> | rename the current window |



Debugging outline



GUI
for GDB



1. GDB text-user-interface
2. GDB DASHBOARD
3. GDBGUI
4. EMACS
5. DDD
6. NEMIVER, ECLIPSE, NETBEANS, CODEBLOCKS,
many others..



You can start gdb with a text-user-interface:

```
%> gdb -tui
```

Or you can activate/deactivate it from gdb itself:

```
(gdb) ctrl-x a
```

```
(gdb) ctrl-x o
```

```
(gdb) ctrl-x 2
```

```
(gdb) layout src
```

```
asm
```

```
split
```

```
regs
```

change focus

shows assembly windows

display src and commands

display assembly and commands

display src, asm and commands

display registers window



GDB built-in tui



```
gdb_try_breaks.c
372 {
373
B+> 374 if ( argc > 1)
375     // arg 0 is the name of the program itself
376     {
377         printf( "\nexploring my %d argument%c:\n", argc-1, (argc>2)?'s':' ' );
378         for ( int i = 1; i < argc; i++ )
379             {
380                 printf( "\targument %d is : %s\n", i, *(argv+i) );
381             }
382         printf( "\n" );
383     }
384
385     else
386
387         printf ( "no arguments were given, using default: %d\n\n", DEFAULT_ARG1 );
388
389
390     int arg1;
391
392     if ( argc > 1 )
393         arg1 = atoi( *(argv+1) );
394
395     else
396         arg1 = DEFAULT_ARG1;
397
398     int ret;
399
400     ret = function 1( arg1 );

native process 8943 In: main
(gdb) l
360 in /home/luca/code/tricks/gdb_try_breaks.c
(gdb) break main
Breakpoint 1 at 0xd38: file gdb_try_breaks.c, line 374.
(gdb) r
Starting program: /home/luca/code/tricks/gdb_try_breaks

Breakpoint 1, main (argc=1, argv=0x7fffffffdaa8) at gdb_try_breaks.c:374
(gdb) □
```



GDB built-in tui



```

gdb_try_breaks.c
B+> 374 if ( argc > 1)
375     // arg 0 is the name of the program itself
376     {
377         printf( "\nexploring my %d argument%c:\n", argc-1, (argc>2)?'s':' ' );
378         for ( int i = 1; i < argc; i++ )
379             {
380                 printf( "\targument %d is : %s\n", i, *(argv+i) );
381             }
382         printf( "\n" );
383     }
384
385     else
386
387         printf ( "no arguments were given, using default: %d\n\n", DEFAULT_ARG1 );

B+> 0x55555554d38 <main+15>      cmpl    $0x1,-0x14(%rbp)
0x55555554d3c <main+19>      jle     0x55555554db7 <main+142>
0x55555554d3e <main+21>      cmpl    $0x2,-0x14(%rbp)
0x55555554d42 <main+25>      jle     0x55555554d4b <main+34>
0x55555554d44 <main+27>      mov     $0x73,%edx
0x55555554d49 <main+32>      jmp     0x55555554d50 <main+39>
0x55555554d4b <main+34>      mov     $0x20,%edx
0x55555554d50 <main+39>      mov     -0x14(%rbp),%eax
0x55555554d53 <main+42>      sub     $0x1,%eax
0x55555554d56 <main+45>      mov     %eax,%esi
0x55555554d58 <main+47>      lea     0x2bf(%rip),%rdi      # 0x5555555501e
0x55555554d5f <main+54>      mov     $0x0,%eax
0x55555554d64 <main+59>      callq   0x55555554d68 <printf@plt>
0x55555554d69 <main+64>      movl    $0x1,-0xc(%rbp)

native process 9102 In: main                                     L374  PC: 0x55555554d38
(gdb) break main
Breakpoint 1 at 0xd38: file gdb_try_breaks.c, line 374.
(gdb) r
Starting program: /home/luca/code/tricks/gdb_try_breaks

Breakpoint 1, main (argc=1, argv=0x7fffffffdaa8) at gdb_try_breaks.c:374
(gdb) layout split

```



GUI for
GDB

GDB dashboard



<https://github.com/cyrus-and/gdb-dashboard>

```
dash
Source
3 void fun(int n, char *data[])
4 {
5     int i;
6     for (i = 0; i < n; i++) {
7         printf("%d: %s\n", i, data[i]);
8     }
9 }
10 }
11
12 int main(int argc, char *argv[])
13 {
14     // Assembly
15     0x0000000010000f18 b0 00 fun+56 mov $0x0,%al
16     0x0000000010000f1a e8 4b 00 00 fun+58 callq 0x100000f6a
17     0x0000000010000f1f 89 45 e8 fun+63 mov %eax,-0x18(%rbp)
18     0x0000000010000f22 8b 45 ec fun+66 mov -0x14(%rbp),%eax
19     0x0000000010000f25 85 01 00 00 fun+69 add $0x1,%eax
20     0x0000000010000f2a 89 45 ec fun+74 mov %eax,-0x14(%rbp)
21     0x0000000010000f2d e9 c4 ff ff fun+77 jmpq 0x100000ef6 <fun+22>
22     0x0000000010000f32 48 83 c4 20 fun+82 add $0x20,%rsp
23     0x0000000010000f36 5d fun+86 pop %rbp
24     0x0000000010000f37 c3 fun+87 retq
25
26     // Threads
27     [1] id 4355 from 0x0000000010000f2a in fun+74 at scrot.c:7
28
29     // Stack
30     [0] from 0x0000000010000f2a in fun+74 at scrot.c:7
31     arg n = 3
32     arg data = 0x7fff5bffb60
33     loc i = 1
34     [1] from 0x0000000010000f62 in main+34 at scrot.c:14
35     arg argc = 3
36     arg argv = 0x7fff5bffb60
37     (no locals)
38
39     // Registers
40     rax 0x0000000000000002 rbx 0x0000000000000000 rcx 0x0000010000000203
41     rdx 0x0000020000000200 rsi 0x0000000000012068 rdi 0x00007fff79e86118
42     rbp 0x00007fff5bffb620 rsp 0x00007fff5bffb600 r8 0x0000000000000040
43     r9 0x00007fff79e86110 r10 0xffffffffffffffff r11 0x0000000000000246
44     r12 0x0000000000000000 r13 0x0000000000000000 r14 0x0000000000000000
45     r15 0x0000000000000000 rip 0x0000000100000f2a eflags [ TF IF ]
46     cs 0x0000002b ss <unavailable> ds <unavailable>
47     es <unavailable> fs 0x00000000 gs 0x00000000
48
49     // Expressions
50     [1] data[1] = 0x7fff5bffb60 "hello"
51
52     // Memory
53     0x00007fff5bffb60c 01 00 00 00 60 fb bf 5f ff 7f 00 00 00 00 00 00 .....b....
54     0x00007fff5bffb610 03 00 00 00 40 fb bf 5f ff 7f 00 00 02 0f 00 00 .....b....
55     0x00007fff5bffb61c 01 00 00 00 60 fb bf 5f ff 7f 00 00 03 00 00 00 .....b....
56     0x00007fff5bffb620 00 00 00 00 50 fb bf 5f ff 7f 00 00 ad 15 f7 9c .....P.....
57
58     0x00007fff5bffb6f0 68 65 6c 6c 6f 00 47 44 42 00 54 45 52 4d 5f 50 hello.GDB.TERM_P
59
60     // History
61     $$1 = 63
62     $$0 = {[0] = 0x7fff5bffb60 "hello", [1] = 0x7fff5bffb60 "GDB"}
```




GUI for
GDB

GDBgui



[gdbgui](#) [Download](#) [Docs](#) [Examples](#) [Screenshots](#) [Videos](#) [About](#) [Chat](#) [GitHub](#)

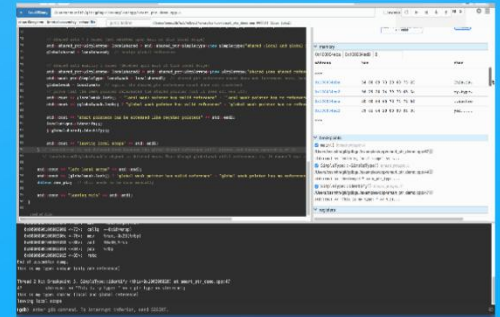
<https://gdbgui.com/>



Browser-based debugger for C, C++, go, rust, and more
gdbgui turns this

```
>>> gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type 'help'.
Type 'apropos word' to search for commands related to 'word'.
>>>
```

into this



[Download Now »](#)

[Download Now »](#)



that's all, have fun



“So long
and thanks
for all the fish”

The image features a 3D rendered scene. At the top, there is a horizontal band of dark, textured wood. Below this, the background is a solid, medium-blue color. In the center, the phrase "So long and thanks for all the fish" is rendered in large, white, 3D block letters. The letters are arranged in three lines: "So long" on the top line, "and thanks" on the middle line, and "for all the fish" on the bottom line. The letters have a slight shadow on the blue surface below them. On the left side of the first line, there is a small white 3D quotation mark, and on the right side of the second line, there is a small white 3D closing quotation mark.