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

Multi-threads capability of **gdb**

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 0x7fffefffff700 (LWP 24146)]
Thread #2 says: " Hello World! "
Creating thread 3
[New Thread 0x7ffff65d6700 (LWP 24147)]
Thread #3 says: " Hello World! "
[Thread 0x7fffefffff700 (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)
```

Multi-threads capability of **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]
```

Multi-threads capability of **gdb**

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

gdb and MPI

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.

gdb and MPI

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 **attacching 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 you MPI program start 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);

  MPI_Barrier(MPI_COMM_WORLD);
#endif
```

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

wait is 1, so each process is just spinning

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

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 is set to the MPI rank of the process, i.e. it is > 0 for all the procs but for the 0 one.
In other words, all the MPI tasks but the 0^{th} will wait at the subsequent barrier.
This way, you can avoid to manually change **get_through** for *all* the tasks and unlock only the 0^{th}

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:

screen		start the screen
Ctrl+a	?	get help on commands
	"	get the list of active windows
	c	create a new window
	A	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..

GDB built-in **tui**

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

```
%> gdb -tui
```

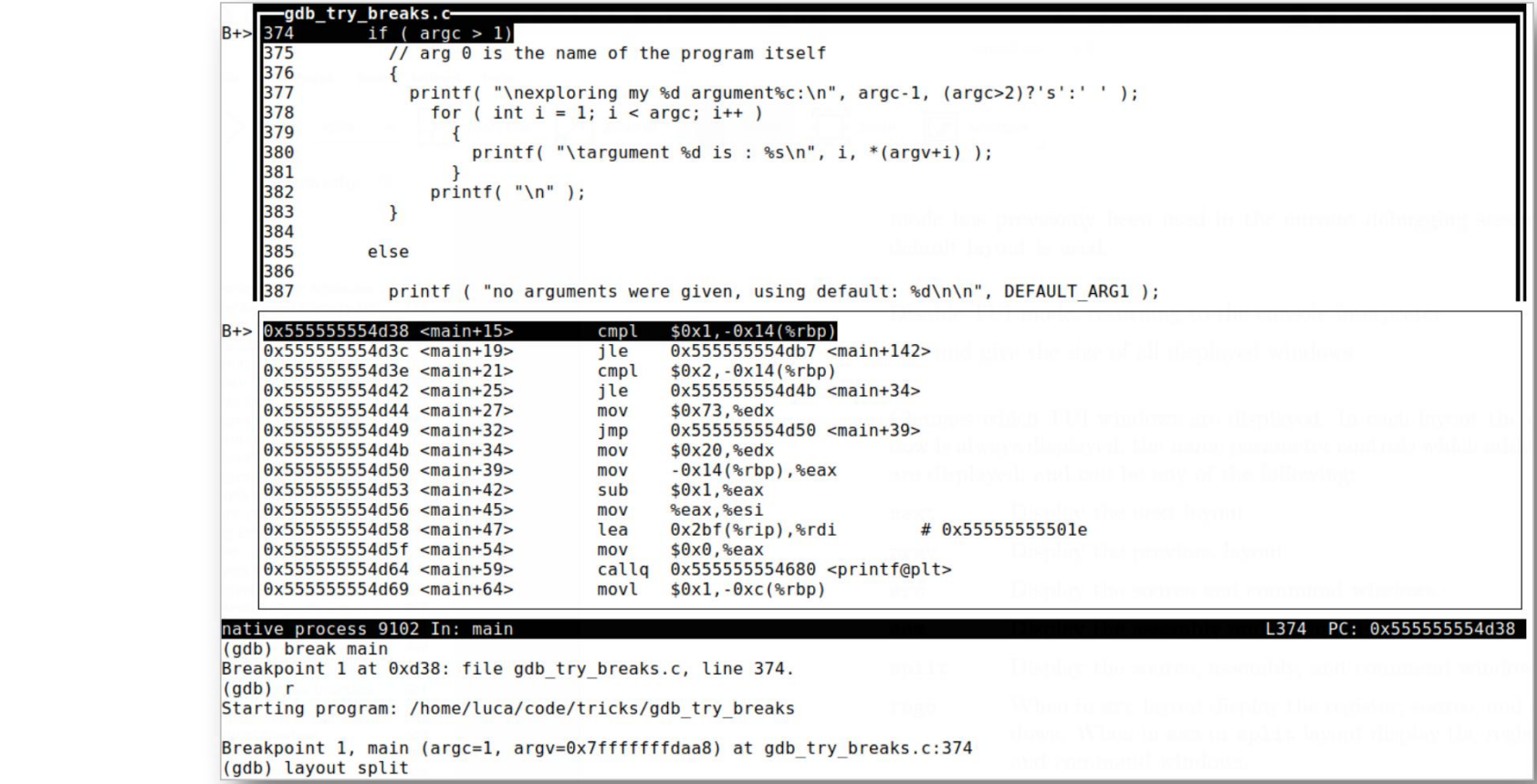
Or you can activate/deactivate it from gdb itself:

`(gdb)ctrl-x a`	
`(gdb)ctrl-x o`	change focus
`(gdb)ctrl-x 2`	shows assembly windows
`(gdb)layout src`	display src and commands
`asm`	display assembly and commands
`split`	display src, asm and commands
`regs`	display registers window

GDB built-in **tui**

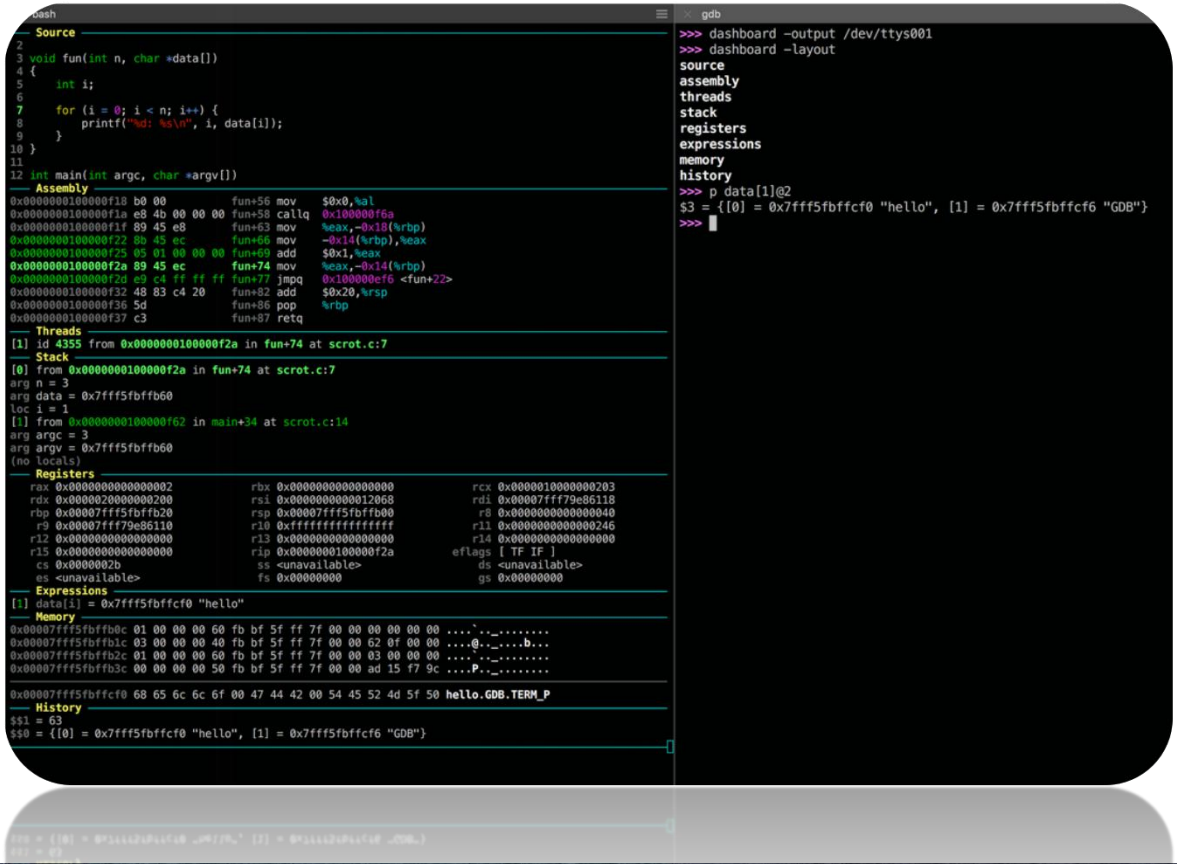
```
gdb_try_breaks.c
372     {
373
B+>374        if ( argc > 1)
375          // arg 0 is the name of the program itself
376          {
377            printf( "\nexploring my %d argument%c:\n", argc-1, (argc>2)?'s':' ' );
378              for ( int i = 1; i < argc; i++ )
379                {
380                  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                                    L374  PC: 0x555555554d38
(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**

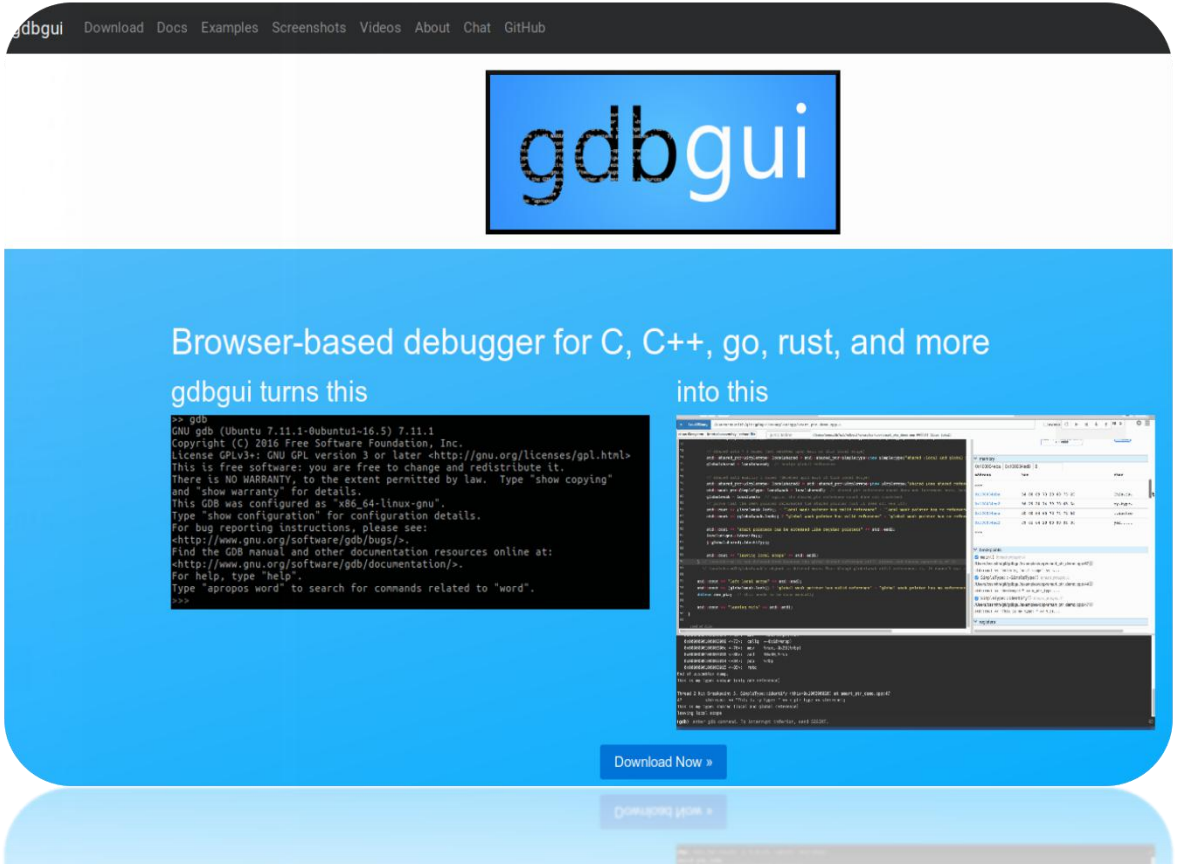


GDB dashboard

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



https://gdbgui.com/



that's all, have fun