

Debugging & Profiling

Luca Tornatore - I.N.A.F.



“Foundation of HPC” course



DATA SCIENCE &
SCIENTIFIC COMPUTING
2021-2022 @ Università di Trieste

Outline



Profiling



Debugging



Profiling outline



Mapping



Profiling
Tools



Profiling
Behaviour



Profiling
Efficiency



Map the workflow

A call tree (more precisely, a **call graph**) is a control flow graph that exhibits the calling relationships among routines in a program.

A *node* in the graph represent a routine, while an *edge* represents a calling relationship.

We'll concentrate on *dynamic call graphs* – i.e. the records of program executions.

The most complete graph is context-sensitive, which means that every call stack of a procedure is recorded as a separate node (the resulting graph is called *calling context tree* instead of *call tree*).

However, that requires a larger amount of memory for large program, it is useful in case of code reuse (the same code being executed at different points by different call paths).



INSTRUMENTATION

Inserts extra code at compile time wrapping function calls to count how many times it calls / is called and how much time it takes to execute.

SAMPLING

The profiler ask for interrupts $N_{samples}$ per sec + interrupts at function calls + interrupts at selected events, and records on a histogram the number of occurrences in every part of the program. The call graph is inferred from these data.

DEBUGGING

The profiler ask for interrupts at every line code and function call (more correct: enters the by-step execution mode)



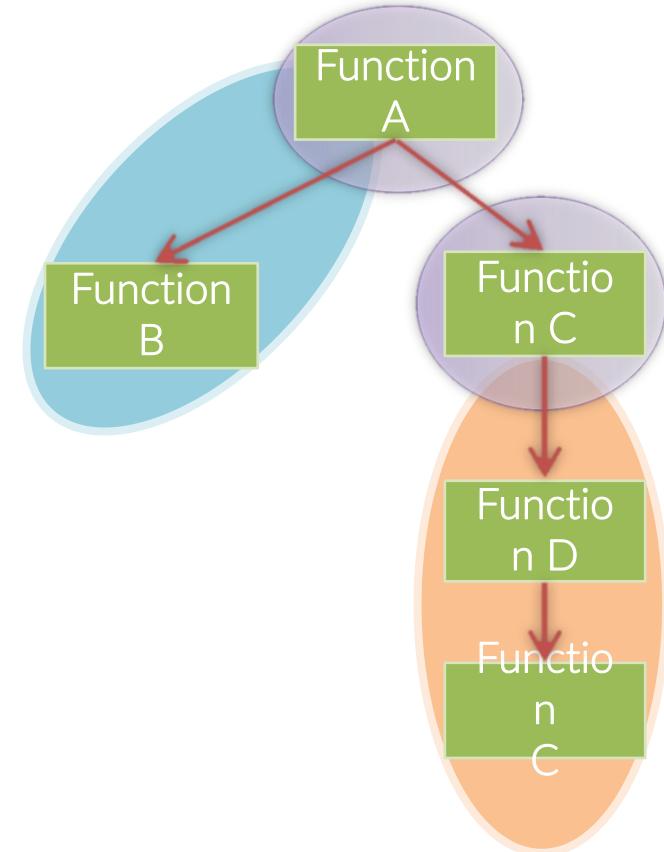
Basic profiling concepts

Collect program events

- Hardware interrupts
- Code instrumentation
- Instruction set simulation
- tracing

Periodic sampling

- Top of the stack (exclusive)
- All stack (children inclusive)





| How to obtain the call tree

There are several way to obtain a dynamic call tree.

The main open source alternatives are:

1. Using `gcc` and `gprof`
2. Using linux `google perftools`
3. Using `valgrind`
4. Using `perf`
5. Some others, mostly non-free (among significant free: `CodeXL` by AMD)

Many IDE uses the aforementioned tools or their own plug-ins (`eclipse`, `netbeans`, `code::block`, `codelite`, ...)



Using **gcc + gprof**

Just compile your source using **-pg** option.

You should also profile turning on the optimizations you're interested in.

```
gcc -[my optimizations] -pg -o myprogram.x myprogram.c
```

*Note: don't use the option **-p**. It provides less information than **-pg**.*

After that, run your program normally. Profiling infos will be written in the file **gmon.out**.

- ▶ You can read (options and details in the man page) the informations by
gprof myprogram.x

- ▶ You can visualize the call graph by (read the man pages...)
gprof myprogram.x | gprof2dot.py | dot -T png -o callgraph.png



| The call-tree



Well, now that you just met, say goodbye to the glorious **gprof**

He's a dinosaur from the past decades..

- lacks real multithread support
- lacks real line-by-line capability
- does not profile shared libraries
- need recompilation
- may lie easier than other tools (see later..)

You may still consider it for some call counts business

...you may still consider it for some call counts business

- may lie easier than other tools (see later..)
- need recompilation



How to induce your profiler to lie

gprof (gcc -pg)
does not record
the call stack

```
#include <stdlib.h>
void loop(int n)
{
    int volatile i; // does not optimize out
    i = 0;
    while(i++ < n);
}

void light(int n) { loop(n); } void heavy(int n)
{ loop(n); }

int main(void)
{
    light(100000);
    heavy(100000000);
    return 0;
}
```



Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
101.30	0.20	0.20	2	101.30	101.30	loop
0.00	0.20	0.00	1	0.00	101.30	heavy
0.00	0.20	0.00	1	0.00	101.30	light



How could it happen?
The man page is clear:

We assume that the time for each execution of a function can be expressed by the total time for the function divided by the number of times the function is called.

Thus the time propagated along the call graph arcs to the function's parents is directly proportional to the number of times that arc is traversed.



How to induce your profiler to lie

VALGRIND seems to better understand the situation, because it doesn't record only the number of calls to a function but also the time spent in a function given a call path (which, at odds, gprof infers backwardly).

That's why the valgrind measure of time spent in a function plus its callees is reliable.

However, it may as well end up to a misleading picture if we stick in and additional layer of complexity, unless you explicit tells it to track separately different call stacks with the command-line option -

-separate-callers = N

Type	Callers	All Callers	Callee Map	Source Code
#	Ir	Source		
0		---		From '/home/luca/code/HPC_LECTURES/'
14				void heavy(int n) { loop(n); }
15				
16		int main(void)		
17	0.00 {			
18	0.00 light(100000);			
19	0.10 1 call(s) to 'light' (lie1.g: lie1.c)			
19	0.00 heavy(100000000);			
19	99.88 1 call(s) to 'heavy' (lie1.g: lie1.c)			
20	0.00 return 0;			
21	0.00 }			
22				



Using google perftools

That is the CPU profiler used at Google's. It provides a **thread-caching malloc**, a **heap-checker**, **heap-profiler**, **CPU profiler**.

As for the latter, basically there are 3 phases:

1. linking the library to the executable
2. running the executable
3. analyzing results

[1] LINKING

There are two options:

- Link `-lprofiler` at the executable
- Adding the profiler at run time `LD_PRELOAD="/usr/lib/libprofiler.so"`
`/path/to/exec`

This does not start the CPU profiling, though.



Using google perftools

[2] RUNNING

- Define

```
env CPUPROFILE= exec.prof /path/to/exec
```

You may define a signal, too

```
env CPUPROFILE= exec.prof /path/to/exec \
CPUPROFILESIGNAL= XX exec.prof &
```

so that to be able to trigger the start and stop of the profiling by

```
killall -XX exec
```

- In the code, include `<gperftools/profiler.h>` and encompass the code segment to be profiled within

```
ProfilerStart("name_of_profile_file")
...
ProfilerStop()
```

`CPUPROFILE_FREQUENCY=x` modifies the sampling frequency



Using google perftools

[3] ANALYZING RESULTS

```
pprof exec exec.prof
```

“interactive mode”

```
pprof --text exec exec.prof
```

output one line per procedure

```
pprof --gv exec exec.prof
```

annotated call graph via ‘gv’

```
pprof --gv --focus = some_func ...
```

restrict to code paths including “*some_func”

```
pprof --list = some_func ...
```

per-line annotated list of some_func

```
pprof --disasm= some_func ...
```

annotated disassembly of some_func

```
pprof -callgrind exec exec.prof
```

output call infos in callgrind format



Using perf

[1] COMPILING

```
gcc -g -fno-omit-frame-pointer my_prog.c -o my_prog
```

[2] RUNNING

```
perf record -F ffff -call-graph <fp|lbr|dwarf> \
my_prog.c <args>
```

[3] ANALYZING

```
perf report --call-graph=graph < --stdio >
```



Using valgrind

[1] COMPILING

```
gcc -g -fno-omit-frame-pointer my_prog.c -o my_prog
```

[2] RUNNING

```
Valgrind --tool=callgrind --callgrind-out-file= $CALLGRIND_OUT --dump-instr=yes --coollect-jumps=yes --cache-sim=yes --branch-sim=yes < --I1=... >  
< --D1=...> my_prog <args>
```

[3] ANALYZING

```
kcacheGrind $CALLGRIND_OUT
```



- BASIC / SYSTEM tools
 - gprof / gdb / perf / gperf tools
 - Valgrind – cachegrind,
callgrind, ..
- HARDWARE COUNTER / PMU interface
 - perf
 - PAPI
 - Intel PMI
 - Likwid
 - ...
- HPC Tools
 - HPCToolkit
 - OpenSpeedShop
 - TAU
 - SCOREP +
- VENDOR tools
 - ARM-Allinea
 - CodeXL (AMD)
 - Intel tools
 - ...



| Tools : valgrind

An instrumentation framework for building dynamic analysis tools.

Valgrind basically runs your code in a virtual “sandbox” where a synthetic CPU (the same you have) is simulated and executes an instrumented code.

There are various Valgrind based tools for debugging and profiling purposes.

- **Memcheck** is a memory error detector correctness
- **Cachegrind** is a cache and branch-prediction profiler velocity
- **Callgrind** is a call-graph generating cache profiler. It has some overlap with Cachegrind
- **Helgrind** is a thread error detector correctness
- **DRD** is also a thread error detector.
Different analysis technique than Helgrind
- **Massif** is a heap profiler memory efficiency using less memory
- **DHAT** is a different kind of heap profiler memory layout inefficiencies
- **SGcheck** (experimental tool) that can detect overruns of stack and global arrays

KCacheGrind is a very useful GUI



Memcheck : highlighting memory errors

- Invalid memory access: overrunning/underrunning of heap blocks or top of stack, addressing freed blocks, ...
- Use of variables with undefined values
- Incorrect freeing of heap memory
- Errors in moving memory (unwanted src/dst overlaps, ...)
- Memory leaks

Cachegrind: simulating the cache

It can report how many hits (L1, L2 and L3, I- and D-) and how many misses.

It can analyze CPU's branch prediction.

I_r, I1mr, LLmr, Dr, D1mr, DLmr, Bc, Bcm, Bi, Bim, ...

Callgrind: profiling the CPU

It collects the number of instructions executed, links them to source lines, records the caller/callee relationship between functions, and the numbers of such calls. It can collect data on cache simulation and/or branches.

Outline



Debugging



Debugging outline



```
if(q > 0)
{
    switch(q)
    {
        case(1): // row y = 0 and/or plane symmetry
            if(subregions[i][BOTTOM][...y_] == 0)
                // this subregion in quadrant 1 contains the
                // set-up corners for seed sub-region generation
                SBL[_y_] = 0, SBL[_x_] = Nmesh - subregions
                STR[_y_] = 1, STR[_x_] = Nmesh - subregions
                // find horizontal extension in this quadrant
                Np = STR[_x_] - SBL[_x_];
                // allocate memory for seeds in this string
                SEED_0 = (unsigned int*)malloc(sizeof(unsigned
                if(!internal.nmic_original_seedtable)
                    {
```



Introduction
Debugging
a code

Inspecting
a code crash

Debugging
running
process(es)



Introducing debugging



Smithsonian



It has long been recognized and documented that insects are the most diverse group of organisms, meaning that the numbers of species of insects are more than any other group. In the world, some 900 thousand different kinds of living insects are known. This representation approximates 80 percent of the world's species. The true figure of living species of insects can only be estimated from present and past studies. Most authorities agree that there are more insect species that have not been described (named by science) than there are insect species that have been previously named. Conservative estimates suggest that this figure is 2 million, but estimates extend to 30 million. In the last decade, much attention has been given to the entomofauna that exists in the canopies of tropical forests of the world. From studies conducted by Terry Erwin of the Smithsonian Institution's Department of Entomology in Latin American forest canopies, the number of living species of insects has been estimated to be 30 million. Insects also probably have the largest biomass of the terrestrial animals. At any time, it is estimated that there are some 10 quintillion (10,000,000,000,000,000,000) individual insects alive.



Introducing debugging



9/9	0.800	1.000	Anutan started	stopped ✓	1.2700 { 9.037 897 021 13° UC (03) MP - MC 1.56217000 9.037 896 995 (03) PRO 2 2.130476415 4.6159250 const 2.130676315
	1.000		Relays 6-2 in 033 failed special speed test in relay	" 10.000 test.	
	1.100	1.525	Started Cosine Tape (Sine check)	Relays changed	
	1.545		Started Multi Adder Test.		
	1.600		 Relay #70 Panel F (moth) in relay.		
	1.700		First actual case of bug being found. Anutan started.		
	1.700		closed down.		

The usage of *bug* and, hence, of *de-bugging* referred to programming is a long-standing tradition, whose origin is difficult to trace back.

An often-told story is about Mark-II calculator located at Harvard: On Sept. 9th, 1945, a technician found a moth in a relay that caused a flaw in a “program” execution.

Adm. Grace Hopper is reported to have written in its diary about that as “first actual case of bug being found”.

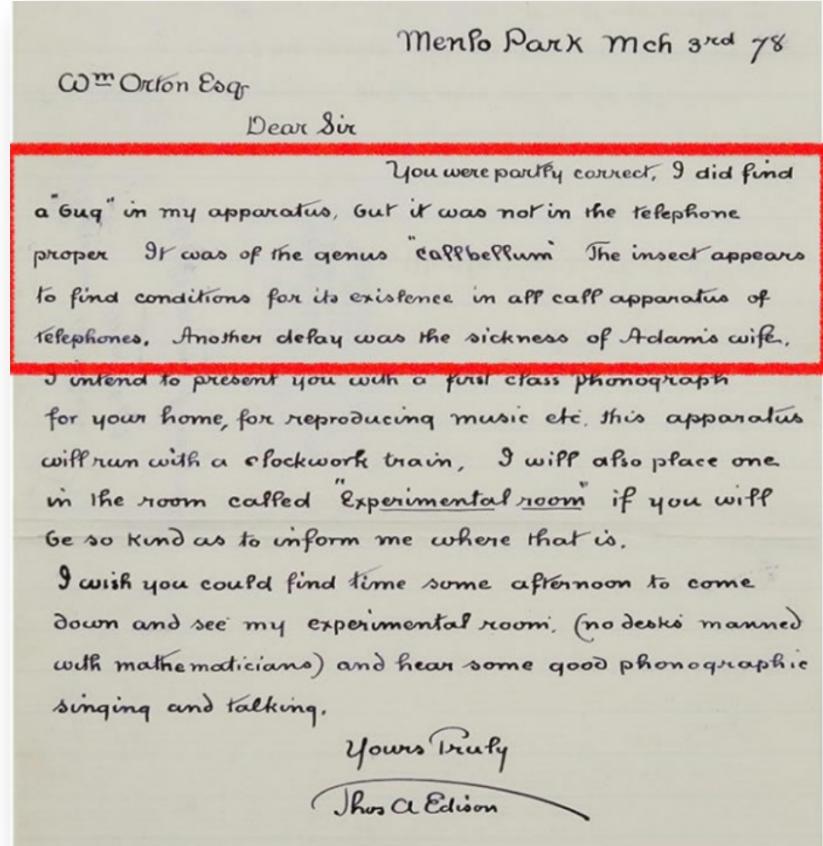


Introducing debugging

However, Thomas Edison found another *actual* bug in one of his phones, as he reports in a letter to an associate.

He later writes:

"It has been just so in all of my inventions. The **first step** is an intuition and comes with a burst, **then difficulties arise**—this thing gives out and [it is] then that "**bugs**"—as such little faults and difficulties are called—show themselves and **months of intense watching, study and labor are requisite** before commercial success or failure is certainly reached."





Introducing debugging



It is difficult to trace back how this term has been poured into computer programming jargon. However, already in the early 60s it was appearing in technical papers without need of explanation.

The immortal I. Asimov used it in a 1944 short robot story “Catch that rabbit”, and his enormous influence contributed much to make the term popular.

More funny infos:

[IEEE Annals of the History of Computing](#)
(Volume: 20 , [Issue: 4](#) , Oct-Dec 1998)

ask for the PDF if you're interested and do not have access

Stalking the Elusive Computer Bug

PEGGY ALDRICH KIDWELL

From at least the time of Thomas Edison, U.S. engineers have used the word “bug” to refer to flaws in the systems they developed. This short word conveniently covered a multitude of possible problems. It also suggested that difficulties were small and could be easily corrected. IBM engineers who installed the ASCC Mark I at Harvard University in 1944 taught the phrase to the staff there. Grace Murray Hopper used the word with particular enthusiasm in documents relating to her work. In 1947, when technicians building the Mark II computer at Harvard discovered a moth in one of the relays, they saved it as the first actual case of a bug being found. In the early 1950s, the terms “bug” and “debug,” as applied to computers and computer programs, began to appear not only in computer documentation but even in the popular press.

Introduction

S talking computer bugs—that is to say, finding errors in computer hardware and software—occupies and has occupied much of the time and ingenuity of the people who design, build, program, and use computers.¹ Early programmers realized this with some distress. Maurice Wilkes recalls that in about June of 1949:

I was trying to get working my first non-trivial program, which was one for the numerical integration of Airy's dif-

published in the *Annals* in 1981. Here the time is given as the summer of 1945, but the computer is the Mark II, not the Mark I. A photograph shows the moth taped in the logbook, labeled “first actual case of bug being found.” Small problems with computers have been called bugs ever since.³



Introducing debugging

That maybe is too dramatic and emphatic, but the point to get from Edison is that the de-bugging activity is an inherent and intrinsic one in software development

"It has been just so in all of my inventions. The **first step** is an intuition and comes with a burst, **then difficulties arise**—this thing gives out and [it is] then that "**bugs**"—as such little faults and difficulties are called—show themselves and **months of intense watching, study and labor are requisite** before commercial success or failure is certainly reached."

T. Edison



Provided that

- you'll have close encounters with bugs in your life;
- de-bugging is a fundamental and unavoidable part of your work;

what is the best way to proceed ?



1. Do not insert bugs

Highly encouraged, but rarely works



1. Do not insert bugs

Highly encouraged, but rarely works

2. Add `printf` statements everywhere

Highly discouraged, but sometimes it works.

In case of memory problems – typically due to pointers chaos – you may see that the problem “disappear”, or changes its appearance, when you insert a new `printf`



Introducing debugging



1. Do not insert bugs

Highly encouraged, but rarely works

2. Add `printf` statements everywhere

Highly discouraged, but sometimes works.

In case of memory problems – typically due to aos

3. Use a DEBUGGER

*That is definitely the best choice and, fortunately,
the subject of this lecture.*



Introducing debugging



gdb is almost certainly the best free, extremely feature-rich command-line debugger ubiquitously available on *nix systems.



There are 3 basic usages^(*) of GDB:

1. Debugging a code
quite mandatory to compile with -g
2. Inspecting a code crash through a core file
3. Debugging / inspecting a running code

^(*) Highly advised: learn keyboard commands. Although many gui exist (we'll see some later), keyboard is still the best productivity tool.



Debugging outline



Introduction

```
if(q > 0)
{
    switch(q)
    {
        case(1): // row y = 0 and/or plane symmetry
            if(subregions[i][BOTTOM] == 0)
                // this subregion in quadrant 1 contains the
                // set-up corners for seed sub-region generation
                SBL[_y_] = 0, SBL[_x_] = Nmesh - subregion
                STR[_y_] = 1, STR[_x_] = Nmesh - subregion
                // find horizontal extension in this quadrant
                Np = STR[_x_] - SBL[_x_];
                // allocate memory for seeds in this subregion
                SEED_0 = (unsigned int*)malloc(sizeof(unsigned
                if(internal.nmic_original_seedtable)
                    {
```



Debugging
a code

Inspecting
a code crash



Debugging
a running
process(es)



In order to include debugging information in your code, you need to compile it with `-g` family options (read the gcc manual for complete info):

-g

produce dbg info in O.S. native format

-ggdb

produce gdb specific extended info, as much as possible

-glevel

default level is 2. 0 amounts to no info, 1 is minimal, 3 includes extra information (for instance, macros expansion) – this allows macro expansion; add **-gdwarf-n** in case, where possible, where *n* is the maximum allowed (4)

-ggdblevel

you can combine the two to maximize the amount of useful info generated

remember: **-fno-omit-frame-pointer**, especially if you are using `-Ox`



Debugging
a code

Debugging: start



You just start your code under gdb control:

```
%> gdb program
```

You can define the arguments needed by your program already at invocation:

► live demo with `gdb_try_args.c`

```
%> gdb --args program arg1 arg2 ... argN
```

Or you can define the arguments from within the gdb session:

```
%> gdb program
Reading symbols from program...done.
(gdb) set args arg1 arg2 ... argN
(gdb) run
```



Debugging
a code

Debugging: run & stop



You may just want the code to run, for instance to reach the point of a seg fault:

```
%> (gdb) run
```

Or, you may want to stop it from the beginning to have full control of each step:

```
%> (gdb) break main  
%> (gdb) run
```

Or you may already know what is the problematic point to stop at:

```
%> (gdb) break location  
%> (gdb) run
```



Breakpoints are a key concept in debugging. They are stopping point at which the execution interrupts and the control is given back to you, so that you can inspect the memory contents (variables values, registers values, ...) or follow the subsequent execution step by step.

You can define a breakpoint in several way

(gdb) break

(gdb) break \pm
offset

(gdb) break
filename:linenum

(gdb) break
functionname

There are more
options, just
check the manual

insert a break at the current pos

insert a break *offset* lines after/before the current line

insert a break at *linenum* of file *filename*

insert a break at the entry point of function *functionname*



A breakpoint may be defined as dependent on a given condition:

```
(gdb) break my_function if (arg1 > 3 )
```

This sets a breakpoint at function *my_function* : the condition will be evaluated each time the point is reached, and the execution is stopped only if it is true.

Condition can be any valid expression.

```
(gdb) info break
```

gives you informations on active breakpoints.

```
(gdb) delete [n]
      clear [location]
<disable | enable> see the manual
```



You can define a list of commands to be executed when a given breakpoint is reached:

```
(gdb) break my_function if (arg1 > 3 )
Breakpoint 1 at 0x.....: file blabla.c, line 42
(gdb) command 1
Type commands for when breakpoint 1 is hit, one per
line.
End with a line saying just "end".
> print arg1
> print another_useful_variable
> x/10wd a_global_integer_array
```



Debugging: run & stop



When you have the control of the program execution, you can decide how to proceed:

(gdb) cont [c]

continue until the end / next stop

(gdb) cont *count-ignore*

continue ignoring the next *count-ignore* stops (for instance, a bp)

(gdb) next [n] | *count*
nexti

continue to the next src line *in the current stack frame*

(gdb) step [s] | *count*
stepi

continue to the next src line

(gdb) until [u] | *count*

continue until a src line past the current one is reached in the current stack fr.

(gdb) advance *location*

continue until the specified location is reached

► live demo with
[gdb_try_breaks.c](#)



Debugging: run & stop



You can also rewind your execution step-by-step

(gdb) reverse-continue [rc]

(gdb) reverse-step [*count*]
reverse-stepi

(gdb) reverse-next [*count*]
reverse-nexti

(gdb) set exec-direction <verse | forward >



Debugging: source list



Often, when you are debugging, you may have the need of looking at either the source lines or at the generated assembler:

(gdb) *list linenum*

print src lines around line
linenum of the current source file

(gdb) *list function*

print the source lines of *function*

(gdb) *list location*

print src lines around *location*

(gdb) *set listszie count*

control the number of src lines
printed

(gdb) *disass*
[*/m*] [*function*] [*location*]

show the assembler

► live demo with
[gdb_try_breaks.c](#)

Debugging: stack examination



Examining the stack is often of vital importance. With GDB you can have a quick and detailed inspection of all the stack frames.

backtrace [args] print the backtrace of the whole stack

n print only the *n* innermost frames

-n print only the *n* outermost frames

full print local variables value, also

where, info stack additional aliases

► live demo with
`gdb_try_breaks.c`

Debugging: memory examination



Accessing to the content of memory is a fundamental ability of a debugger. You have several different ways to do that:

(gdb) *print variable*
p/F *variable*

print the value of *variable*
print *variable* in a different format
(x, d, u, o, t, a, c, f, s)
see then manual for advanced location

► live demo with
`gdb_try_breaks.c`

(gdb) *x/FMT address*

Explore memory starting at address *address*
-> see at live demo how to use this

(gdb) *display expr*
display/fmt *expr*
display/fmt *addr*

Add *expr* to the list of expressions to display each time your program stops



Memory can be searched to find a particular value,
of a given size

► live demo with
`gdb_try_breaks.c`

(gdb) `find [/sn]
start, end, val1
,val2, ...]`

`find [/sn]
start, +len, val1
,val2, ...]`

Search memory for a
particular sequence of
bytes.

s is the size of type to be
searched

n is the max number of
occurrences



Examining registers may be also useful (although it's something that only quite advanced users can conceive)

► live demo with
`gdb_try_breaks.c`

(gdb) info registers	print the value of all registers
(gdb) info vector	print the content of vector registers
(gdb) print \$rsp	print value of the stack pointer
(gdb) x/10wd \$rsp	print values of the first 10 4-bytes integers on the stack
(gdb) x/10i \$rip	print the next 10 asm instructions



If there are macros in your code, they can be expanded, provided that you compiled the code with the appropriate option:

`-g3 [gdb 3] [-dwarf-4]`

(gdb) `macro expand macro`

shows the expansion of macro *macro*; expression can be any string of tokens

(gdb) `info macro [-a|-all]
macro`

shows the current (or all) definition(s) of *macro*

(gdb) `info macros location`

shows all macro definitions effective at *location*



Debugging: watch points



You can set *watchpoints* (aka “keep an eye on this and that”) instead of breakpoints, to stop the execution whenever a value of an expression / variable / memory region changes

(gdb) *watch variable*

keep an eye onto *variable*

(gdb) *watch expression*

stops when the value of expression changes (the scope of variables is respected)

(gdb) *watch -l expression*

Interpret *expression* as a memory location to be watched

(gdb) *watch -l expression*
[mask *maskvalue*]

a mask for memory watching: specifies what bits of an address should be ignored (to match more addresses)

(gdb) *rwatch [-l] expr*
[mask *mvalue*]

stops when the value of *expr* is read

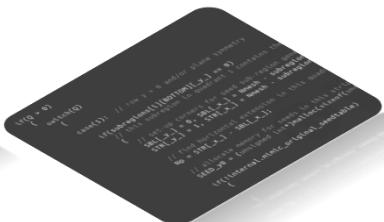
► live demo with `gdb_try_watch.c`



Debugging outline



Introduction



Debugging
a code



Inspecting
a code crash



Debugging
a running
process(es)



It happens that you have code crashes in conditions not easily reproducible when you debug the code itself, for a number of reasons.

However, the O.S. can dump the entire “program status” on a file, called the *core file*:

```
luca@GGG:~/code/tricks% ./gdb_try_watch
no arguments were given, using default: 100

something wrong at point 2
[1] 8435 segmentation fault (core dumped) ./gdb_try_watch
luca@GGG:~/code/tricks% ls -l core
-rw----- 1 luca luca 413696 nov  8 15:45 core
luca@GGG:~/code/tricks% 
```

In order to allow it to dump the core, you have to check / set the core file size limit:

```
%> ulimit -c [size limit in KB]
```



Once you have a core, you can inspect it with GDB

```
%> gdb executable_name core_file_name
```

Or

```
%> gdb executable_name
(gdb) core core_file_name
```

The first thing to do, normally, is to unwind the stack frame to understand where the program crashed:

```
(gdb) bt full
```



Debugging outline



Introduction



Debugging
a code



Inspecting
a code crash



Debugging
a running
process



Attach to a process

In order to debug a running process, you can simply attach gdb to it:

▶ live demo with
`gdb_try_attach.c`

```
%> gdb
(gdb) attach process-id
```

and start searching it to understand what is going on



Debugging outline



Debugging
parallel
processes



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 0x7fffff77ef700 (LWP 24144)]
Thread #0 says: " Hello World! "
Creating thread 1
[New Thread 0x7fffff6fee700 (LWP 24145)]
Thread #1 says: " Hello World! "
Creating thread 2
[New Thread 0x7fffffeffff700 (LWP 24146)]
Thread #2 says: " Hello World! "
Creating thread 3
[New Thread 0x7fffff65d6700 (LWP 24147)]
Thread #3 says: " Hello World!
[Thread 0x7fffffeffff700 (LWP 24146) exited]
[Thread 0x7fffff6fee700 (LWP 24145) exited]
[Thread 0x7fffff77ef700 (LWP 24144) exited]
[Thread 0x7fffff7fac700 (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 target-async on
(gdb) set non-stop on
```



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

live demo



Multi-threads capability of gdb

Change focus to *threas_no*

(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

Shows info on active threads.
* tags the active thread

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, deal with 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
- ▶ Get to 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> ...
```



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

Next slides

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



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.
```



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.



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.
```



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



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



A little bit more of flexibility:

```
char *env_ptr;
if ( (env_ptr = getenv("DEBUG_THIS")) != NULL) &&
    (strcasecmp(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.



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 0 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



A handy alternative to `screen` 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
(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      else
385
386
387      printf ( "no arguments were given, using default: %d\n\n", DEFAULT_ARG1 );
B+> 0x555555554d38 <main+15>    cmpl    $0x1,-0x14(%rbp)
0x555555554d3c <main+19>    jle     0x555555554db7 <main+142>
0x555555554d3e <main+21>    cmpl    $0x2,-0x14(%rbp)
0x555555554d42 <main+25>    jle     0x555555554d4b <main+34>
0x555555554d44 <main+27>    mov     $0x73,%edx
0x555555554d49 <main+32>    jmp     0x555555554d50 <main+39>
0x555555554d4b <main+34>    mov     $0x20,%edx
0x555555554d50 <main+39>    mov     -0x14(%rbp),%eax
0x555555554d53 <main+42>    sub     $0x1,%eax
0x555555554d56 <main+45>    mov     %eax,%esi
0x555555554d58 <main+47>    lea     0x2bf(%rip),%rdi      # 0x5555555501e
0x555555554d5f <main+54>    mov     $0x0,%eax
0x555555554d64 <main+59>    callq   0x555555554680 <printf@plt>
0x555555554d69 <main+64>    movl   $0x1,-0xc(%rbp)

native process 9102 In: main
(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>

The screenshot shows the GDB dashboard interface with several panes:

- Source:** Displays the C code for `scrot.c`, including the `fun+74` function and the `main+34` function.
- Assembly:** Shows the assembly code for the current instruction at address `0x00000000100000f2a`.
- Threads:** Lists two threads: thread 1 (id 4355) and thread 0 (id 0).
- Stack:** Displays the stack frames for both threads.
- Registers:** Shows the register values for both threads.
- Expressions:** Shows the value of `data[i]` as `0xffff5fbffcf0 "hello"`.
- Memory:** Shows memory dump from `0x00007ffff5fbffbf0` to `0x00007ffff5fbffbf3c`.
- History:** Shows the command history with entries like `dashboard -output /dev/ttys001` and `dashboard -layout`.



GUI for
GDB

GDBgui



<https://gdbgui.com/>

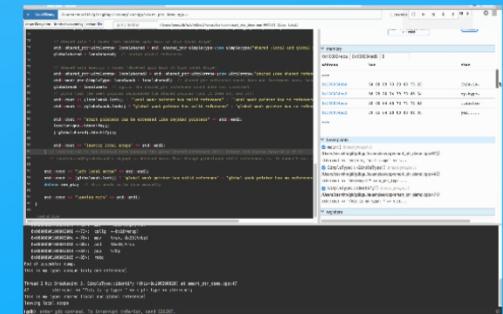
[gdbgui](#) Download Docs Examples Screenshots Videos About Chat GitHub



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 program was configured for "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 »](#)

[Documentation](#)

that's all, have fun

"So long
and thanks
forall the fish"