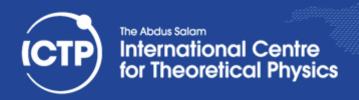




# Debugging & Profiling with Open Source SW Tools

Ivan Girotto – igirotto@ictp.it

Information & Communication Technology Section (ICTS)
International Centre for Theoretical Physics (ICTP)





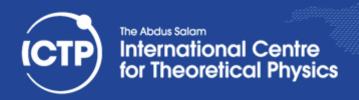
## What is Debugging?!

- Identifying the cause of an error and correcting it
- Once you have identified defects, you need to:
  - find and understand the cause
  - remove the defect from your code
- In a large number of cases bug fixes are wrong:
  - they remove the symptom, but not the cause
- Improve productivity by getting it right the first time
- A lot of programmers don't know how to debug!
  - Doesn't add functionality & doesn't improve the science
- Debugging needs practice and experience:
  - understand the science and the tools





Lot of time debugging. We did learn also from it, but I have the feeling we could have learnt more things about Quantum Espresso if we hadn't had to be debugging for so long (some of the bugs we had were due to our lack of excellence in programming skills and were not specific to QE issues) (Cit. from ICTP Activity evaluation)





#### **Errors are Opportunities**

- Learn from the program you're working on:
  - Errors mean you didn't understand the program. If you knew it better, it wouldn't have an error. You would have fixed it already
- Learn about the kinds of mistakes you make:
  - If you wrote the program, you inserted the error
  - Once you find a mistake, ask yourself:
    - Why did you make it?
    - How could you have found it more quickly?
    - How could you have prevented it?
    - Are there other similar mistakes in the code?





## The Nature of Bugs

- Straightforward bug to intercept and solve
- The program crashes unexpectedly
  - the problem can be easily reproduced (lucky)
  - bug whose causes are too complex to be reliably reproduced; it thus defies repair
  - bug disappears when debugging a problem (compiling with -g or adding prints)
- The produced numbers differ from what we expected
  - bug generated by an invalid operations
  - bug disappears when debugging a problem (compiling with -g or adding prints)





## Main Reasons of Debugging

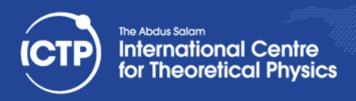
- Floating Point Exceptions (FPE)
  - Overflow
  - Invalid Number
  - Division by Zero
- Out of bound
- Segmentation Fault
- Not expected execution flow
- The Program Hangs





#### Purpose of a Debugger

- More information than print statements
- Allows to stop/start/single step execution
- Look at data and modify it
- 'Post mortem' analysis from core dumps
- Prove / disprove hypotheses
- No substitute for good thinking
- But, sometimes good thinking is not a substitute for effectively using a debugger!
- Easier to use with modular code





#### **Approaches**

- Print Messages and Variables ©
- Compiler Debug Options
- Core analysis
- Run the Program with a Debugger
- Attach Debugger to a running process
- Ask for help!





## Using a Debugger

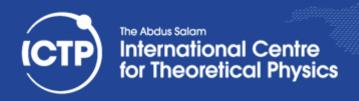
- When compiling use -g option to include debug info in object (.o) and executable (and possibly -O0)
- 1:1 mapping of execution and source code only when optimization is turned off
  - problem when optimization uncovers bug
- GNU compilers allow -g with optimization
  - not always correct line numbers
  - variables/code can be 'optimized away'
  - progress confusing with loop unrolling





## Using gdb as a Debugger

- gdb ex01-c launches debugger, loads binary, stops with (gdb) prompt waiting for input:
- run starts executable, arguments are passed Running program can be interrupted (ctrl-c)
- gdb./prog --args arg1 -flag passes all arguments to the run command inside gdb
- continue continues stopped program
- finish continues until the end of a subroutine
- step single steps through program line by line
- next single steps but doesn't step into subroutines





#### More Basic gdb Commands

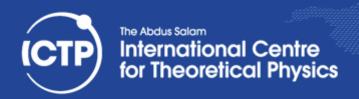
- print displays contents of a known data object
- display is like print but shows updates every step
- where shows stack trace (of function calls)
- up/down allows to move up/down on the stack
- break sets break point (unconditional stop), location indicated by file name+line no. or function
- watch sets a conditional break point (breaks when an expression changes, e.g. a variable)
- delete removes display or break points





#### Post Mortem Analysis

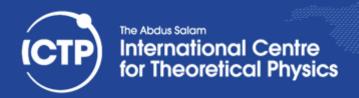
- Enable core dumps: ulimit -c unlimited
- Run executable until it crashes; will generate a file core or core.<pid> with memory image
- Load executable and core dump into debugger gdb myexe core.<pid>
- Inspect location of crash through commands: where, up, down, list
- Use directory to point to location of sources





## Using valgrind

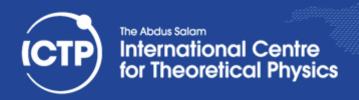
- Run valgrind -v ./exe to instrument and run
- --leak-check=full --track-origins=yes
- Output will list individual errors and summary
- With debug info present can resolve problems to line of code, otherwise to name of function
- Also monitors memory allocation / deallocation to flag memory leaks ("forgotten" allocations)
- Instrumentation slows down execution
- Can produce "false positives" (flag non-errors)





## How to NOT do Debugging

- Find the error by guessing
- Change things randomly until it works (again)
- Don't keep track of what you changed
- Don't make a backup of the original
- Fix the error with the most obvious fix
- If wrong code gives the correct result, and changing it doesn't work, don't correct it.
- If the error is gone, the problem is solved.
   Trying to understand the problem, is a waste of time





## **Debugging Tools**

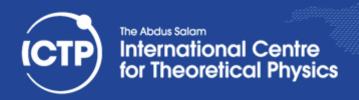
- Source code comparison and management tools: diff, vimdiff, emacs/ediff, cvs/svn/git
  - Help you to find differences, origins of changes
- Source code analysis tools: compiler warnings, ftnchek, lint
  - Help you to find problematic code
    - Always enable warnings when programming
    - Always take warnings seriously (but not all)
    - Always compile/test on multiple platforms
- Bounds checking allows checking of (static) memory allocation violations (no malloc)





## More Debugging Tools

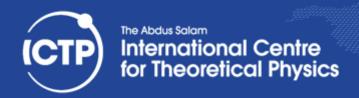
- Using different compilers (Intel, GCC, Clang, ...)
- Debuggers and debugger frontends:
   gdb (GNU compilers), idb (Intel compilers), ddd (GUI),
   eclipse (IDE), and many more...
- gprof (profiler) as it can generate call graphs
- valgrind, an instrumentation framework
  - Memcheck: detects memory management problems
  - Cachegrind: cache profiler, detects cache misses
  - Callgrind: call graph creation tool





## How to Report a Bug(?) to Others

- Research whether bug is known/fixed
  - web search, mailing list archive, bugzilla
- Provide description on how to reproduce the problem. Find a minimal input to show bug.
- Always state hardware/software you are using (distribution, compilers, code version)
- Demonstrate, that you have invested effort
- Make it easy for others to help you!

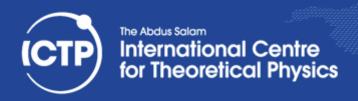




## **Profiling**

- Essential operation for code optimization
- Profiling usually means:
  - Instrumentation of code (e.g. during compilation)
  - Automated collection of timing data during execution
  - Analysis of collected data, breakdown by function
- Example: gcc -o some\_exe.x -pg some\_code.c
  - ./some\_exe.x
  - gprof some\_exe.x gmon.out
- Profiling is often incompatible with code optimization or can be misleading (inlining)

```
[igirotto@argo-login2 C_source]$ icc =03 transport_serial.c -pg -g
[igirotto@argo-login2 C_source]$ ./a.out
initialization done
cpu time in seconds 0.000751
                ransport Programming Exercise
evolution done
cpu time in seconds 0.8
A taball for the files needed for this exercise are available at transport-source.tgz
save_data done
IO time in seconds 1.49
total cpu time in seconds 2.29
[igirotto@argo-login2 C_source]$ gprof ./a.out gmon.out
Flat profile:
d/dx + d/dy = -d/dt
Each sample counts as 0.01 seconds.
  % cumulative
                    self
                                        self
                                                  total
                            ___calls ms/call__
                                                ms/call
 time
        seconds
                   seconds.
                                                          name tion. The points of the grid represen
                               15000
                                                          evolve represented in white. Data are e
 93.69
            0.74
                      0.74
                                          0.05
                                                0.05
  2.53dinate sv0.76
                      0.02
                                                           __intel_ssse3_rep_memcpy
                      0.01
                               30000
                                          0.00
  1.27
            0.77
                                                    0.00
                                                          ix2x
                                                          update_boundartes_PBCmple with t
  1.27
            0.78
                      0.01
                               15001
                                          0.00
                                                    0.00
                                                          _intel_fast_memcpy
  1.27
            0.79
                      0.01
  0.00
            0.79
                      0.00
                                 300
                                          0.00
                                                    0.00
                                                           iy2y
                      0.00
                               ---8---
                                                          seconds end dat', the data after the tran
  0.00
            0.79
                                          0.00
                                                    0.00
  0.00
            0.79
                      0.00
                                          0.00
                                                    3.33
                                                          save_gnuplot
  0.00
            0.79
                                          0.00
                      0.00
                                                    3.33
                                                           init_transport
                                                                      en3d; splot 'transport.
```





#### PERF – Hardware Assisted Profiling

- Modern x86 CPUs contain performance monitor tools included in their hardware
- Linux kernel versions support this feature which allows for very low overhead profiling without instrumentation of binaries
- perf stat ./a.out -> profile summary
- perf record ./a.out; perf report -i perf.data
- gprof like function level profiling (with coverage report and disassembly, if debug info present)

```
3K of event 'cycles', Event count (approx.): 1847839734
Samples:
         a.out
                 a.out
                                         evolve
                                          __intel_ssse3_rep_memcpy
         a.out
                 a.out
                 libc-2.12.so
  1.29%
                                          __printf_fp
         a.out
                                         0xffffffff8103ba6a
                 [kernel.kallsyms]
 0.65%
         a.out
 0.61%
         a.out
                 a.out
                                         _intel_fast_memcpy
 0.55%
                                          update_boundaries_PBC
         a.out
                 a.out
                                         __mpn_mul_1
 0.41%
                 libc-2.12.so
         a.out
 0.25%
                 libc-2.12.so
                                         hack_digit.15673
         a.out
                                         vfprintf
 0.15%
                 libc-2.12.so
         a.out
 0.13%
                 libc-2.12.so
                                          __isinf
         a.out
                                         _intel_fast_memcpy.P
 0.12%
         a.out
                 a.out
                                         __mpn_rshift
 0.12%
                 libc-2.12.so
         a.out
 0.09%
         a.out
                 libc-2.12.so
                                           _mcount_internal
                                          __strlen_sse2
 0.09%
                 libc-2.12.so
         a.out
 0.08%
                                          exp.L
         a.out
                 a.out
 0.07%
                 libc-2.12.so
                                         __strchrnulncluded in their
         a.out
 0.06%
                                         ix2x
         a.out
                 a.out
 0.03%
                 libc-2.12.so
                                         __mpn_di∨rem
         a.out
 0.03%
                 libc-2.12.so
         a.out
                                          _mcount
                                         __mpn_lshift
 0.03%
                 libc-2.12.so
         a.out
                                          __mpn_extract_double
 0.03%
         a.out
                 libc-2.12.so
                                     [.] _IO_file_xsputn@@GLIBC_2.2.5
 0.03%
                 libc-2.12.so
         a.out
Ivan Girotto
                      Debugging & Profiling with Open Source SW Tools
                                                                        21
```





## Function for Internal Profiling

```
#include <time.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/time.h>
double seconds()
  /* Returns elepsed seconds past from the last call to timer rest */
    struct timeval tmp;
    double sec:
    gettimeofday( &tmp, (struct timezone *)0 );
    sec = tmp.tv sec + ((double) tmp.tv usec) / 1000000.0;
    return sec;
```







convergence	пот	achieved a	fter	5 iterat	ions: s	topping			
Writing outp	ut d	ata file c	8_atn	1213_k111.s	save				
init_run		93.79s	CPU	93.79s	WALL (	1	calls)		
electrons		961.37s	CPU	961.37s	WALL (	1	calls)		
Called by in	it_ri	un:							
wfcinit		69.37s	CPU	69.37s	WALL (	1	calls)		
potinit		4.76s	CPU	4.76s	WALL (	1	calls)		
Called by el	ectr	ons:							
c_bands	:	883.32s	CPU	883.32s	WALL (	5	calls)		
sum_band	:	40.30s	CPU	40.30s	WALL (	5	calls)		
v of rho		1.10s	CPU	1.10s	WALL (	6	calls)		
c_bands sum_band v_of_rho mix_rho		1.51s	CPU	1.51s	WALL (	5	calls)		
Called by c_									
init_us_2			CDII	0.50c	MALL C	11	calle)		
cegterg		882.015	CPU (PU	882.015	WALL (	5	calls)		
cogoo. g		302.013		0021013			carro		
Called by *e	gter	g:							
h_psi g_psi cdiaghg		259.11s	CPU	259.11s	WALL (	17	calls)		
g_psi		9.02s	CPU	9.02s	WALL (	11	calls)		
cdiaghg		401.37s	CPU	401.37s	WALL (	16	calls)		
Called by h_	psi:								
add_vuspsi		22.44s	CPU	22 <b>.44</b> s	WALL (	17	calls)		
General rout	ines								
calbec	4	17,25s	CPU	17.25s	WALL (	17	calls)		
fft		0 529	CPII	0.525	WALL	66	calls		
ffts	:	0.63s	CPU	0.63s	WALL (	117	calls)		
fftw		231.61s	CPU	231.61s	WALL (	10260	calls)		
ffts fftw davcio		4.72s	CPU	4.72s	WALL (	5	calls)		
Parallel rou	tine	5							
fft_scatter			CPU	63.51s	WALL C	10443	calls)		
ALLTOALL				10.675	WALL (	10252	calls)		
EXX routines		10.003		1013		10232			
DWCCE		7::42 04: 5	DII	17:42 04	WALL				
PWSCF	: 1	/m42.94s (	PU	17m42.94s	WALL				

m_retrig_out	put (	lata file c8_at		Jave			
init_run		119.48s CPU	119.48s	WALL	(	1	calls)
electrons		1369.53s CPU	1369.53s	WALL	(	1	calls)
Called by i	init_r	un:					
wfcinit		98.55s CPU	98.55s	WALL	(	1	calls)
potinit		98.55s CPU 2.15s CPU	2.15s	WALL	(	1	calls)
Called by e	electr	ons:					
c_bands		1289.41s CPU 56.06s CPU	1289.41s	WALL	(	5	calls)
sum_band		56.06s CPU	56.06s	WALL	(	5	calls)
v_of_rho		1.39s CPU	1.39s	WALL	(	6	calls)
mix_rho		1.23s CPU	1.23s	WALL	(	5	calls)
Called by a	_bana	ls:					
init_us_2		0.13s CPU 1288.89s CPU	0.13s	WALL	(	11	calls)
cegterg		1288.89s CPU	1288.89s	WALL	(	5	calls)
Called by *	egter	·g:					
h_psi		409.59s CPU 2.35s CPU 528.61s CPU	409.59s	WALL	(	17	calls)
g_psi		2.35s CPU	2.35s	WALL	(	11	calls)
cdiaghg	:	528.61s CPU	528.61s	WALL	(	16	calls)
Called by h							
add_vuspsi		32.96s CPU	32.96s	WALL	(	17	calls)
General rou	ıtines						
calbec		31.22s CPU 0.62s CPU 0.86s CPU 376.02s CPU	31.22s	WALL	(	17	calls)
fft		0.62s CPU	0.62s	WALL	(	66	calls)
ffts		0.86s CPU	0.86s	WALL	(	117	calls)
fftw		376.02s CPU	376.04s	WALL	(	82004	calls)
davcio		6.38s CPU	6.38s	WALL	(	5	calls)
Parallel ro							
fft_scatter	:	81.64s CPU	81.65s	WALL	(	82187	calls)
PWSCF	: 2	24m57.48s CPU	24m57.48s	s WALL			





## Profiling in Python

- individual functions:
  - import cProfile
  - cProfile.run('some\_func()', 'profile.tmp')
- whole script:
  - python -m cProfile [-o output\_file] [-s sort\_order] myscript.py
- Analyze profile file:
  - import pstats
  - p = pstats.Stats('profile.tmp')
  - p.strip\_dirs().sort\_stats(-1).print\_stats()
- More info at <a href="http://docs.python.org/2/library/profile.html">http://docs.python.org/2/library/profile.html</a>





## **Debugging Python**

- typically very easy to do interactively with "print()" and "exit()" statements in the code
- More featureful debugger available in module "pdb", see:
  - http://docs.python.org/2.7/library/pdb.html







#### References

PERF wiki