Exercise 2 – Profiling

Code profiling is an activity operated on a program to aid software optimization by, for instance:

- measuring space/time complexity of a program
- measuring frequency and duration of function calls
- identifying callers and callees of function calls

This is obtained by using a tool called profiler, which gets the aforementioned informations via different variety of techniques.

The code I chose to profile is a (very inefficient) program I wrote in C to invert a square matrix. It's called `squarematrix.c`.

The software is structured in this way:

- The inverse is calculated in the following way:
  - Given an $(n \times n)$ matrix A, the minor or the matrix for row i and column j ($A_{i,j}$) is the determinant of the submatrix of A obtained by removing row i and column j from A itself.
  - The inverse is then: $1/det(A) \cdot \{(-1)^{i+j} \cdot A_{i,j}\}_{i,j \in \{1,\dots,n\}}$
- To calculate the inverse, one has to calculate the determinant as well. This is done again by using minors:
  $det(A) = \sum_{j=1}^{n} (-1)^{i+j} \cdot A_{i,j}$ with i an arbitrary row (in the program it's row 1 of the matrix).
- The program hence has three core functions:
  - `inverse`
  - `determinant`
  - `get_minors`

  both inverse and determinant call get_minors, which, on the other hand, calls determinant to calculate the minor(s) (which is/are itself a determinant of a smaller matrix).
- There're some minor functions, utilities to print and initialize the (initial) matrix.

The program runs for matrices of side fixed at 10 (→ total size of 100).

Profiling
I used two tools for profiling: gprof and valgrind.

1. **GPROF**
   Gprof is a Unix profiler which injects code into the source to gather information on function calls and callers, and measure time spent in calls via sampling (i.e. checking which function the program is running each *x* seconds, usually 0.01).
   To enable profiling, the code must be compiled regularly using (for instance) `gcc` with the `-pg` flag. The executable is then run regularly; after this run, a gprof dump file called `gmon.out` will be produced.
   By typing `gprof ./my-executable gmon.out > profiling.txt`, gprof will produce a text file containing the "flat profile" of the program, i.e. the summary of the calls to the functions and the time spent in all calls during the aforementioned run.

The main function in my program first initializes a 10x10 matrix, then calculates and outputs the determinant, then calculates the inverse. I then expect a large call over determinant, and a smaller call over get_minors, and one call over inverse. Let's see what gprof produces:

This is the *flat profile* for my program:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 72.85      0.64     0.64 31278000     0.00     0.00  ablate_matrix
 15.94      0.78     0.14 31278002     0.00     0.00  determinant
  9.68      0.87     0.09  9505202     0.00     0.00  get_minors
  1.14      0.88     0.01        1    10.02    10.02  initialize_matrix
  0.00      0.88     0.00        2     0.00     0.00  print_matrix
  0.00      0.88     0.00        1     0.00   857.99  inverse
```

What can be immediately noticed is that, as expected, there's a large number of **calls** to determinant, and an almost-as-large number of calls to `ablate_matrix`, a subroutine of `get_minors` that returns the submatrix obtained by removing one arbitrary row and column from the given matrix. As far as time is concerned, we can see that the program spends the majority of time in `ablate_matrix` (72.85%), while the remaining time is mainly spent in `determinant` (~16%) and `get_minors` (~10%). That result must be taken with a grain of salt however, since the same.

Gprof can also produce a call graph (a graph in which functions are nodes and the oriented edges represent calls), but the output is a text file which is hardly human readable. For another profiling, and for a more readable call graph, I'll resort to **valgrind**.

2. VALGRIND

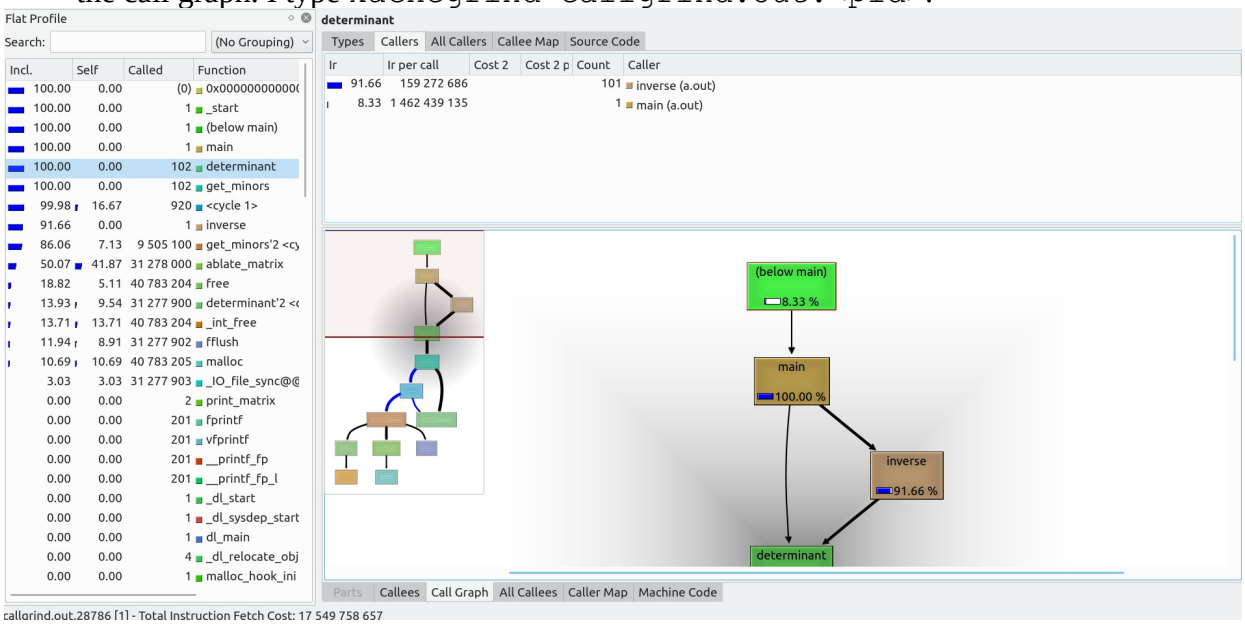
Valgrind is far more than a profiler: profiling is one of its task, but it may also be used for memory debugging and memory leak detection.
Instead of injecting code and sampling, as gprof, valgrind instead runs the program on a virtual machine intercepting what he's asked to check: dynamical memory allocation and subsequent `free`s, calls, etc.
Compilation doesn't need any special flag on, but we're advised to turn on at least the debugging flag (`-g`), otherwise …
First, to check if there're memory leaks, we can run valgrind's memcheck tool by typing
`valgrind --tool=memcheck ./my-executable`
This is the output:

```
==28741== HEAP SUMMARY:
==28741==     in use at exit: 0 bytes in 0 blocks
==28741==   total heap usage: 40,783,205 allocs, 40,783,205 frees,
1,795,757,664 bytes allocated
==28741==
==28741== All heap blocks were freed -- no leaks are possible
==28741==
==28741== For counts of detected and suppressed errors, rerun with: -v
==28741== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

That means that there're no memory leaks and all dynamically allocated pointers have been `free`d during the program execution.

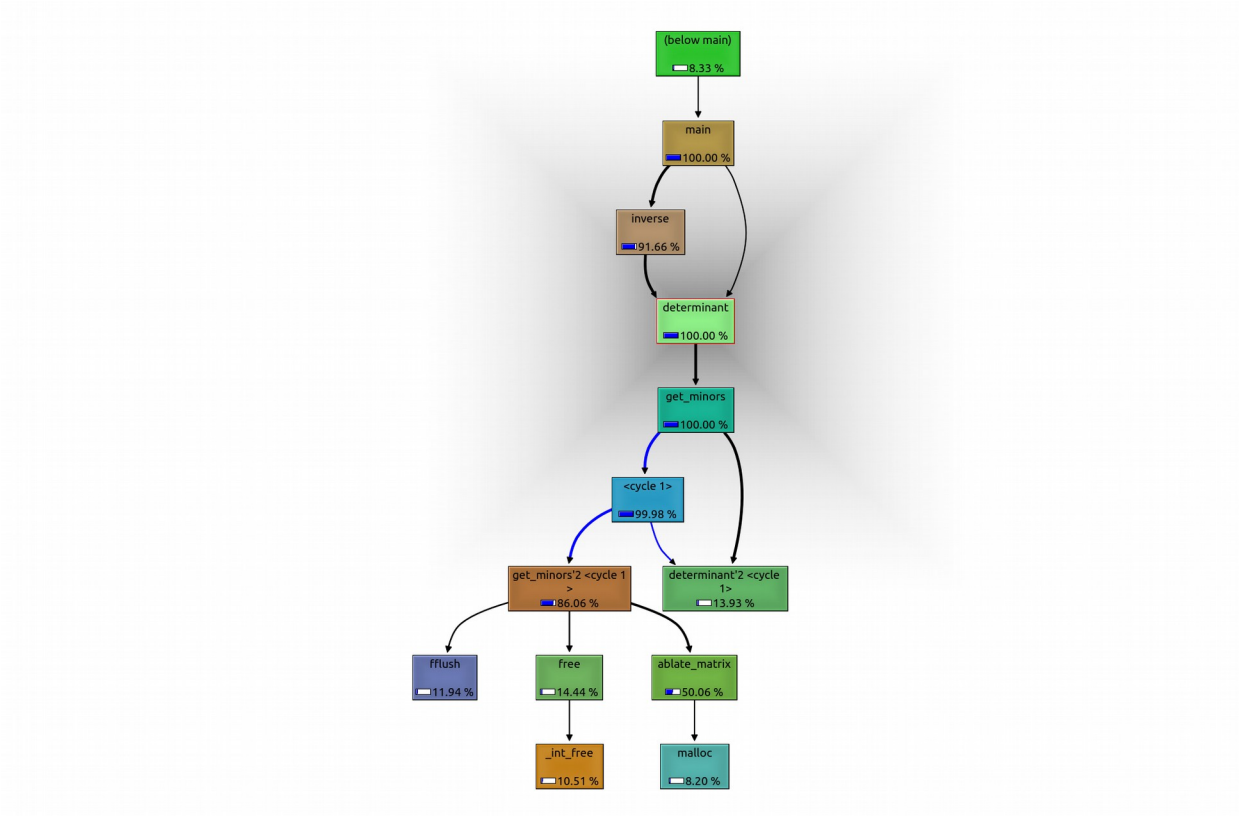
I re-run the program with valgrind, this time using the callgrind tool. This will intercept calls and callers and produce a call graph with a specification of time spent in each call:
`valgrind --tool=memcheck ./my-executable`

What can first be noted is that the execution of the program within the valgrind virtual machine is far slower than a normal execution, or the execution using gprof tool, this will, though, have positive result on the output I'll get, as the call graph will be indicating the correct time spent in calls.
After the run, a file `callgrind.out.<pid>` will be produced, containing the profiling information.
There exists a GUI tool, called **kcachegrind**, which enables an user-friendly visualization of the call graph. I type `kachegrind callgrind.out.<pid>`:



This is how the GUI looks like. To the left there's the **flat profile**, similar to how we saw it on gprof, but with a more extensive list of functions.
To the right, on the lower screen we have the **call graph**, above we can choose a detailed view on some informations about the function selected in the flat profile.

The flat profile is ordered by "Inclusive cost", the time "cost" of the function + time spent in all of the function it calls. The time spent in the function *per se* is the "Self cost". I can see that the most *costly* function in the code is `ablate_matrix`, while determinant and `get_minors`, while having an extremely large inclusive cost, all they do the majority of the time is to call themselves or other functions: in the end, what determinant really does, besides recursive calls, is to do 2 multiplications and 1 sum in order to return the determinant of a 2x2 submatrix; get_minors, instead; it has a 10-iterations cycle in which it calls other functions and does 1 free per cycle, hence its self time of 7.13% of the program execution. `Ablate_matrix`, on the other hand, has a `malloc`, then one nested-cycles of 9 iterations each (actually, two nested cycles all initializing one half of the newly created submatrix) and some FLOPs to obtain the element of the matrix to be initialized. The other functions are mainly I/O routines.
As far as inclusive cost is concerned, other than `main`, we see that `determinant` is the dominating function here, being called everywhere in the program.
The function most frequently called is `malloc` (along with its counterpart, `free`), with other 40M calls, then we have `ablate_matrix`; after that, we need to resort to the call graph to better understand what valgrind is telling, as there's a cycle in the program with `determinant` and `get_minors` repeated:

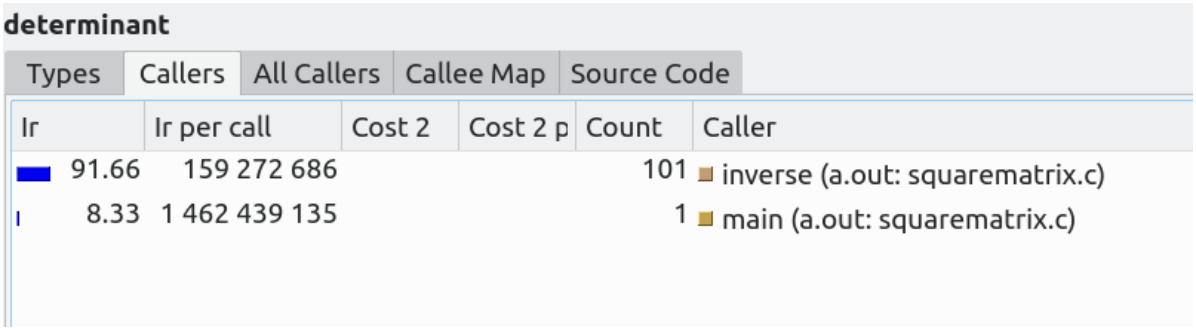


The graph tells the following:

- `Main` calls both `inverse` and `determinant`; the latter is later called by the former.
- The majority of the computation is spent in obtaining the `inverse` (91.66%); the remaining part is needed in order to get the first `determinant` (8.34%, by exclusion).
- `determinant` then calls `get_minors`; the self time spent in the first call to `determinant` is negligible, hence the 100% of inclusive time for both `determinant` and `get_minors`
- We then enter a cycle where `get_minors` and `determinant` get recursively called (`get_minors<cycle1>, determinant<cycle1>`). A true and reliable call graph would indicate all the recursive calls in a very long loopy graph, but that would be so long that it would be unreadable. A highly recursive program like that isn't the best example for a callgraph.
- In the end, `get_minors` operates I/O calls and some `frees`, and `ablate_matrix` does `mallocs`, but this is a rather simplified view: also `determinant` operates `frees`, and some I/O calls are executed by `main` as well; moreover, other minor functions (i.e. `initialize_matrix`) are called by `main`; all these calls aren't shown in the call graph (probably because the time spent in these calls is neglegible). One could argue, though, that this isn't important in profiling, since we need to find bottlenecks and/or part of the code on which to focus optimization, and then we need only the most *costly* function calls.

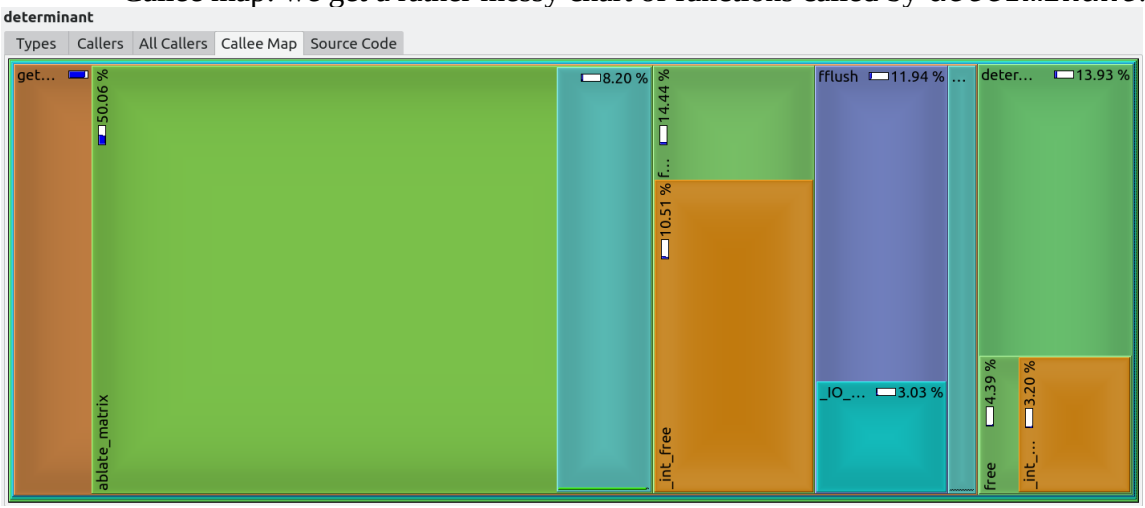
By inspecting the upper-right panel, instead, I can see infos about functions selected in the left panel. Let's see an example with `determinant` for two important functionalities:

- Callers: it's called (outside the cycles) 101 times by `inverse` (100 times to calculate the 100 minors inside the 10x10 inverse matrix, plus 1 to calculate the determinant at the denominator of the formula), and 1 time by `main`.



The tab “All Callers” is an extended view of “callers” where we see also function who called functions calling `determinant`.

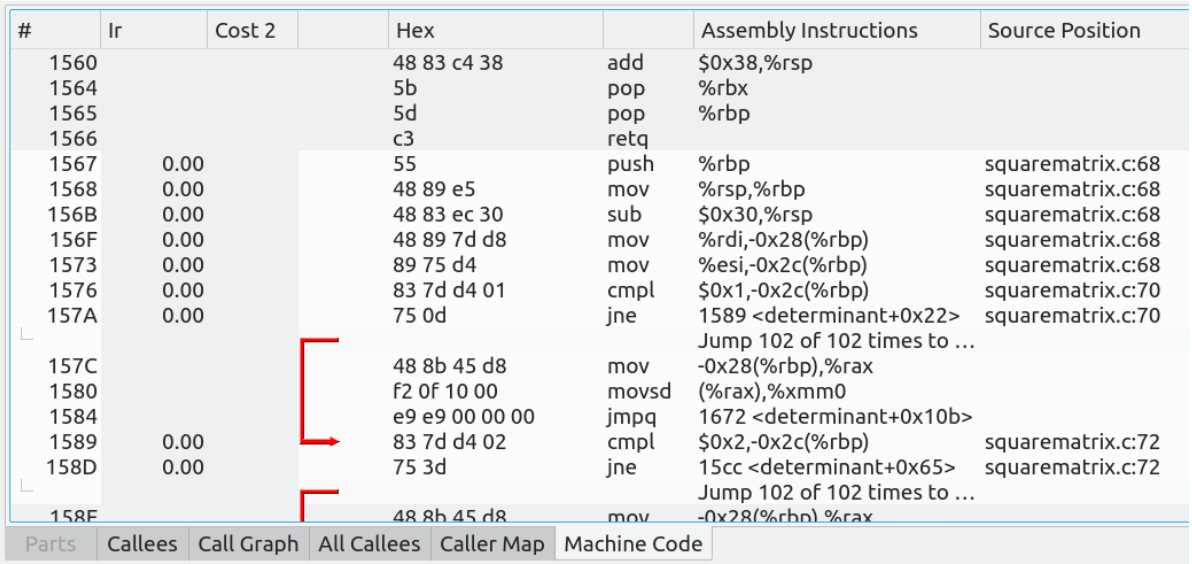
- Callee map: we get a rather messy chart of functions called by `determinant`:



`Determinant` calls `get_minors`, which calls `ablate_matrix` where the program spends the most time into, then we can only see some of the function names (`free`, `fflush`, then again `determinant` - *inside the cycle*).
One can navigate both the map and the callers menu above by double clicking the selected function, and see either the portion of the map on a finer grain, or the functions called by the selected function.

On the other hand, in the bottom-right pane we can see a similar menu with the callees called by the function selected in the left pane, and the caller map, similar to the callees map, but showing the share of functions calling `determinant`.

An interesting note: by calling valgrind with flags `--dump-instr=yes --collect-jumps=yes`, one can inspect the assembly code with conditional jumps corresponding to the selected function in the “Machine Code” slide (bottom-right pane) – again the view for `determinant`:



#	Ir	Cost 2		Hex		Assembly Instructions	Source Position
1560				48 83 c4 38	add	$0x38,%rsp	
1564				5b	pop	%rbx	
1565				5d	pop	%rbp	
1566				c3	retq		
1567	0.00			55	push	%rbp	squarematrix.c:68
1568	0.00			48 89 e5	mov	%rsp,%rbp	squarematrix.c:68
156B	0.00			48 83 ec 30	sub	$0x30,%rsp	squarematrix.c:68
156F	0.00			48 89 7d d8	mov	%rdi,-0x28(%rbp)	squarematrix.c:68
1573	0.00			89 75 d4	mov	%esi,-0x2c(%rbp)	squarematrix.c:68
1576	0.00			83 7d d4 01	cmpl	$0x1,-0x2c(%rbp)	squarematrix.c:70
157A	0.00			75 0d	jne	1589 <determinant+0x22>	squarematrix.c:70
						Jump 102 of 102 times to …	
157C				48 8b 45 d8	mov	-0x28(%rbp),%rax	
1580				f2 0f 10 00	movsd	(%rax),%xmm0	
1584				e9 e9 00 00 00	jmpq	1672 <determinant+0x10b>	
1589	0.00			83 7d d4 02	cmpl	$0x2,-0x2c(%rbp)	squarematrix.c:72
158D	0.00			75 3d	jne	15cc <determinant+0x65>	squarematrix.c:72
						Jump 102 of 102 times to …	
158F				48 8b 45 d8	mov	-0x28(%rbp),%rax	

Parts | Callees | Call Graph | All Callees | Caller Map | Machine Code

In conclusion, I presented two profiling tool:

- gprof, a “vanilla” profiler, maybe a bit outdated, that lets us have a glance at what may be a hotspot in a program, i.e. where to focus the attention in a possible optimization. It has a call graph functionality but it’s not user friendly;
- valgrind (callgrind tool), a powerful profiler, maybe more accurate than gprof, but at a cost: it runs much slower (~10x slower than a normal execution of the program). valgrind, coupled with kcachegrind, lets us view the profiling output with a (mostly) intuitive GUI which gives far more insights than gprof.

Both profilers, albeit with different timings, indicate that the function where the program spends most of the time is `ablate_matrix`: this because it has to `malloc` and initialize a submatrix with a size up to 89 elements, and it gets called more than 31M times (both gprof and valgrind report the same number), so it consumes both time (around 41% of the execution time, according to valgrind) and space (it creates a huge number of submatrices, all `malloc`’d). This is the reason why the execution time explodes with the problem size.
If I want to further optimize my code, I should focus at this function. One way to optimize it would be: instead of storing the submatrix in a newly `malloc`’d array, I could re-use the same matrix in `get_minors`, by cleverly and carefully playing with the indices.