HPC Tools 2024, Exercise 2 solutions

Bruno Oliveira Cattelan, Laurent Chôné

Problem 1

In order to investigate the limit at which access does not cause a crash, one can loop over the index, and print it to the console before accessing `a[idx]`. Piping the output (without the error stream) to a file can be achieved using:

```
$ ./prog > progData.txt
```

One can then collect the last indices of each run to an ASCII file using bash (make sure `oldData.txt` is initialised):

```
( cat oldData.txt ; tail -n 1 progData.txt ) > newData.txt
mv newData.txt oldData.txt
```

Automating this to collect many samples yields the data in Figure 1:

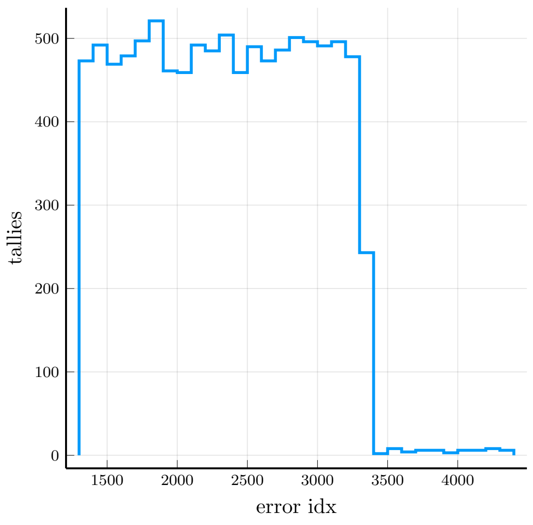


Figure 1: Histogram of the highest index before a crash, over 10000 runs of a C++ program.

We can see that, not only the minimum index causing a segmentation fault is much greater than the actual size of the array, but it also fluctuates significantly from one run to the next. As a result, accessing data beyond bounds can both pollute data without causing errors, and cause segmentation faults with an inconsistent pattern, making it difficult to debug without using proper tools.

Problem 2

In this problem, we investigate the crash in the program of problem 1 using `gdb`. In order to allow the debugger to inspect the inner workings of the program, it must be compile with the appropriate flag. E.g. the present `C++` program is compiled and launched as:

```
$ g++ -g ex02pb01.cpp -o prog
$ gdb prog
...
(gdb) run
...
Program received signal SIGSEGV, Segmentation fault.
0x0000555555555243 in main () at ex02pb01.cpp:16
16              value = array[idx];
```

While we are in the gdb console, we are able to inspect variables in their last state using the `print` command. For the sake of illustration, only half of the values of `array` where initialised to $a_i = i^2$. Printing `array` shows that it does contain 10 elements, with the uninitialised elements containing junk data. Printing `idx` shows that the program crashed for `idx`$= 1320$. Interestingly, running the code several times *in the same terminal* produces the same behaviour, including after exiting `gdb` and recompiling the program.

```
(gdb) print array
$1 = {0, 1, 4, 9, 16, 32767, -134562168, 32767, -134563520, 32767}
(gdb) print idx
$2 = 1320
```

Problem 3

In Figure 2 we show the runtimes of each program version. As expected, we see that for this optimization option (-O1) float is the fastest option, closely followed by double. Long or float128 is the slowest. This can be explained by the fact that float128 is not implemented in hardware for most computers, which was the case in our experimental setup. Double and float are quite similar, but since float is half the size of double, it makes sense that without any advanced optimizations this difference would be visible.

It is also important to note the variance in runtimes for each program. This indicates the need for statistical analysis of program runtimes. This is specially true of faster programs, where different concurrent processes can have a larger impact on their perceived runtime. In the case of this exercise we can see that most likely executing only once would have been enough to see the difference between them.

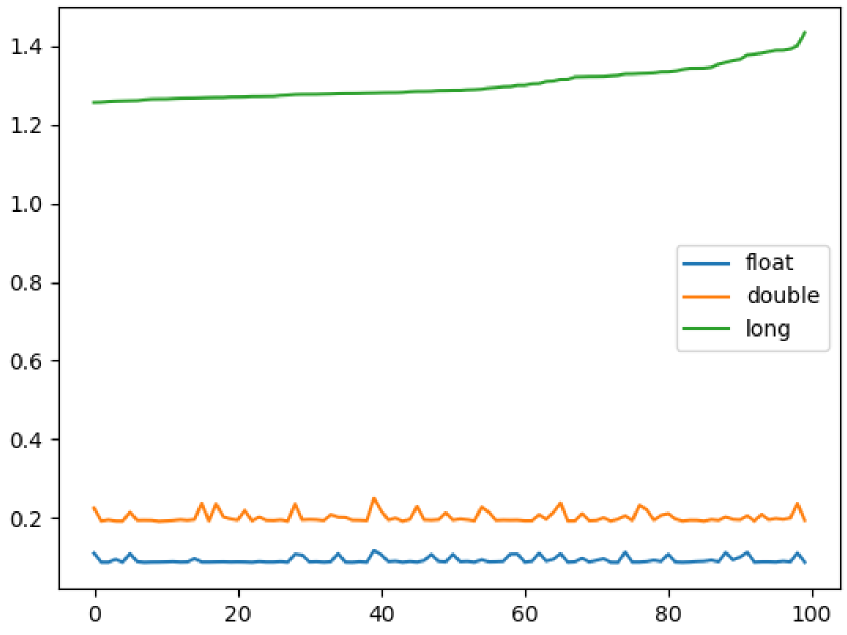
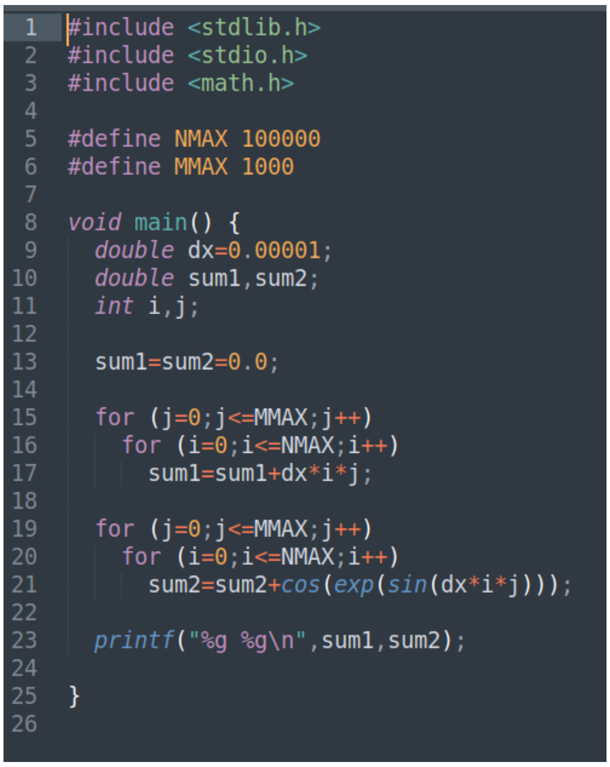


Figure 2: Runtimes of float, double and float128.

Problem 4

For this problem we were given a code both in C and Fortran. Because I am more comfortable with C, my example will be using it. However, the difference between the two should be minimal. The code is in Figure 3.

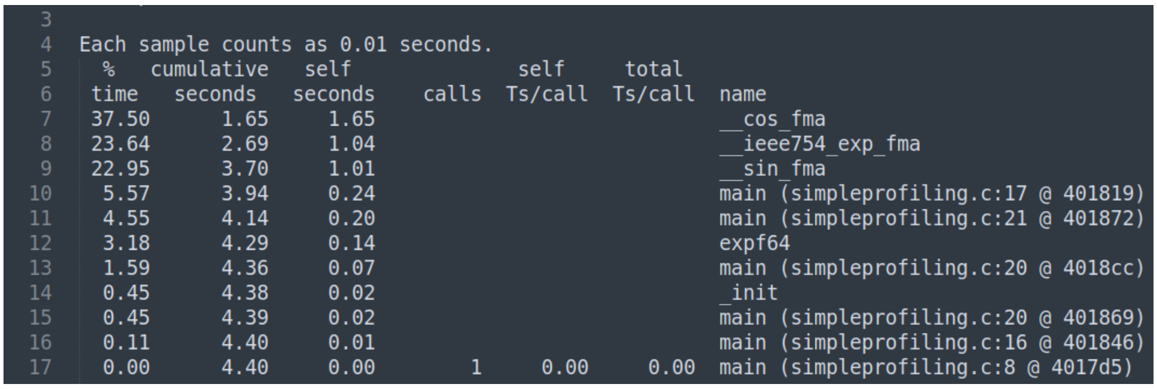


```c
 1  #include <stdlib.h>
 2  #include <stdio.h>
 3  #include <math.h>
 4
 5  #define NMAX 100000
 6  #define MMAX 1000
 7
 8  void main() {
 9    double dx=0.00001;
10    double sum1,sum2;
11    int i,j;
12
13    sum1=sum2=0.0;
14
15    for (j=0;j<=MMAX;j++)
16      for (i=0;i<=NMAX;i++)
17        sum1=sum1+dx*i*j;
18
19    for (j=0;j<=MMAX;j++)
20      for (i=0;i<=NMAX;i++)
21        sum2=sum2+cos(exp(sin(dx*i*j)));
22
23    printf("%g %g\n",sum1,sum2);
24
25  }
26
```

Figure 3: Code profiled

In Figure 4 we show the stack of different calls and the time the program spent in them. If we look for the time spent in lines 17 and 21 (inside the loops) we would see the times 0.24 for line 17 and 0.21 for line 21. One might ask themselves, is this correct? Why is line 17 slower than line 21 when the latter does so much more work?

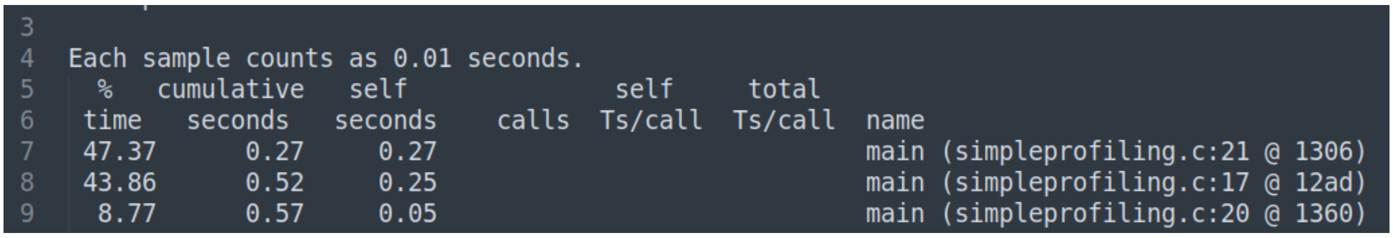
This is because for line 21 we also need to take into account the time spent inside the functions called. With that in mind, the time spent in it jumps from 4.14 to $4.14 + 1.65 + 1.04 + 1.01 + 0.14 = 7.98$.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 37.50      1.65     1.65                             __cos_fma
 23.64      2.69     1.04                             __ieee754_exp_fma
 22.95      3.70     1.01                             __sin_fma
  5.57      3.94     0.24                             main (simpleprofiling.c:17 @ 401819)
  4.55      4.14     0.20                             main (simpleprofiling.c:21 @ 401872)
  3.18      4.29     0.14                             expf64
  1.59      4.36     0.07                             main (simpleprofiling.c:20 @ 4018cc)
  0.45      4.38     0.02                             _init
  0.45      4.39     0.02                             main (simpleprofiling.c:20 @ 401869)
  0.11      4.40     0.01                             main (simpleprofiling.c:16 @ 401846)
  0.00      4.40     0.00        1     0.00     0.00  main (simpleprofiling.c:8 @ 4017d5)
```

Figure 4: Profiled stack

As a small note, what would happen if we did not use the static flag as requested by the exercise? Look no further than Figure 5! We lose the detailed information of time spent inside the functions.



```

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 47.37      0.27     0.27                             main (simpleprofiling.c:21 @ 1306)
 43.86      0.52     0.25                             main (simpleprofiling.c:17 @ 12ad)
  8.77      0.57     0.05                             main (simpleprofiling.c:20 @ 1360)
```

Figure 5: Profiled stack - no static flag