HPC Tools 2024, Exercise 3 solutions

Bruno Oliveira Cattelan, Laurent Chôné

Problem 1

In this exercise, we investigate the call stack using gdb. In order to produce a large stack, we use a program that calculate the Ackermann function recursively. In the implementation used, the recursive function is `ackermann(m,n)`$= A(m, n)$.

Let us compile the program in debug mode, run it in gdb and interrupt it with ctrl+c (alternatively ctrl+z)

```
$ g++ -g ex03pb01.cpp -o prog
$ gdb prog
...
(gdb) run
...
^C
Program received signal SIGINT, Interrupt.
0x00005555555551df in ackermann (m=1, n=14219) at ackermann.cpp:6
6           if ( m == 0)
```

We access the call stack with the command `where`:

```
(gdb) where
#0  0x00005555555551df in ackermann (m=1, n=14219) at ackermann.cpp:6
#1  0x0000555555555227 in ackermann (m=1, n=14220) at ackermann.cpp:19
#2  0x0000555555555227 in ackermann (m=1, n=14221) at ackermann.cpp:19
...
#18131 0x0000555555555227 in ackermann (m=3, n=13) at ackermann.cpp:19
#18132 0x0000555555555238 in ackermann (m=4, n=1) at ackermann.cpp:19
#18133 0x00005555555552b4 in main () at ackermann.cpp:38
```

We observe that the recursive function is called a very large number of times. Inspecting the stack is useful for instance to investigate the values of variables at different instants in the execution:

```
(gdb) frame 2
#2  0x0000555555555227 in ackermann (m=1, n=14221) at ackermann.cpp:19
19                  return ackermann(m - 1, ackermann(m, n-1));
(gdb) print m
$1 = 1
(gdb) print n
$2 = 14221
(gdb) frame 18131
#18131 0x0000555555555227 in ackermann (m=3, n=13) at ackermann.cpp:19
19                  return ackermann(m - 1, ackermann(m, n-1));
(gdb) print m
$3 = 3
(gdb) print n
$4 = 13
```

Problem 2

In this problem we are investigating the program provided with the profiler `gprof`.

To this end, we must first make sure to add the appropriate compiler flags in the `Makefile`:

```
CC= gcc -O2 -p -g -static
F90= gfortran -O2 -p -g -static
```

Remark: sometimes using `O2` optimisation can result in the same line appearing several time in the flat profile (with the total time being the sum). To avoid that problem, one can downgrade the optimisation to `O0`

a)

First we are interested in finding the line at which the program spends the most time during execution. We compile and run the program as instructed in the `README`, and profile as follows:

```
$ gprof -p -l mdmorse
```

Where the `-p` flag requests `gprof` only produces the flat profile, and the `-l` flag ensure that profiling is done line by line. We find the flat profile:

```
%   cumulative   self
time   seconds   seconds ...   name
10.78      5.67     5.67       GetForces (forces.c:88 @ 405331)
7.83       9.79     4.12       __ieee754_exp_fma
5.02      12.43     2.64       UpdateNeighbourlist (neighbourlist.c:60 @
   4041c5)
5.00      15.06     2.63       GetForces (forces.c:69 @ 405180)
4.37      17.36     2.30       GetForces (forces.c:86 @ 40530f)
...
```

And the line in question, where the program spends $\approx 11\%$ of the execution time:

```
88|  dVdr=(dVdr/(m*u))/aunit/r;
```

b)

Next we want to find in which function the program spends the most time. To do so, we merely remove the `-l` flag:

```
$ gprof -p mdmorse
...
%   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
59.55     31.33    31.33    30000     0.00     0.00  GetForces
23.65     43.77    12.44     6000     0.00     0.00  UpdateNeighbourlist
7.83      47.89     4.12                             __ieee754_exp_fma
3.82      49.90     2.01                             __ieee754_sqrt
2.17      51.04     1.14                             expf64
...
```

And so, we find that the program spends $\approx 60\%$ of the time in `GetForces`, which means that if we seek to optimise it, our efforts should be concentrated in this function.

Remark: While in this case the most expensive line is contained in the most expensive function, this is not necessarily the case. In fact you can notice that only $1/6$ of the execution time of `GetForces` is spent at line 88.

Problem 3

In this exercise we were provided a code for profiling. The problem itself asks for optimization levels of $0, 1, 2, 3$. For sake of completion, I also added level s. This optimization level focus on executable size instead of runtime performance. The results can be seen in Figure 2. The executions here were ordered using optimization 0 as the key since it was the largest execution time for all iterations.

As expected, we have level 0 as the slowest version. Level 3 was the fastest in all cases, whereas 1 and 2 despite following expectation have very similar runtimes. Finally, level s seems to behave similarly to level 2.

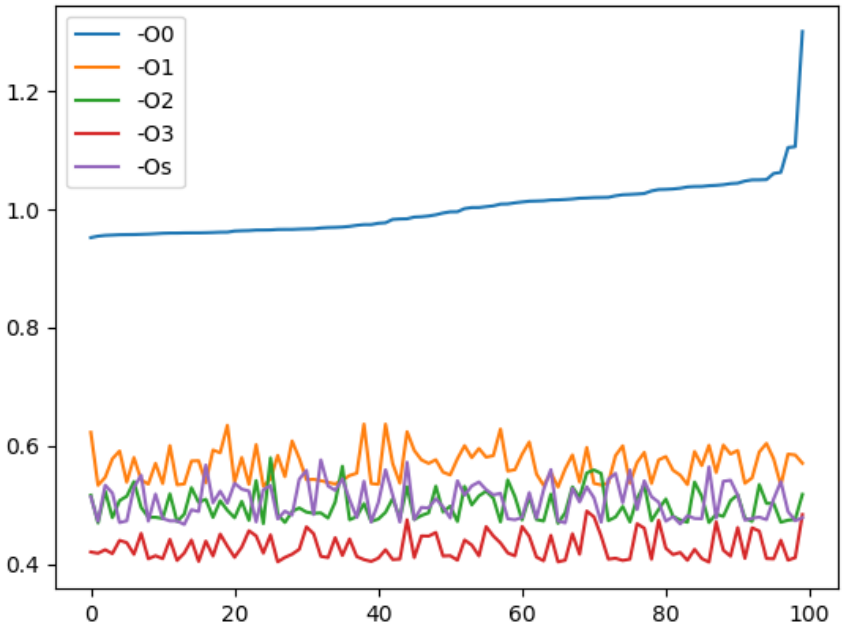

Figure 1: Runtimes of 100 executions with different optimizations

Problem 4

For problem 4 we were asked to create a custom code that takes a considerable amount of time to run. For this, I used the following code.

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <math.h>

#define N 10000
int a[N][N];

void main() {
  int i,j;
  double t1,t2;
  struct timeval time;
  double clocktime;

  gettimeofday(&time, NULL);
  t1=(double)time.tv_sec + (double)time.tv_usec/1000000.0;
  /* Begin measurement */
  for (i=0;i<N;i++)
    for (j=0;j<N;j++)
      a[i][j]=exp((i+j)/2);
  /* End measurement */
  gettimeofday(&time, NULL);
  t2=(double)time.tv_sec + (double)time.tv_usec/1000000.0;

  double sum = 0;
  for (i=0;i<N;i+=1) for (j=0;j<N;j+=1) sum+=a[i][j];
  printf("M: %f\n", sum);

  clocktime=t2 - t1;
  printf("%f\n",clocktime);

}
```

For sake of simplicity, I show only the clocktime version. For the cpu time, please ask me individually. In any case, the main loop here computes this quite large matrix using its indexes and exponential function. The matrix is then summed and printed, which is not counted for runtime. I mark the matrix output with a "M:" which I use to later remove from the output files. This allows me to then use the output times in a csv format and plot the figure shown next.

We see that is a rather large discrepancy between these two times. CPU clock measures the time spent running in the CPU, whereas Wall clock measures the real time spent in the code from the user point of view. One example that can show the difference is for I/0 heavy programs. In our case the distance between the lines is not too large. Still, this difference in runtimes could potentially be explained by other programs with higher priority being executed first and having our code put to sleep. There is also a high variance on both cases. This can be explained by me using my computer for other purposes as the code ran. An example of CPU time variance cause could be cache use or memory bandwidth competing with other programs.

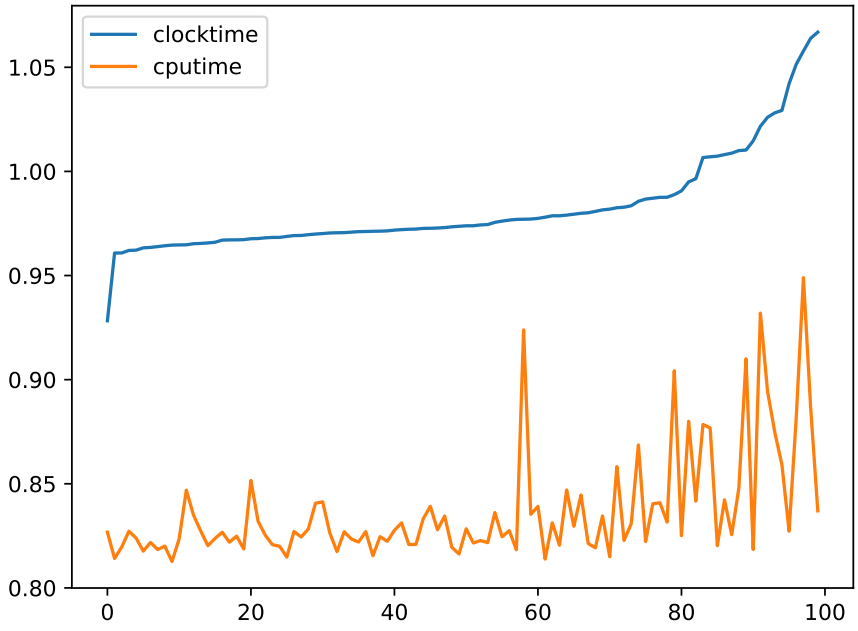

Figure 2: Clocktime and cputime of 100 executions