A2: Profiling Tools

Francesca Cairoli

The assignment requires to perform a basic code profiling by means of valgrind, perf and gprof. The chosen code is the implementation of the **QuickSort** algorithm. QuickSort applies a well-known divide and conquer technique. It picks an element, called pivot, which divides the array into two halves in such a way that elements in the left half are smaller than pivot and elements in the right are greater than pivot. It recursively perform three steps: 1. it brings the pivot to its appropriate position; 2. it quick sort the left part; 3. it quick sort the right part. In our version the pivot is selected as the last element. The key process in `quicksort.c` is `partition()`. Target of partitions is, given an array and an element x of array as pivot, put x at its correct position and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element. The code `quicksort.c`, provided in the Appendix, was compiled as follows:

```
gcc -pg quicksort.c -o qs
```

We considered an array of $100,000$ random integer numbers. The program's execution takes around 0.022 seconds on my personal laptop.

Valgrind

We start by using the Valgrind framework to spot memory errors. Memory errors may consists in accessing already freed memory locations, in memory leaks, i.e., memory allocated and not freed before the program termination or, more in general, in an incorrect freeing of the heap. More precisely, the command below runs the **Memcheck** tool.

```
$ valgrind ./qs
```

From the obtained results, shown below, we may conclude that the Memcheck tool did not find any error in the management of the memory. This is not surprising as our application is not performing any explicit dynamic allocation of the memory.

Memcheck results:

```
==11147== Memcheck, a memory error detector
==11147== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward
    et al.
```

```
==11147== Using Valgrind-3.13.0 and LibVEX; rerun with -h for
    copyright info
==11147== Command: ./qs
==11147==
==11147== HEAP SUMMARY:
==11147==     in use at exit: 0 bytes in 0 blocks
==11147==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==11147==
==11147== All heap blocks were freed -- no leaks are possible
==11147==
==11147== For counts of detected and suppressed errors, rerun with:
     -v
==11147== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
    from 0)
```

The execution time, with valgrind, is 0.745 seconds. The Valgrind framework, present another tool, called **Callgrind**. The tool is launched with the command below, where we did set some arguments in order to make valgrid display the desired informations:

```
$ valgrind --tool=callgrind --dump-instr=yes --cache-sim=yes --
    branch-sim=yes --collect-jumps=yes ./qs
```

The output is the following:

```
==11157== Callgrind, a call-graph generating cache profiler
==11157== Copyright (C) 2002-2017, and GNU GPL'd, by Josef
    Weidendorfer et al.
==11157== Using Valgrind-3.13.0 and LibVEX; rerun with -h for
    copyright info
==11288== I   refs:      130,662,331
==11288== I1  misses:            958
==11288== LLi misses:            947
==11288== I1  miss rate:        0.00%
==11288== LLi miss rate:        0.00%
==11288==
==11288== D   refs:       83,437,118  (54,459,537 rd + 28,977,581
    wr)
==11288== D1  misses:         58,716  (    51,438 rd +      7,278
    wr)
==11288== LLd misses:          8,910  (     2,091 rd +      6,819
    wr)
==11288== D1  miss rate:         0.1% (       0.1%   +        0.0%
     )
==11288== LLd miss rate:         0.0% (       0.0%   +        0.0%
     )
==11288==
==11288== LL refs:            59,674  (    52,396 rd +      7,278
    wr)
==11288== LL misses:           9,857  (     3,038 rd +      6,819
    wr)
==11288== LL miss rate:          0.0% (       0.0%   +        0.0%
     )
==11288==
==11288== Branches:       11,669,289  (10,323,019 cond +  1,346,270
     ind)
==11288== Mispredicts:       780,324  (   780,176 cond +        148
     ind)
```

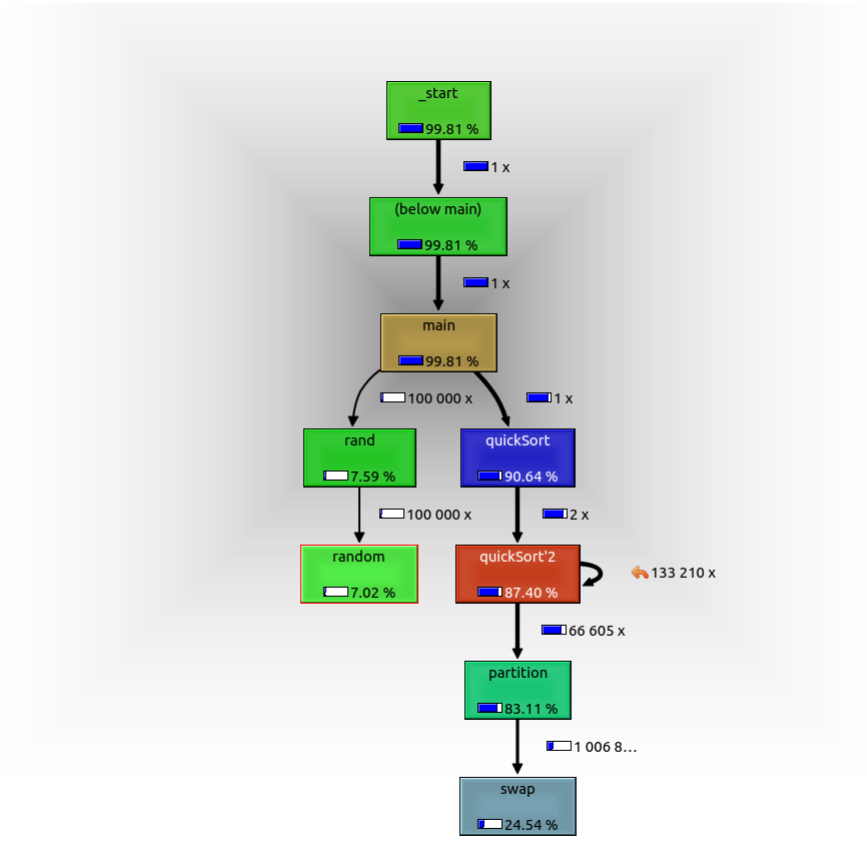

Figure 1: Call graph generated by the callgrind tool.

```
22 ==11288== Mispred rate:          6.7% (        7.6%      +
       0.0%    )
```

The callgrind tool creates a file to store the results, called `callgrind.out.XXX` where `XXX` is the process identifier. We use KCacheGrind to open this file. It is indeed a visualization tool for profiler-generated files. From KCacheGrind we are able to export the call graph of our program. The call graph is shown in Figure 1. It graphically represents the workflow of the program. The arrows make it easy to understand the dependencies inside the code and also the number of times a function is called by its parent function. Since the function `quickSort` is recursive, the graph presents an additional cell, `quickSort'2`, representing the second level of recursion.

In the following sections, the executable is called `qs_perf` but compilation settings are the same.

Perf

We now use the **perf** tool to profile the core events during the execution of our program. In particular, from the `perf list`, we choose to monitor the number of cycles, the number of instructions, the number of cache misses, the number of branches and finally the number of branch misses.

```
$ perf stat -e cpu-cycles:u,instructions:u,cache-misses:u,branches:u,branch-misses:u ./qs

Performance counter stats for './qs_perf':
86.849.577      cpu-cycles:u
130.646.219      instructions:u     #    1,50  insn per cycle
2.300      cache-misses:u
19.809.254      branches:u
680.503      branch-misses:u        #    3,44% of all branches
    0,036426911 seconds time elapsed
```

We observe that the number of cache and branch misses is considerably lower than the total number of cycles and branches. In fact, perf measure a rate of 0.00176% for cache misses and a rate of 3.44% for branch misses. These results are comparable with the ones shown above, obtained using callgrind. We also tried to run the same command on a Ulysses's node, and we obtained a even smaller (almost negligible) number of observed cache misses. We also tried to compile `quicksort.c` with various optimization flags (from `-O1` to `-O3`). Interestingly the rate of branch-misses increases togheter with the level of optimization: from 7.4% with `-O1` to 12.26% with `-O3`. The cache-misses rate remains unchanged. This result tells us that an aggressive optimization at compilation time is not helpful in branch predictions. Therefore, it is better not to add any optimization flag. Finally, this results guarantee that the code is not wasting resources in an inefficient use of the memory. Hence, its execution is bounded by CPU performances.

The execution of the command above took only 0.013 seconds. A comparison of the execution times of perf and valgrind shows an interesting advantage of the perf tool. In fact, the execution time does not increase as much as in the valgrind framework.

```
$perf report --call-graph graph,5,100,caller

Samples: 34  of event 'cycles:ppp', Event count (approx.):
    126845375
  Children      Self  Command  Shared Object      Symbol
+   93,87%     5,74%  qs_perf  qs_perf            [.] quickSort
+   91,65%    40,21%  qs_perf  qs_perf            [.] partition
+   48,19%    11,01%  qs_perf  qs_perf            [.] swap
+   39,47%    10,29%  qs_perf  libc-2.27.so       [.] _mcount
+   26,62%    26,62%  qs_perf  libc-2.27.so       [.]
    __mcount_internal
+    9,36%     0,00%  qs_perf  qs_perf            [.] _start
+    9,36%     0,00%  qs_perf  libc-2.27.so       [.]
    __libc_start_main
+    9,36%     0,00%  qs_perf  qs_perf            [.] main
+    5,82%     5,82%  qs_perf  ld-2.27.so         [.]
    _dl_load_cache_lookup
```

```
+    5,82%     0,00%  qs_perf  ld-2.27.so          [.] _start
+    5,82%     0,00%  qs_perf  ld-2.27.so          [.] _dl_start
+    5,82%     0,00%  qs_perf  ld-2.27.so          [.]
    _dl_sysdep_start
+    5,82%     0,00%  qs_perf  ld-2.27.so          [.] dl_main
+    5,82%     0,00%  qs_perf  ld-2.27.so          [.]
    _dl_map_object_deps
+    5,82%     0,00%  qs_perf  ld-2.27.so          [.]
    _dl_catch_exception
+    5,82%     0,00%  qs_perf  ld-2.27.so          [.] openaux
+    5,82%     0,00%  qs_perf  ld-2.27.so          [.]
    _dl_map_object
```

Gprof

Let's now analyse **gprof** results.

```
$gprof ./qs_perf
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 50.16      0.01     0.01  1046089     0.00     0.00  swap
 50.16      0.02     0.01    66606     0.00     0.00  partition
  0.00      0.02     0.00        1     0.00    20.06  quickSort
```

The table above shows the flat profile of our application. The first column, *% time*, indicates the percentage of the total running time of the program used by each function. The column *cumulative seconds* indicates a running sum of the number of seconds accounted for the indicated function and the functions above. The column *self seconds* indicates the number of seconds accounted for by each function alone (this values are used to sort the table). The column *calls* indicates the number of times each function was invoked. The *total ms/cal* column indicates the average number of milliseconds spent in each function and its descendents per call. The flat profile shows that the program spends half of its running time in the `swap` function and the other half in the `partition` function. This is not surprising, as these two functions represent the core part of the algorithm. The `swap` function is called 1046089 times throughout the entire execution, so it is clear how it plays a key role in the program. The time spent in the recursive `quickSort` function is instead almost negligible.

Finally, we used **gprof** to generate another call graph. We seek to compare the latter with the one generated by callgrind (Figure 1).

```
index % time    self  children    called     name
                0.01    0.01   66606/66606       quickSort [2]
[1]     100.0    0.01    0.01   66606         partition [1]
                0.01    0.00 1046089/1046089     swap [4]
-----------------------------------------------
                              133212              quickSort [2]
                0.00    0.02       1/1           main [3]
[2]     100.0    0.00    0.02       1+133212  quickSort [2]
                0.01    0.01   66606/66606       partition [1]
                              133212              quickSort [2]
-----------------------------------------------
                                                  <spontaneous>
```

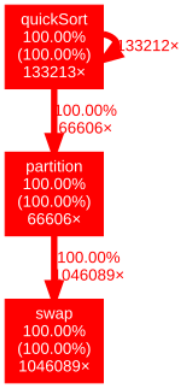

Figure 2: Graphical representation of the call graph generated by gprof.

```
[3]     100.0    0.00    0.02                  main [3]
                 0.00    0.02       1/1            quickSort [2]
-----------------------------------------------
                 0.01    0.00 1046089/1046089      partition [1]
[4]      50.0    0.01    0.00 1046089          swap [4]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function and the lines below it list the functions this one called. The called values indicates the number of times the function was called. If the function called itself recursively, as for the quickSort method, the number only includes non-recursive calls and is followed by a '+' and the number of recursive calls.

Figure 2 present a graphical interpretation of the call graph shown in the table above. It was generate using the following command:

```
$ gprof ./qs_perf | gprof2dot | dot -Tsvg > qs_callgraph_gprof.svg
```

Appendix

Implementation, in C language, of the QuickSort algorithm (file `quicksort.c`.

```c
/* C implementation QuickSort */
#include<stdio.h>
#include<stdlib.h>

// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
```

```
}

/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];    // pivot
    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
 arr[] --> Array to be sorted,
  low  --> Starting index,
  high  --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Driver program to test above functions
int main()
{
    int N = 100000;
    int arr[N];
    for(int f = 0;f<N;f++){
        arr[f] = (int) (rand());
   }
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);

    return 0;
```

```
68 }
```