Code Profiling

Gabriele Sarti

December 7, 2018

Abstract

The purpose of this assignment is to perform a basic code profiling of a piece of code at choice. In my case, I will inspect an OpenMP implementation of the Dijkstra's algorithm for finding the shortest path between nodes in a graph using different profiling tools.

1 Introduction

In order to profile the execution of the code contained in dijkstra_omp.c we will use all the tools that were presented to us during the profiling lesson to compare their results and their effectiveness:

- First we will try the Valgrind framework, and more specifically the tools Memcheck, used to detect memory errors, and Callgrind, a profiling tool that records the call history among functions in a program's run as a call-graph [1].
- After that, we will compare the results obtained with the ones produced by gprof, a performance analysis tool for Unix applications [2], and perf, a performance analysis tool included in Linux kernel [3].
- Lastly, we will try the Google Performance Tools open-source framework in order to seek for additional insights. [4]

About the code we are profiling, Dijkstra's algorithm is probably the most famous piece of code used for shortest-path finding. Our version represents a variant in which a shortest-path tree is produced from the minimal distances between our node 0 and all the other nodes in the graph, using the multithreading library OpenMP in order to speed up the process.

For profiling purposes, we will create a graph of 1000 nodes and 3000 connections between edges, all of which are initialized with random weight values between 0 and 100. A typical run with this problem size takes around 1.3 seconds to finish, excluding I/O.

2 Profiling Dijkstra's Algorithm

2.1 Profiling with Valgrind

Before performing any test, we compiled the code linking to the OpenMP library in order to enable multithreading for our application and adding debug information which are necessary for a complete profiling.

```
gcc -g -fopenmp dijkstra_omp.c -o dijkstra_omp
```

2.1.1 Finding memory errors with Memcheck

We run the Memcheck tool and obtained the following results:

```
> valgrind ./dijkstra_omp

==6982== Memcheck, a memory error detector
==6982== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6982== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==6982== Command: ./dijkstra_omp

OpenMP Results for Graph:
Path to Vertex 0 is 0
...
Path to Vertex 999 is 209
Running time: 39885.462309 ms

==6982== HEAP SUMMARY:
==6982==     in use at exit: 3,312 bytes in 7 blocks
==6982==   total heap usage: 9 allocs, 2 frees, 37,152 bytes allocated
==6982==
==6982== LEAK SUMMARY:
==6982==    definitely lost: 0 bytes in 0 blocks
==6982==    indirectly lost: 0 bytes in 0 blocks
==6982==      possibly lost: 864 bytes in 3 blocks
==6982==    still reachable: 2,448 bytes in 4 blocks
==6982==         suppressed: 0 bytes in 0 blocks
==6982== Rerun with --leak-check=full to see details of leaked memory
==6982==
==6982== For counts of detected and suppressed errors, rerun with: -v
==6982== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

It is immediately evident how Memcheck didn't find any errors concerning memory management. This is quite normal in this context, given that our application is not performing any explicit dynamic allocation. Another interesting thing to observe is that execution was forty times slower than the original one, which is normal according to the Valgrind manual [5].

2.1.2 Generating the call graph

We used the following command in order to run the Callgrind tool on our application. The arguments used ensure that event counting is performed at source line granularity, that cache and branch prediction simulation are performed fully and that information on jumps inside code is collected.

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

We can immediately note that almost the entire number of cycles inside the program are spent for the findEdge routine, which is used to match nodes with edges that rely them to other nodes. Figure 1 shows the loop inside the function, where we verify for each edge in the graph if it contains the current node or not.

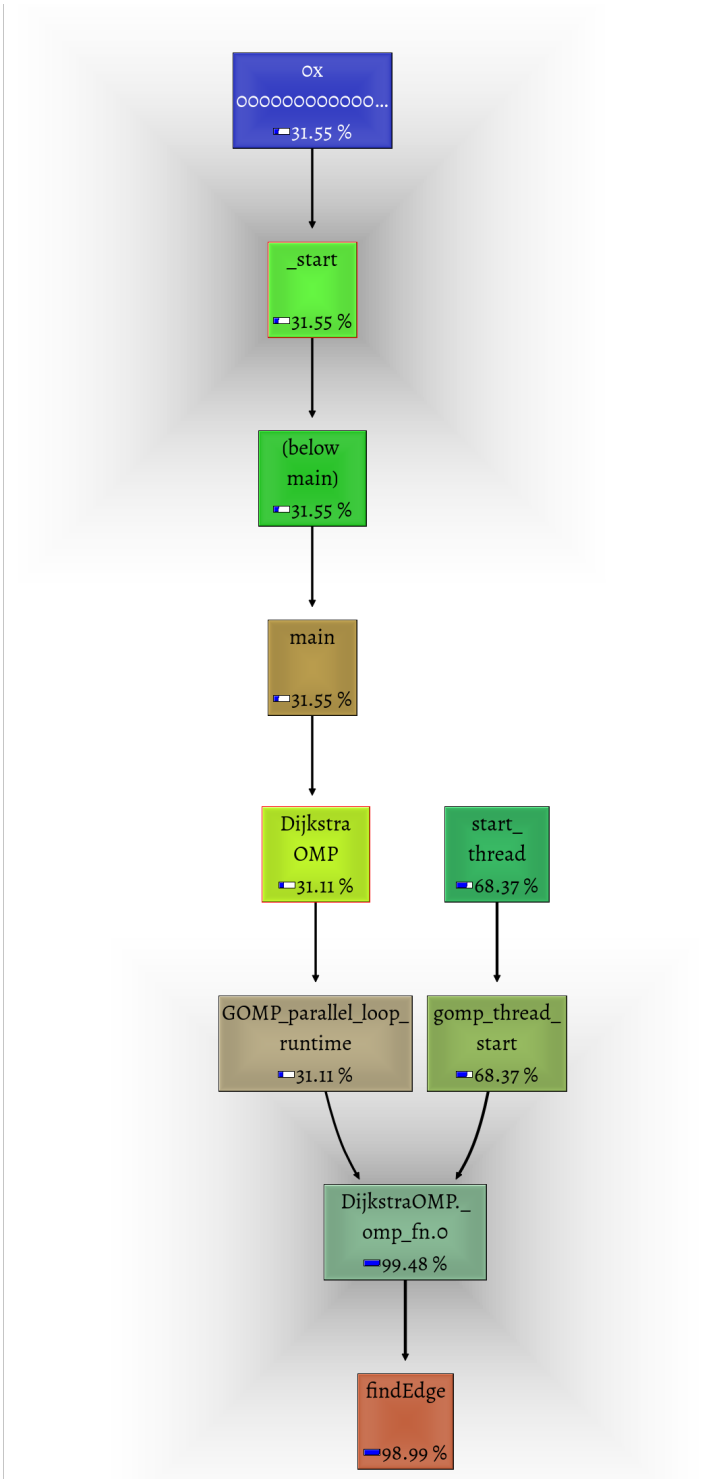
Figure 1: Source code for the findEdge function, for indication of total number of cycles line-by-line.

```
40                  //Finds the edge that connects Vertex u with Vertex v
41                  int findEdge(Vertex u, Vertex v, Edge *edges, int *weights)
42    0.02          {
43                      int i;
44   24.77              for(i = 0; i < E; i++)
                    Jump 1 496 179 831 of 1 496 677 800 times to dijkstra_omp.c:46
45                      {
46   74.39                  if(edges[i].u == u.title && edges[i].v == v.title)
                    Jump 1 496 178 300 of 1 497 694 661 times to dijkstra_omp.c:44
47                          {
48    0.00                      return weights[i];
                    Jump 1 531 times to dijkstra_omp.c:52
49                          }
50                      }
51    0.00              return INT_MAX;
52    0.01          }
```

It is evident how most cycles inside the application are wasted inside this loop, especially with high numbers of nodes. If we define V and E as respectively the number of vertex (nodes) and the edges in the graph, we could be executing the content of this loop $V \times V \times E$ times in the worst case scenario. From Figure 1 annotation, we see the real number of cycles is roughly 1.5 billions, around half of our original maximal estimate. It is interesting to note that the most relevant function in terms of L1 cache misses is the DijkstraOMP function, which perform a lot of subsequent operations inside unoptimized loops.

We then opened the dump file generated by callgrind with Kcachegrind, a visualization tool for profiler-generated files, and exported the call graph for our application. The graph is presented in Figure 2 and also available at full-size in the attachments, along with the callgrind output itself.

Figure 2: Callgraph for dijkstra_omp created with the Callgrind tool with informations on total cycles partition between functions.



2.2 Profiling with Perf and Gprof

First, we run a simple profiling of the core events, namely number of cycles, number of instruction, cache misses, number of branches and branch misses, using the perf tool.

```
> perf stat -e cpu-cycles:u,instructions:u,cache-misses:u,branches:u,branch-
    misses:u ./dijkstra_omp
[...]
Running time: 1269.744131 ms

 Performance counter stats for './dijkstra_omp':

    14,654,067,543      cycles:u
    18,147,998,178      instructions:u            #    1.24  insn per cycle
            13,286      cache-misses:u
     3,032,781,830      branches:u
         2,826,840      branch-misses:u

       1.334322793 seconds time elapsed
       5.069145000 seconds user
       0.003331000 seconds sys
```

We can see that the total number of cache and branch misses is quite low compared to the total number of cycles and branches, which is a good indicator that our application is not wasting resources. The fact that the number of instructions per cycle is above 1 indicates that the application is CPU-bound, so it could use more processing power to run faster. It is also interesting to note how perf doesn't slow down our application as Valgrind was doing before.

After this preliminary analysis, we created callgraphs using both perf and gprof, in order to compare them with the one obtained with Valgrind. It is important to note that, in order to generate the gprof graph successfully, the code should be recompiled with the -pg flags and the new executable should be run in order to generate the gmon.out file.

```
# Records a run of our application with perf
> perf record -F1000 --call-graph lbr ./dijkstra_omp
# Creates a perf.data file with informations for our run
> perf report --call-graph graph,5,100,caller
# Generates the perf callgraph in svg format
> perf script | gprof2dot --format=perf | dot -Tsvg > callgraph_perf.svg
# Generates the gprof callgraph in svg format
> gprof ./dijkstra_omp | gprof2dot | dot -Tsvg > callgraph_gprof.svg
```

From the callgraphs presented in Figure 3 and 4, which are also available in the attachments both in full-size SVG and PNG formats, one can note that the output generated is very similar to the one of Callgrind with a single noticeable difference, namely the fact that Gprof callgraph doesn't take OpenMP functions in account. This doesn't have a noticeable effect on cycle share between functions, since findEdge still accounts for almost all the total number of cycles, confirming the profiling results obtained through Callgrind.

Figure 3: Callgraph for dijkstra_omp generated with the Perf tool with informations on total cycles partition between functions.

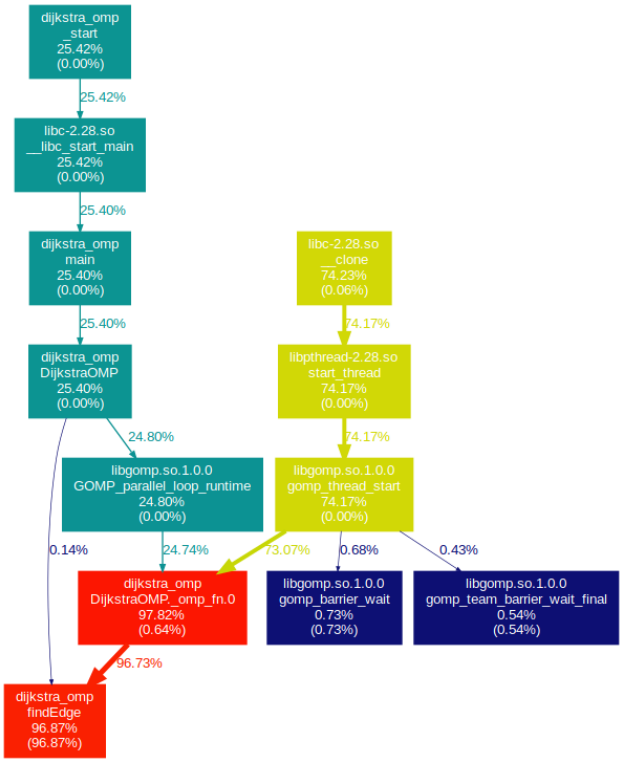
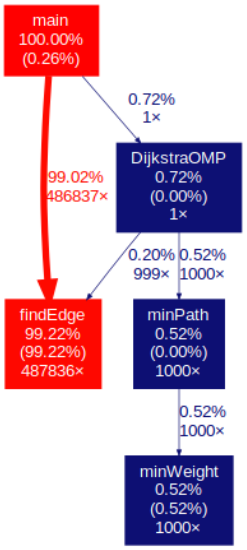


Figure 4: Callgraph for dijkstra_omp generated with the Gprof tool with informations on total cycles partition between functions and total instructions estimates.

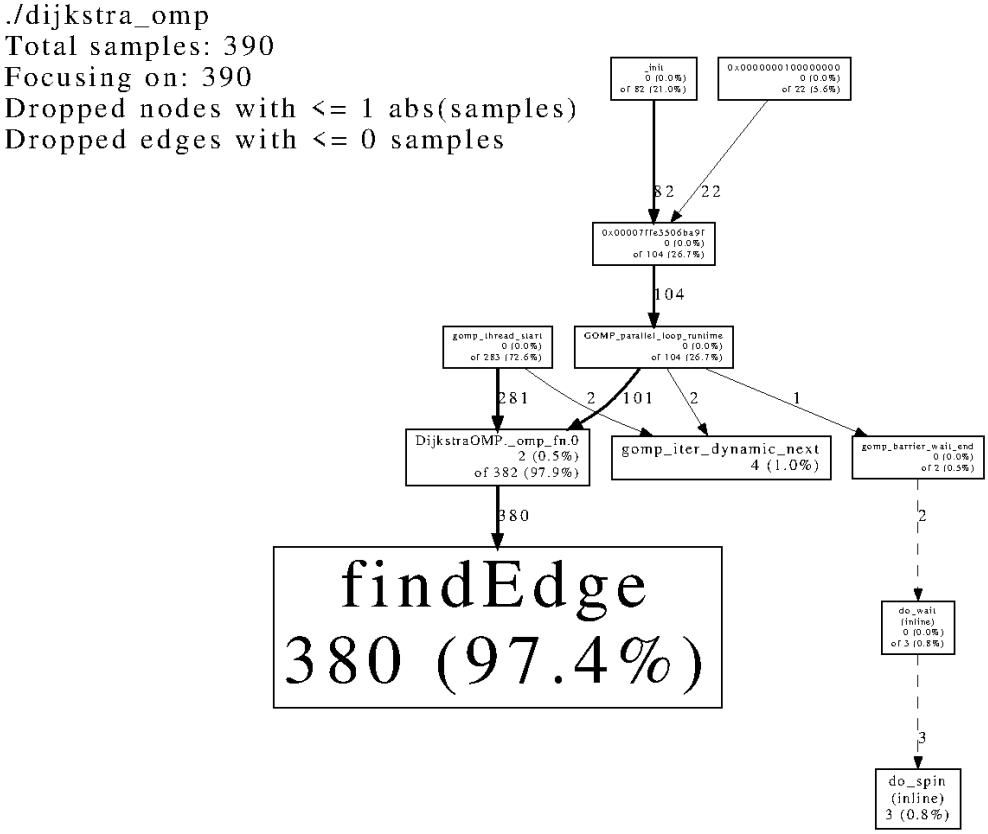


2.3 Profiling with GPerfTools

For the sake of completeness, we also performed a general profiling of our code using the GPerfTools suite powered by Google, generating yet again a callgraph which is presented in Figure 5 and also available in attachments.

```
# Compile again the code, linking it against the gperftools profiler
> gcc -fopenmp -DWITHGPERFTOOLS -lprofiler dijkstra_omp.c -o dijkstra_omp
# Set the variable as the output name in order to avoid changing the code
> env CPUPROFILE=dijkstra.prof ./dijkstra_omp
# Generates the Gperftools callgraph
> pprof --gv ./dijkstra_omp dijkstra.prof
```

Figure 5: Callgraph for dijkstra_omp generated with the Gperftools using a sampling method, where each sample corresponds to roughly 10 milliseconds of execution time.



The two main advantages of using Gperftools in generating the callgraph are that the function are scaled with their weight in term of samples, which makes the bottlenecks immediately evident as in our case with findEdge, and that using sampling based on a temporal estimate we can obtain an indication of execution times for each function [6]. For example, since we know that findEdge takes 380 samples and each sample is taken roughly each 10 ms, the function takes 3.80 ± 0.001 seconds to execute, on a total execution time of 3.90 seconds. We can also note that profiling with gperftools slows down the execution by a factor of 3.

3 Conclusions

Our final take-aways from this assignment, concerning both the analyzed code and the tools we used to profile it:

- Our application is probably CPU-bound, and it is not wasting resources through cache and branch misses in a noticeable way. This indicates the best solution to make our code more fast is to run it on more capable systems. However, if we really seek to perform any improvement on it, some loop optimization techniques can be tried inside the findEdge function.
- Each tools used proved to have its strengths and weaknesses, but the final results we obtained were very similar in each case, so one can probably use the one that fits better the purposes of the profiling task to be performed.
- It is also worth mentioning that we discovered OpenMP methods to be very efficients in term of cycles, since they always took a minimal fraction of the total number of cycles used by the application.

References

[1] `http://valgrind.org/docs/manual/cl-manual.html`

[2] `https://en.wikipedia.org/wiki/Gprof`

[3] `https://en.wikipedia.org/wiki/perf_(Linux)`

[4] `https://github.com/gperftools/gperftools`

[5] Memcheck adds code to check every memory access and every value computed, making it run 10-50 times slower than natively.

[6] `https://gperftools.github.io/gperftools/cpuprofile.html`, Section "Node information"