Profiling

Rabindra Khadka

DSSC, UNITS

Overview:

Profiling helps to optimize the code by measuring space and time complexity, the frequency of function call, the time spent by functions and other optimizing parameters. In this lab exercise, a program was chosen to be investigated using three different profiling tools namely valgrind/callgrind, gprof and gperftools. The output obtained through different profiling tools are discussed under relevant section below:

Code:

CITIES is a code chosen for profiling in this instance which solves the problems involving intercity distances. Some of the problems tackled by the code are travelling salesman problem, k-means calculations, to minimize the total distance from each city to its nearest particular city, weighted K-means or K-medians, minimal spanning trees which constructs shortest highway system that connects all the cities, voronoi diagrams which assign each spot of land to the nearest city. The entire source code is appended in the appendix.

Gprof:

Gprof is a Unix tool for performance analysis which uses hybrid approach between instrumentation and sampling. Instrumentation method accumulates the total function call made and sampling method gathers profiling information. Sampling data is output and saved in gmon.out and can be analysed with the gprof command line.

Gprof outputs flat profile and the call graph. The information regarding the total execution time spent in each function and its callees can be read by looking at flat profile. It also gives how many times a function ran which provides us the hotspot of the code. The call graph depicts clearly which functions called (parent) to which function (child).

One of the drawback of gprof is that it cannot profile shared libraries.

 # generating Object file; -c does not invoke the linker

g++ -c -pg cities.cpp

g++ -c -pg cities_prob.cpp

 # linking and compiling with the profiling support (-pg)

g++ -pg cities.o cities_prb.o -o cities.x

 # printing the callgraph

gprof cities.x

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 28.57      0.02     0.02   194688     0.00     0.00  dms_to_radians(int*)
 28.57      0.04     0.02        3     6.67     6.67  r8mat_write(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char
> >, int, int, double*)
 28.57      0.06     0.02        1    20.00    39.94  dms_to_dist(int, int*, int*)
 14.29      0.07     0.01        2     5.00     5.00  r8mat_nint(int, int, double*)
  0.00      0.07     0.00   194688     0.00     0.00  i4_sign(int)
  0.00      0.07     0.00   103861     0.00     0.00  std::setw(int)
  0.00      0.07     0.00    98452     0.00     0.00  std::setprecision(int)
  0.00      0.07     0.00    97828     0.00     0.00  r8_abs(double)
  0.00      0.07     0.00    48516     0.00     0.00  dms_to_distance_earth(int*, int*, int*, int*)
  0.00      0.07     0.00     4992     0.00     0.00  i4_huge()
  0.00      0.07     0.00     1176     0.00     0.00  ch_eqi(char, char)
  0.00      0.07     0.00      920     0.00     0.00  s_len_trim(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
 >)
```

Fig 1: Flat profile obtained from running gprof profiling tool.

The callgraph results shows that how much time the program spent in each function and how many times the function was called. This also holds the information how much time was spend by the children and the parent function.

Valgrind:

Valgrind is an analytical tool for programming which aids in detecting memory leaks, memory debugging and profiling. It does not execute directly the code but just simulates the on the fly generated UCode. There are numerous valgrind tools but for this exercise callgrind, memcheck and cachegrind are used.

KCachegrind is used to visualize the collected data of the program run. It is simple to use and produces greatly overview of the program calls. The following steps were followed to create the profile of our chosen code.

1. Compiled the code with –pg

2. Executed with valgrind –tool=callgrind. /city.x

3. Opened the profiling file with KCachegrind

4. /usr/bin/time valgrind --track-origin=yes ./cities.x (to see the time)

   kcachegrind callgrind.out.5039

While comparing with gprof, valgrind is simple to implement and no special compiling flags were required but one of the drawbacks of using valgrind is it adds more overhead but with better accuracy.

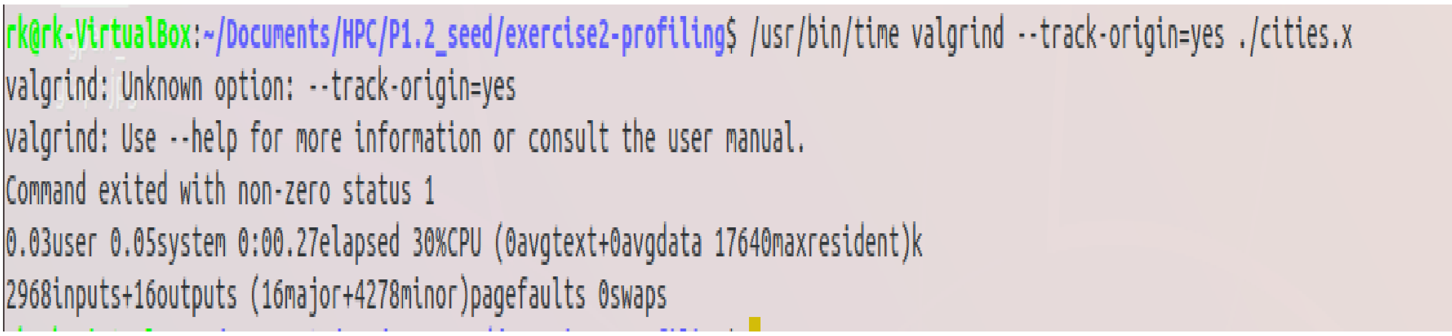

Fig2: Shows the time captured for running valgrind tool which stands 27 sec elapsed time.

The kcachegrind graph has been attached below:

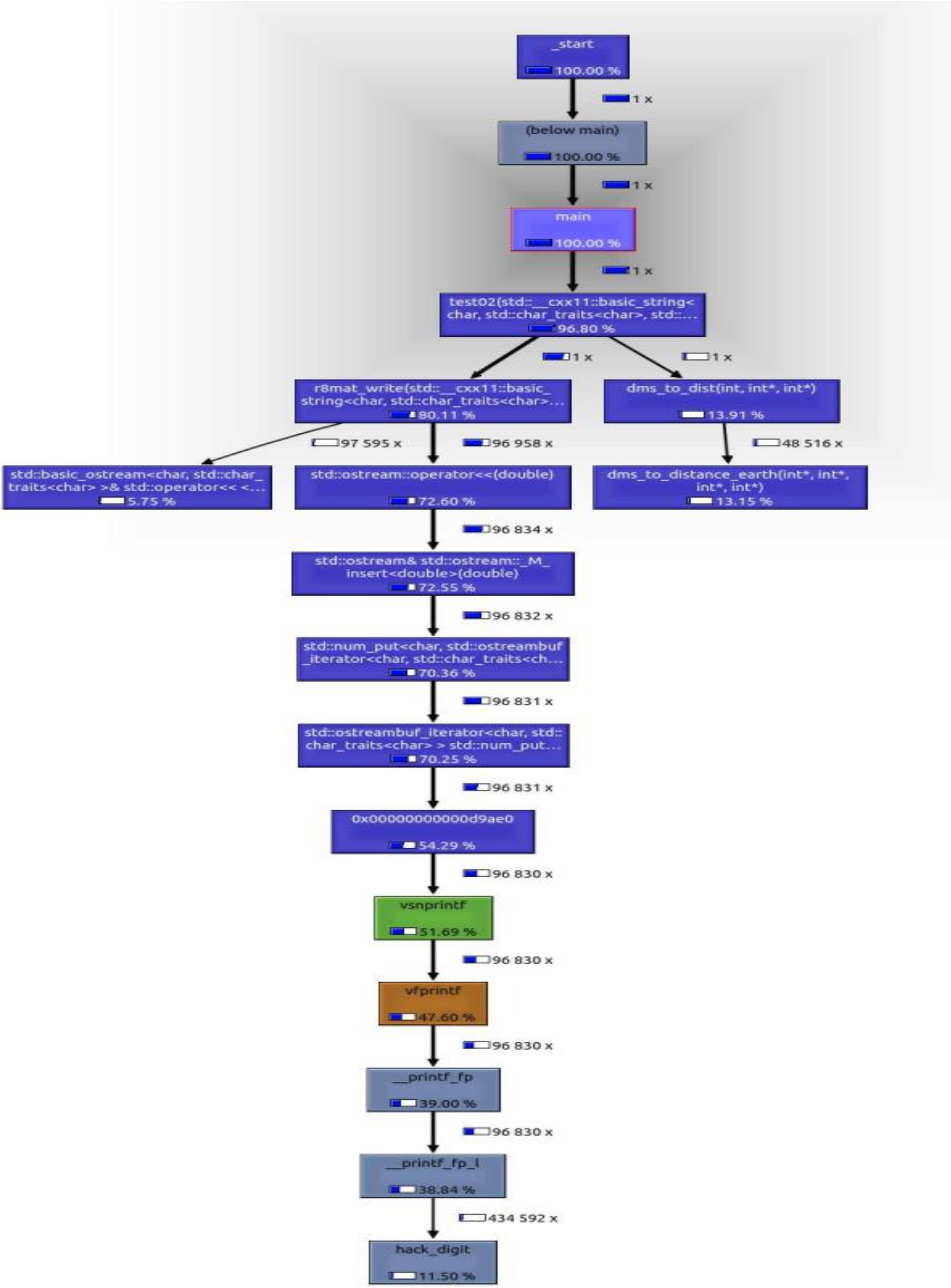

Fig 3: Callgrpah of the application code extracted with kcachegrind

Incl.	Self	Called	Function	Location
71.48	12.78	99 148	std::ostreambuf_iterato...	libstdc++.so.
2.97	7.73	194 688	dms_to_radians(int*)	city.x
4.99	7.16	197 550	std::basic_filebuf<char, ...	libstdc++.so.
2.60	6.09	197 238	std::basic_streambuf<c...	libstdc++.so.
2.45	5.87	99 208	__cxxabiv1::__vmi_class...	libstdc++.so.
5.44	5.05	108 056	std::basic_ostream<cha...	libstdc++.so.
55.24	4.64	99 148	0x00000000000d9ae0	libstdc++.so.
3.89	4.56	99 378	__dynamic_cast	libstdc++.so.
73.82	4.55	99 148	std::ostream& std::ostre...	libstdc++.so.
1.70	4.23	2	r8mat_nint(int, int, dou...	city.x
1.31	4.11	211 927	std::ostream::sentry::se...	libstdc++.so.
1.95	3.83	103 437	0x00000000001077f0	libstdc++.so.
80.83	3.48	3	r8mat_write(std::__cxx...	city.x
13.07	3.13	48 516	dms_to_distance_earth...	city.x
0.89	2.80	99 148	std::__num_base::_S_fo...	libstdc++.so.

Fig 2: Statistics of different functions that were called.

The figure 2 shows the sorted list of functions with descending order of list of functions with the function with the highest cost at the top. In the chosen program, the 'std: ostream buf' has the highest cost and spends 12.78 sec.

Gperftools:

Gperftools is provide by Google which aids in performance profiling and memory checking. It is a simple tool with low overhead and graphical output too. It provides option to profile the whole process runtime or to profile only a part of the runtime. During this lab exercise we profile the application for the entire runtime and focused on cpu profiling.

In the code, ProfilerStart() and ProfilerStop() were inserted. (These functions are declared in <gperftools/profiler.h>.ProfilerStart () will take the profile-filename as an argument.

The following steps were followed for using gperftools:

1. Program was compiled using g++ -g –lprofiler

2. **CPUPROFILE** environment variable set to the name of the file to store the profile result.

CPUPROFILE=gperftool.prof ./mycity.x

3. Using pprof to convert output into cachegrind format.

google- pprof –callgrind ./mycity.x gperftool.prof > gperftool.callgrind

4. Kcachegrind gperftool.out (displays the callgraph)

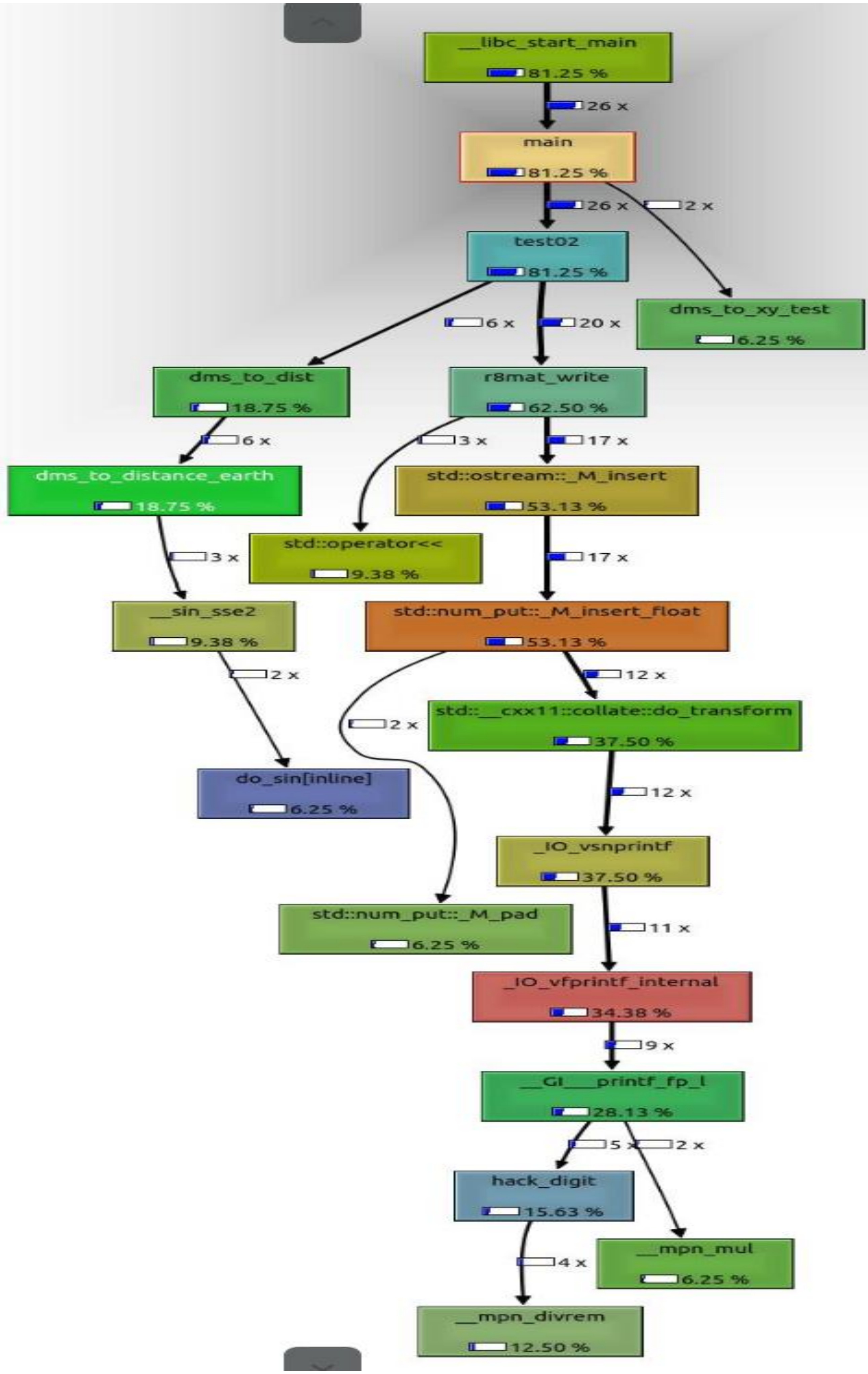

Fig5: Call graph resulted from google perftools***.***

main

Incl.	Self	Distance	Calling	Callee
15.38	15.38	10	4	__mpn_divrem (divrem.c)
34.62	7.69	8	9	__GI___printf_fp_l (printf_fp.c)
42.31	7.69	7	11	_IO_vfprintf_internal (vfprintf.c)
7.69	7.69	5	2	do_sin[inline] (s_sin.c)
65.38	7.69	4	17	std::num_put::_M_insert_float (??)
3.85	3.85	10	1	__mpn_mul_1 (mul_1.S)
19.23	3.85	9	5	hack_digit (printf_fp.c)
7.69	3.85	9	2	__mpn_mul (mul.c)
3.85	3.85	8	1	__strchrnul_sse2 (strchr.S)
3.85	3.85	6	1	std::locale::id::_M_id (??)
7.69	3.85	5	2	std::num_put::_M_pad (??)
3.85	3.85	5	1	std::locale::facet::_S_get_c_locale (??)
11.54	3.85	4	3	__sin_sse2 (s_sin.c)
3.85	3.85	4	1	dms_to_radians (cities.cpp)
3.85	3.85	4	1	__ieee754_acos_sse2 (e_asin.c)
3.85	3.85	4	1	__cos_sse2 (s_sin.c)
3.85	0.00	7	1	__find_specmb (printf-parse.h)
46.15	0.00	6	12	_IO_vsnprintf (vsnprintf.c)
46.15	0.00	5	12	std::__cxx11::collate::do_transform (??)
65.38	0.00	3	17	std::ostream::_M_insert (??)
23.08	0.00	3	6	dms_to_distance_earth (cities.cpp)
19.23	0.00	3-4 (3/0)	5	std::operator<<
3.85	0.00	3	1	r8mat_transpose_print_some (cities.cpp)

Fig 6: Obtained from Gperf tools which shows all callees

The above figure shows different callees of our cities.cpp. It shows that the function '_mpn_diverm' created the hotspot with 15.38 sec spent inside the function.

Summary:

Gprof with more runtime overhead and requirement of special flags and compatibility issue with profiler is not so popular profiling tool now for today's large projects.

Valgrind is quicker than gprof and gives more accurate performance profiling. Kcachegrind gives a well-placed graph for visualisation but however it can have large overhead for longer application.

Gperftools from google has very small overhead in compare to the other profiling tools so this can be suitable tool for large applications.