|epcc|

Performance Analysis using Google Performance Tools

B159973

1 Introduction

1.1 Gperftools

Originally Google Performance Tools, Gperftools [1] are state of the art heap and CPU performance profilers. Engineers at Google worked hard for this benchmark framework and professionals extensively use it.

1.1.1 CPU profiler

The Gperftools CPU profiler is an advance CPU profiler that gathers CPU usage by interrupting the program's workflow. The framework can produce an elegant graphical output that helps a developer to understand CPU resource consumption hotspots.

1.1.2 Heap profiler & Heap checker

The heap profiler & checker instrument every memory allocation and deallocation, by keeping track of several sections of information per allocation site. They are responsible for obtaining the program's heap memory in volume at any presented time, discovering and reporting memory leaks and detecting code segments that perform vast allocations.

1.1.3 Thread-Caching Malloc

For running the above profilers, it is necessary that the source code must be linked with TC malloc library. TC Malloc is faster than any other allocation functions (glibc 2.3 malloc & ptmalloc2 & any other malloc). Ptmalloc2 takes approximately 300 nanoseconds to execute a malloc/free pair. TCMalloc implementation takes approximately 50 nanoseconds. The basic engineer behind this is a top scientist, Sanjay Ghemawat.

1.2 Experiments

For instrumenting both tools, two experiments took place regarding the CPU performance in relation with RHO and heap memory in relation with the grid size. The main idea for the present CPU performance analysis is to examine the CPU utilization in all the functions by changing the RHO density parameter and re-execute the program. For heap checking and profiling, the present memory analysis relies on the examination of arralloc() function. Arralloc() allocates the most substantial memory blocks during runtime. Furthermore, the results of a general heap checking are presented.

2 Methodology

2.1 Machines

The experiment took place on, four-core Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz, 8GB RAM and cache size of 4096 KB.The operating system was Ubuntu version 18.04 LTS. Furthermore, Cirrus was used, a 280 computer node, SGI ICE XA system provided by EPCC center [2], using a batch script. Only one node was used, which has 36 cores, 2 sockets with 18 cores. Each Cirrus node has two NUMA nodes and cache levels: L1: 32K, L2: 256K, L3: 46080K.

2.2 Software Packages

Google performance tools (gperftools) were downloaded from the original Github repository [1]. Dependencies for running the google package were libunwind library [3]. After dependencies were installed, the package was configured by following the perftools-specific install notes. Moreover, for plotting the results, python library matplotlib[4] was used extensively. Finally, for automating the jobs and run multiple test cases for gathering measurements, several batch scripts were used.

2.3 Code Segments

In the refactored version of the program, the two header files that call the CPU and Heap profilers were inserted:

```
#include <gperftools/profiler.h>
#include <gperftools/heap-profiler.h>
```

and special functions that start and stop the profiling for the CPU and heap memory. If no profiling functions are embedded inside the code, then the entire application will be profiled instead of a particular segment.

2.3.1 CPU profiling

Firstly, an experiment on CPU profiling showed were the hotspots were located. The two critical parameters that determine the perfomance of the current program is grid size(L) and (RHO). On the initial test case the two parameters were set as L =1000 and RHO = 0.5. All the experiments used the refactored version of the program. (Final version of Assignment 1). The flags:

```
  -lprofiler -Wl,
```

install the CPU profiler into the executable during the building of the source. Therefore, these were added in the config.mk, inside the $CFLAGS variable. Another turn around is to link the libprofiler.so module (which is established by installing the gperftools) by using the environment variable LD_PRELOAD which adds the profiler at run time. Profiling functions were injected at the start and bottom boundaries of main() function.

```
int main(){
    ProfileStart("cpu.prof");
    //..rest of main
    ProfileStop();
}
```

In order for the profiler to run correctly, a specific external variable has to be configured with the name of the profiler ouput. After the declaration of the external variable, the executable must run including the parameters on site.

```
CPUPROFILE=cpu.prof  /refactored_version [args]
```

Furthermore, for more finely control, the environment variable CPUPROFILE_ FREQUENCY which is the number of interrupts/second the profiler samples was set to 100. On smaller grid size and high RHO, the CPUPROFILE_FREQUENCY was increased for better profiling. The executable ran multiple times (about 12 times), and on each occasion, it followed different configuration. The grid size was always constant (L= 1000), while the values of RHO ranged from (0.01 to 0.99).

For analyzing each output profile, the google-pprof module was applied, which is a tool for visualization and interpretation of profiling data. It is not possible to run the executable via a configuration script because the google-pprof visualizer will mismatch the collection of the function names when examining the profile output and will interpret them as hexadecimal addresses. On every run, the tool generated a text and graphical report to visualize and analyze the data. The google-pprof was called with the following command:

```
google-pprof /refactored_version cpu.prof
```

where cpu.prof is an output profile of each successful run using different RHO and constant grid size (L=1000).

2.3.2 Heap profiling & checking

Heap checker & profiler were used to examine the memory heap consumption and leaking of the refactored version of the program. The tools periodically checks any allocation that is established as an active stack trace at the call of malloc, calloc, realloc, or, new. The refactored version source code has to be linked with the following flags:

```
-ltcmalloc
-fno-builtin-malloc -fno-builtin-calloc -fno-builtin-realloc -fno-builtin-free
-no-pie
```

This indicates that the source code must be linked with TCmalloc memory allocation library and disables special handling of the standard C library functions malloc, calloc, realloc and free. It is not obligatory to add the disabling flags, but it is suggested for proper measuring. Also, any occurrence of malloc inside the code would inherit the implementation of TCmalloc, which is much more rapid than any other malloc implementation.

In the boundaries of the main function, profiler functions must be placed:

```
int main(){
    HeapProfilerStart("cpu.prof");
    //..start of main
    //..memory allocations (ex. arralloc() )
    heapProfilerDump();
    //..rest of main
    HeapProfilerStop();
}
```

Since the experiment is concerned about arralloc(), heapProfilerDump() must be signalled for dumping the heap data about the current time-step after the arralloc() functions are called. For more control, the external variable HEAP_PROFILE_ALLOCATION_INTERVAL= 2GB was declared for managing huge grid sizes. The executable ran multiple times on different configurations, with L ranging from 10 to 10000 and RHO constantly to be 0.5. For running the heap profiler, the heap profiler control enviroment veriable was set to:

```
HEAPPROFILER=percolate.hprof  /refactored_version [args]
```

For analyzing the heap output profile, google-pprof was called as:

```
google-pprof /refactored_version percolate.hprof
```

For checking memory heap leaks, heap checker was installed into the building. The configuration for achieving this is as follows: compile the source files using -ltcmalloc flag, which inserts the google's memory allocation library. The advised way to use the heap checker is in the entire program without specifying any function that begins or dispatches the profiling process. In this circumstance, the heap-checker starts tracking memory allocations before the commencement of main(), and investigates again when the program terminates. Moreover, the environment variable HEAPCHECK was set with the legal value of normal:

```
HEAPCHECK=normal  /refactored_version [args]
```

which states that it will track any live memory object and report if data is not reachable when the program stops. Afterwards, the executable runs, the checker outputs to stdout any detected leaks.

2.4 Plotting

Plotting the results was done using python library matplotlib[4]. The results gathered from each run and by inserting them into np arrays [5], were plotted for proper analysis to develop conclusions. Regarding CPU profiling, the top four CPU consuming functions were plotted by the percentage of profiling samples the CPU issued. For heap profiling, the total size of heap memory regarding only a distinct function was plotted in relation with grid size, because all the other functions had minuscule heap memory allocations in comparison with arralloc().

3 Experiment Results

3.1 CPU profiling results

Annotated call-graphs were generated by google-pprof when analyzing the profiler's output. The primary concept is that as the RHO value is becoming more diminutive, the workload inside the function that updates the neighbours of each square in the grid increase almost exponentially. The figures clearly show this correlation:

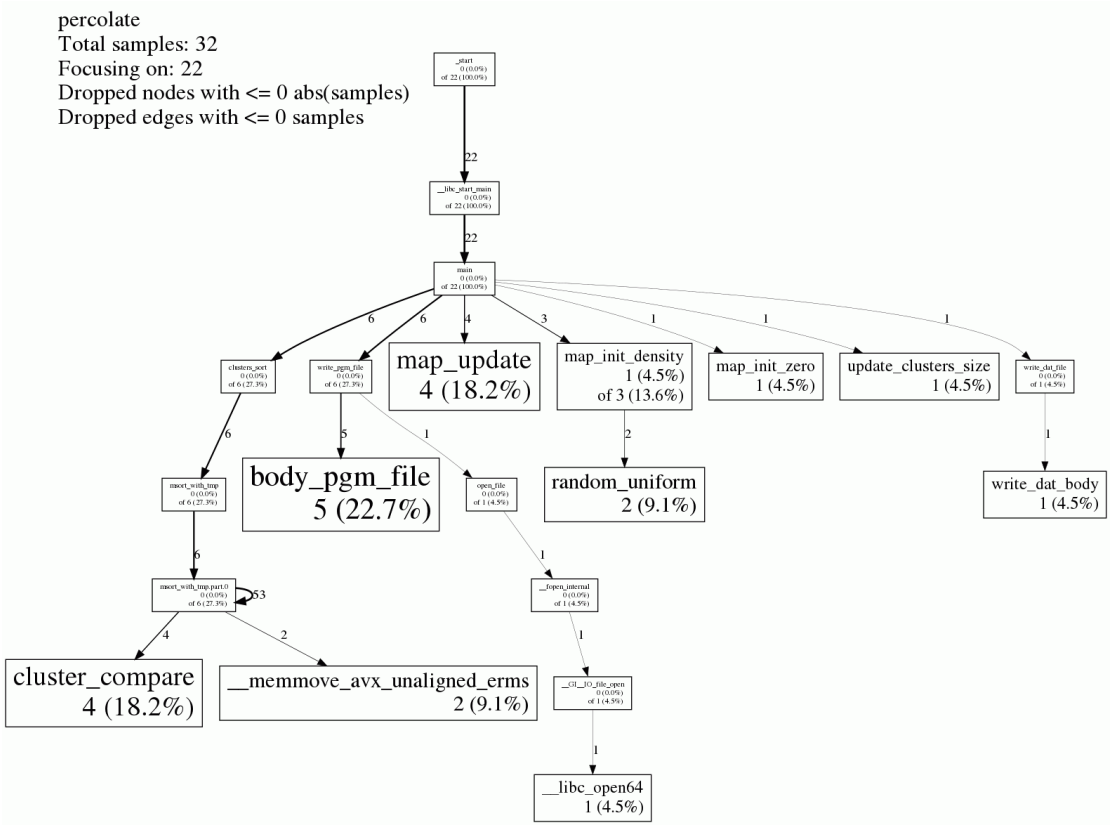

Figure 1: CPU profiling of the refactored program ran with parameters L =1000 and RHO = 0.9. Functions share more or less the load equivalently. There is no outlier function on CPU usage. The size of each box that a function has, indicates the proportion of CPU usage in relation to other functions.

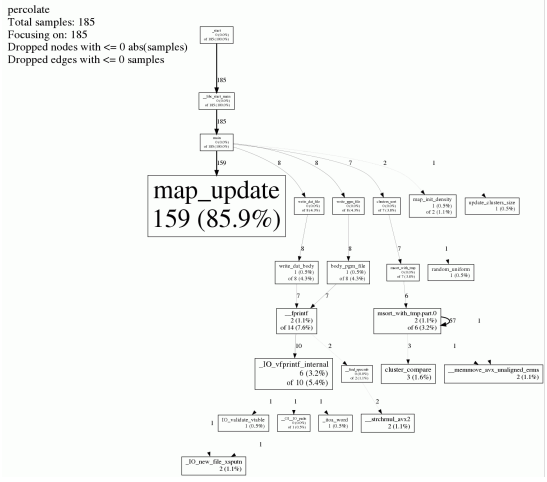

Figure 2: CPU profiling of the refactored program ran with parameters L =1000 and RHO = 0.5. Map_update() seems to significantly outperform any other function.

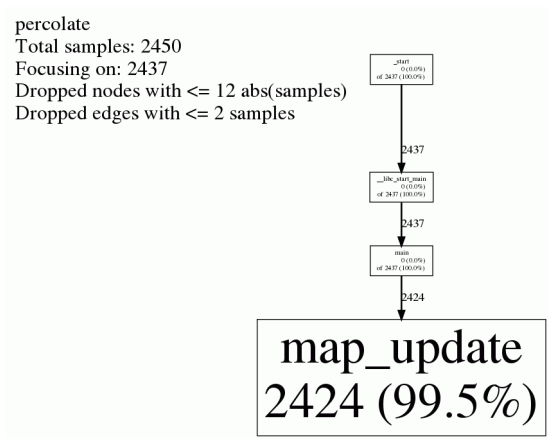

Figure 3: CPU profiling of the refactored program ran with parameters L =1000 and RHO = 0.1. Almost all the workload is in map_update(). All the other functions are not even visualized because of inadequate CPU usage.

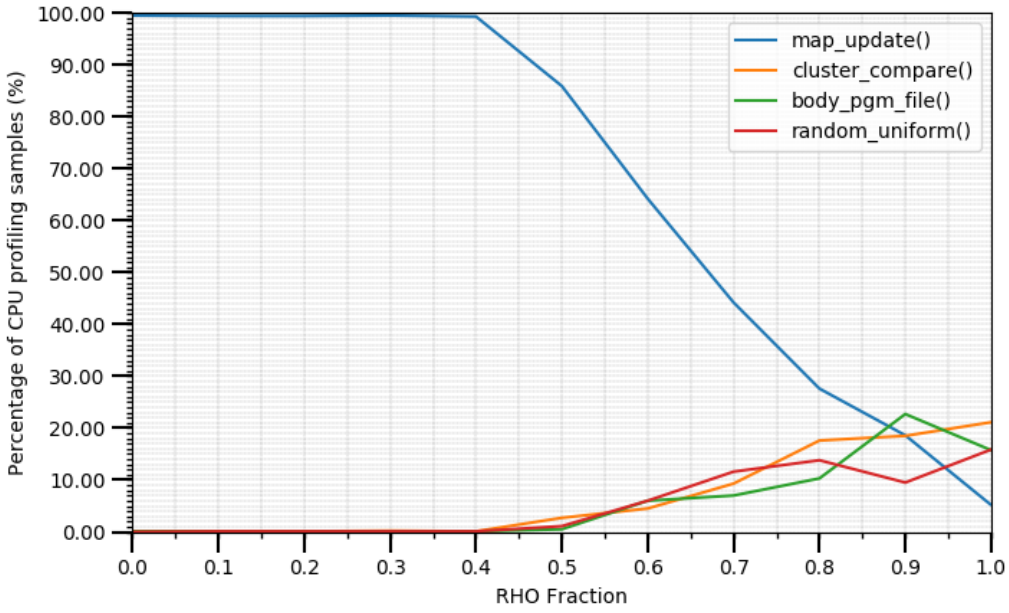

Figure 4: The top 4 CPU resource consuming functions of the program. Map_update() has the most workload for RHO below 0.85. As RHO moves to 0, map_update() outperforms them.

3.2 Heap profiling

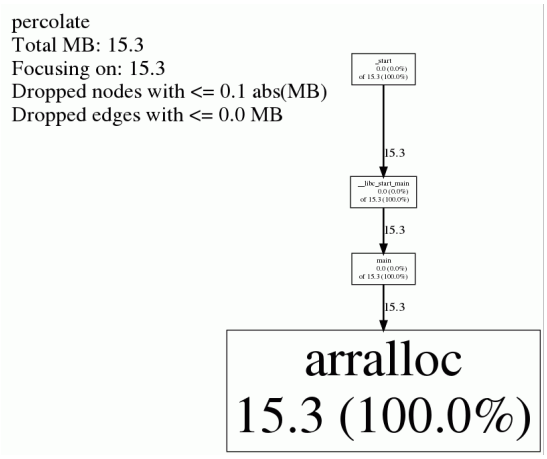

Figure 5: Heap profiling of the refactored program ran with parameters L =1000. Total MB allocated are 15.3 from arralloc function only. Other functions do not allocated any significant data.

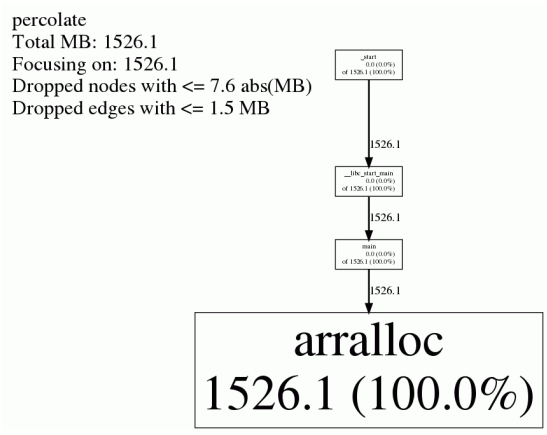

Figure 6: Heap profiling of the refactored program ran with parameters L =10000. Total MB allocated are 1526.1 from arralloc function. Any other function does not allocated any notable data.

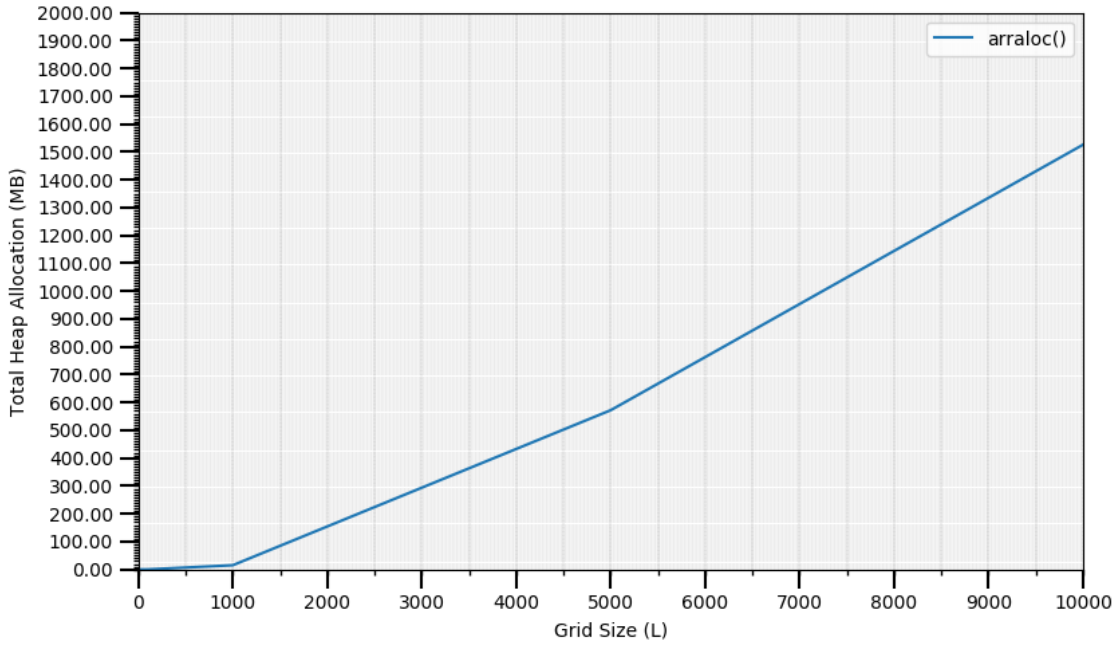

Figure 7: As the grid size increase, total heap allocation by arralloc() is also increasing.

4 Results Discussion & Conclusion

The grid density, which is determined by RHO value, is calculated using the uniform distribution. The function that finds the fraction and compares with the given RHO is random_uniform, located in uni.h, which is called inside the map_init_density(). After this comparison, the square takes either a value of 1 or 0. The lower RHO (max:1 & min:0) is, the higher the number of non-empty squares (allowed_squares) is inside the grid. Later on, each square takes the number from 1 to the allowed_squares, from upper left most in the grid to be 1 and bottom-most right to be the largest (allowed_squares). The function map_update() performs a loop to update the value of each neighbour square (up & down & left & right) of the grid. If more allowed_squares are allocated with numbers in the network, then more computation is needed to compare the neighbours of all the squares each time. Another critical factor that affects CPU performance is the grid size, which obviously the more massive the grid is, further calculations are required.

Map_update() is indeed the most CPU resource consuming function in percolate. By looking at the program's workflow, it's very easy to interpret this conception. Furthermore, data here clearly show that, as there is a decrease in the density of the grid, more iterations are needed in the while loop of map_update(), because there are more available (allowed) squares that need update.

Furthermore, regarding heap profiling, arralloc() appears to do an excellent job and allocate memory precisely as needed. Heap checker stated that in all different executions (L from 10 to 10000), only between 16 to 194 bytes where leaking, a really inadequate memory amount. This is exactly how arralloc() is called inside the code:

```
map = (int **) arralloc(sizeof(int), 2, con.L + 2, con.L + 2);
cluster_list = (cluster *)arralloc(sizeof(cluster), 1, (con.L) * (con.L));
rank_list = (int *)arralloc(sizeof(int), 1, con.L * con.L);
```

where con.L is the grid size (L). The allocated memory in bytes for the particular program should be: $Number_of Lists * Blocks * Size_of Integer$ where a block is a the numbers of integers to allocate. A size of an integer is 4 bytes [sizeof(int)] and size of a cluster is 8 bytes [sizeof(cluster)]. Theoretically, arralloc() should roughly allocate:

1. L = 100 $(4 * 10000 + 4 * 10404 + 8 * 10000)/(1024 * 1024) = 0.1616MB$
Actual is 0.2 MB

2. L = 1000 $(4 * 1000000 + 4 * 1004004 + 8 * 1000000)/(1024 * 1024) = 15.274MB$
Actual is 15.3 MB

3. L = 10000 $(4 * 100000000 + 4 * 100040004 + 8 * 100000000)/(1024 * 1024) = 1526.031MB$
Actual is 1526.1 MB

5 Further Work

A surpassing, more profound experiment is to investigate other algorithm implementations using the current CPU profiler on map_update() function and make comparisons with each one using the total converging time. The way percolate currently starts comparing squares with neighbours looks inefficient. Another likeable future experiment, is to perform a series of precise tests on different memory allocation methods (malloc(), etc) and examine the heap memory.

References

[1] Official Google repository of gperftools https://github.com/gperftools/gperftools Google 2019

[2] EPCC Center(2019) : www.epcc.ed.ac.uk/facilities/demand-computing/cirrus Cirrus Facilities.

[3] Version 1.4 of the Unwind library. https://github.com/libunwind/libunwind David Mosberger

[4] Matplotlib - Python 2D plotting library https://matplotlib.org/contents.html 2019 The Matplotlib development team

[5] SciPY.org - Numpy Array documentation https://docs.scipy.org/doc/numpy/ 2019 The SciPy community