SR. NO.	TITLE	Performance date	Marks out of 10	Sign
1	Study the facilities provided by the google colab.			
2	Demonstrate basic Linux commands.			
3	Using Divide and conquer strategies design a class for concurrent quick sort using c++.			
4	Write a program on an unloaded cluster for several different numbers of nodes and record the time in each case.			
5	Write a program to check task distribution using gprof.			
6	Use intel V-Tune performance analyzer for profiling.			
7	Analyze the code using NVidia-profilers.			
8	Write a program to perform load distribution on GPU using CUDA			
9	Write a simple CUDA program to print “hello world!”.			
10	Write a CUDA program to add two arrays.			

PRACTICAL -5

OBJECTIVE: Write a program to check Task distribution into gprof

What are profilers:

For sequential programs a summary profile is usually sufficient, but performance problems in parallel programs often depend on the time relationship of events, thus requiring a full trace to get an understanding of what is happening. The size of a trace is linear to the program's instruction path length.

Profiling is an important aspect of software programming. Through profiling one can determine the parts in program code that are time consuming and need to be re-written

What is gprof:

Gprof is a profiling tool for the c language that helps analyze the execution time of different functions within a program. It provides valuable insights into the performance of a program by measuring the amount of time spent in each function and identifying potential bottlenecks. Gprof works by instrumenting the program's code and gathering data on function calls and their durations. This data is then used to generate a detailed

report, including a call graph that visualizes the flow of function calls and the time spent in each. By utilizing Gprof, developers can identify critical sections of their code and optimize them to improve overall program efficiency and performance.

Step-1: profiling enabled while compilation

In this first step, we need to make sure that the profilin is enabled when the compilation of code is done. This is made possible by adding the '-pg' option in the compilation step.

-pg: generate extra code to write profile information suitable for the analysis program gprof.

"!gcc -Wall -pg func1.c -o func1"

"!./func1"

Step-2: Execute the code

In the second step, the binary file produced as a result of step-1 (above) is executed so that profiling information can be generated

So, we can see that binary was executed, a new file 'gmon.out' is generated in the current working directory.

Note that while execution if the program changes the current working directory, then gmon.out will be produced in the new current working directory.

Step-3: Run the gprof tool

In this step, the gprof tool is run with the executable name and the above generated 'gmon.out' as argument. This produces an analysis file which contains all the desired profiling information

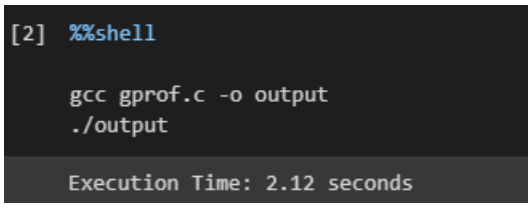
"!gprof func1 gmon.out > func.txt" and " !cat func.txt"

C PROGRAM:

```
%%writefile gprof.c
#include <stdio.h>
#include <time.h>
void timeConsumingTask() {
 for (int i = 0; i < 1000000000; i++) {
 // Perform some computations
} }
int main() {
 clock_t start = clock();
 timeConsumingTask(); // Perform time-consuming task
 clock_t end = clock();
 double executionTime = (double)(end - start) / CLOCKS_PER_SEC;

 printf("Execution Time: %.2f seconds\n", executionTime);
 return 0;
}
```

OUTPUT:



!gprof:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.00      2.10     2.10        1     2.10     2.10  timeConsumingTask

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.

Copyright (C) 2012-2022 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
```

```
                     Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 0.48% of 2.10 seconds

index % time    self  children    called     name
                2.10    0.00       1/1           main [2]
[1]    100.0    2.10    0.00       1         timeConsumingTask [1]
-----------------------------------------------
                                              <spontaneous>
[2]    100.0    0.00    2.10                 main [2]
                2.10    0.00       1/1           timeConsumingTask [1]
-----------------------------------------------

 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.

 Each entry in this table consists of several lines.  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.
 This line lists:
     index      A unique number given to each element of the table.
                Index numbers are sorted numerically.
                The index number is printed next to every function name so
                it is easier to look up where the function is in the table.

     % time     This is the percentage of the `total' time that was spent
                in this function and its children.  Note that due to
                different viewpoints, functions excluded by options, etc,
                these numbers will NOT add up to 100%.

     self       This is the total amount of time spent in this function.

     children   This is the total amount of time propagated into this
                function by its children.

     called     This is the number of times the function was called.
                If the function called itself recursively, the number
                only includes non-recursive calls, and is followed by
                a `+' and the number of recursive calls.
```