Assignment No: B4

Title: Implement a program to check task distribution using Gprof.

Aim: Write a program to check task distribution using Gprof.

Objectives:

- To understand the use of Gprof for task distribution.

Theory:

Gprof

Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing. This information can show you which pieces of your program are slower than you expected, and might be candidates for rewriting to make your program execute faster. It can also tell you which functions are being called more or less often than you expected. This may help you spot bugs that had otherwise been unnoticed.

Since the profiler uses information collected during the actual execution of your program, it can be used on programs that are too large or too complex to analyze by reading the source. However, how your program is run will affect the information that shows up in the profile data. If you don't use some feature of your program while it is being profiled, no profile information will be generated for that feature.

Profiling has several steps:

- You must compile and link your program with profiling enabled.
- You must execute your program to generate a profile data file.
- You must run gprof to analyze the profile data.

Performance is one of the biggest challenges programmers face while developing software. That is the reason why code profiling is one of the most important aspects of software development, as it lets you identify bottlenecks, dead code, and even bugs. If you are a programmer who develops software applications for Linux, the GNU profiler “gprof” is the tool to look out for."gprof" produces an execution profile of C, Pascal, or Fortran77 programs. The effect of called routines is incorporated in the profile of each caller. The profile data is taken from the call graph profile file (gmon.out default) which is created by programs that are compiled with the -pg option of "cc", "pc", and "f77". The -pg option also links in versions of the library routines that are compiled for profiling. "Gprof" reads the given object file (the default is "a.out") and establishes the relation between its symbol table and the call graph profile from gmon.out. If more than one profile file is specified, the "gprof" output shows the sum of the profile information in the given profile files.

"Gprof" calculates the amount of time spent in each routine. Next, these times are propagated along the edges of the call graph. Cycles are discovered, and calls into a cycle are made to share the time of the cycle. Several forms of output are available from the analysis.

The flat profile shows how much time your program spent in each function, and how many times that function was called. If you simply want to know which functions burn most of the cycles, it is stated

concisely here. The call graph shows, for each function, which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function. This can suggest places where you might try to eliminate function calls that use a lot of time. The annotated source listing is a copy of the program's source code, labeled with the number of times each line of the program was executed.

Download and Install

Gprof comes pre-installed with most of the Linux distributions, but if that‟s not the case with your Linux distro, you can download and install it through a command line package manager like apt-get or yum. For example, run the following command to download and install gprof on Debian-based systems:

sudoapt-get install binutils

How to use gprof

Using the gprof tool is not at all complex. You just need to do the following on a high-level:

- Have profiling enabled while compiling the code
- Execute the program code to produce the profiling data
- Run the gprof tool on the profiling data file (generated in the step above).

The last step above produces an analysis file which is in human readable form. This file contains a couple of tables (flat profile and call graph) in addition to some other information. While flat profile gives an overview of the timing information of the functions like time consumption for the execution of a particular function, how many times it was called etc. On the other hand, call graph focuses on each function like the functions through which a particular function was called, what all functions were called from within this particular function etc So this way one can get idea of the execution time spent in the sub-routines too.

Using gprof Task Distribution Performance Analysis:

Consider the following C program as an example:

```
#include <stdio.h>
void func2()
        {
                 int count = 0;
                for(count=0; count < 0XFFFFF; count++);
                return;
        }
 void func1(void)
        {
                 int count = 0;
                for(count=0; count < 0XFF; count++)
                func2();
                return;
        }
 int main(void)
        {
                 printf("\n Hello World! \n");
                func1();
                func2();
```

```
        return 0;
    }
```

Compile the code with the -pg option:

gcc -Wall -pgtest.c -o test

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

You must use this option when compiling the source files you want data about, and you must also use it when linking.

Once compiled, run the program:

./test

After successful execution, the program will produce a file named "gmon.out" that contains the profiling information, but in a raw form, which means that you cannot open the file and directly read the information. To generate a human readable file, run the following command:

gproftestgmon.out>prof_output

This command writes all the profiling information in human readable format to "prof_output" file. Note that you can change the output file name as per your convenience.

Flat profile and Call graph

If you open the file containing profiling data, you'll see that the information is divided into two parts: Flat profile and Call graph. While the former contains details like function call counts, total execution time spent in a function, and more, the latter describes the call tree of the program, providing details about the parent and child functions of a particular function.

For example, the following is the Flat profile in our case:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.00	0.94	0.94	256	3.67	3.67	func2
0.00	0.94	0.00	1	0.00	936.33	func1

The below is the Call graph:

Index	%time	self	children	called	name
		0.00	0.00	1/256	main [2]
		0.94	0.00	255/256	func1 [3]
[1]	100.0	0.94	0.00	256	func2 [1]
---					
[2]	100.0	0.00	0.94		main [2]
		0.00	0.94	1/1	func1 [3]
		0.00	0.00	1/256	func2 [1]

		0.00	0.94	1/1	main [2]
[3]	99.6	0.00	0.94	1	func1 [3]
		0.94	0.00	255/256	func2 [1]

Here is the explanation (taken from the file output) of what each column means:

- **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. For function's parents, this is the amount of time that was propagated directly from the function into this parent. While, for function's children, this is the amount of time that was propagated directly from the child into the function.
- **Children -** This is the total amount of time propagated into this function by its children. For the function's parents, this is the amount of time that was propagated from the function's children into this parent. While, for the function's children, this is the amount of time that was propagated from the child's children to the function.
- **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. For the function's parents, this is the number of times this parent called the function `/' the total number of times the function was called. Recursive calls to the function are not included in the number after the `/'. While, for the function's children, this is the number of times the function called this child `/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the `/'.
- **Name -** The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number. For function's parents, this is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number. For function's children, this is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

Conclusion: Hence we successfully checked task distribution using Gprof..

CODE

```
#include <stdio.h>
void func2()
{
        int count = 0;
        for(count=0; count < 0XFFFFF; count++);
        return;
}
void func1(void)
{
        int count = 0;
        for(count=0; count < 0XFF; count++)
                func2();
        return;
}
int main(void)
{
        printf("\n Hello World! \n");
        func1();
        func2();
        return 0;
}
```

OUTPUT

```
mpiu@DB21:/mirror/mpiu/CL4 prog/B-4$ g++ test.c
mpiu@DB21:/mirror/mpiu/CL4 prog/B-4$ ./a.out

Hello World!
mpiu@DB21:/mirror/mpiu/CL4 prog/B-4$
```