Assessment2-Performance

November 2020

Contents

1 Introduction

In this report, we design and carry out a performance experiment to investigate the behaviour or performance of the program, which solves the *Percolate* problem in C code, when configured with different sized maps of the grid of squares. This report analyzes the performance of the program on seven different sizes of maps. The performance analysis includes the memory overhead and CPU overhead when the entire program is running. We also analyze the performance of the core code running time and the number of loops.

The remainder of this report is organized as follows: In Section 2, we designed a suitable method to analyze the performance of the memory overhead and CPU overhead respectively when running the entire program in the first two subsections. In the next two subsections, we analyze the performance of the running time and the number of loops respectively when running on the core code. In Section 3, we give the results of the above four experiments. In Section 4, we analyze and discuss the results. In Section 5, we summarize our work and give some suggestions for the future work.

2 Method

We selected five sizes of maps for the experiment, the widths/heights are 20, 40, 80, 160, 320, 640 and 1280 respectively with a multiple of 2, while other parameters are default values. We didn't test the larger size because it is a time consuming work. Our program is compiled in *gcc -g -c arralloc.c uni.c percolate.c* | *gcc -g -o percolate arralloc.o uni.o percolate.o -lm* | *./percolate* with *gcc 9.3.0* in *Ubuntu 10.04.1*, *Intel Core i5-9400F(one core)*, *4G RAM.* We run program three times for each size of an experiment and take the average of the results.

2.1 Memory Overhead

We use Valgrind, a free memory debugging tool for Linux, to get the memory overhead of the entire program at runtime. We use command *sudo apt-get install -y valgrind* in Ubuntu to install it. We use command *valgrind ./percolate* to get the memory overhead, also we can add flag *–leak-check=yes* to check whether there exists one(more) memory leak(s).

2.2 CPU Overhead

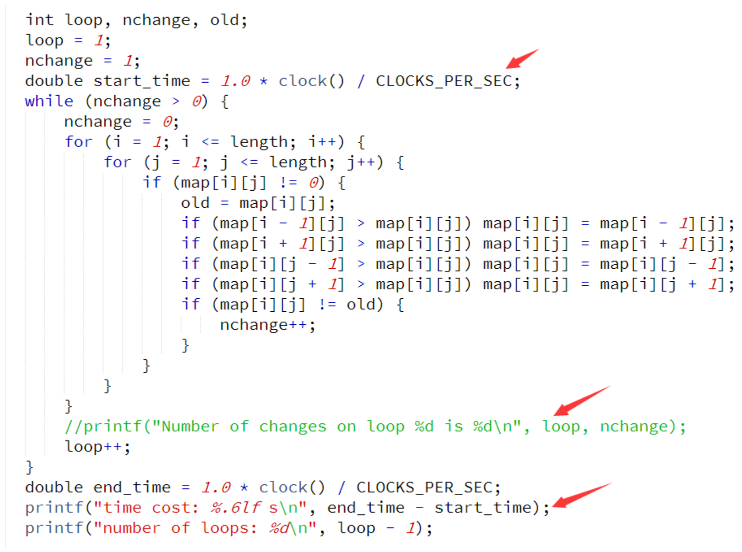
We use *top* command to monitor CPU resource usage on Linux. It shows resource usage of all processes while we just need to show usage of our program, thus we use *grep* command to filter the result. The default refresh time interval is 3 seconds and we can use *-d* flag to set it. The final command we use to get the CPU overhead is *top -d 0.00001* | *grep percolate* and we run our program in another console terminal.

2.3 Running Time

We use a simple method to analyze the running time and number of loops experiments that we modify the program to analyze the running time of the core code as shown in Figure 1.

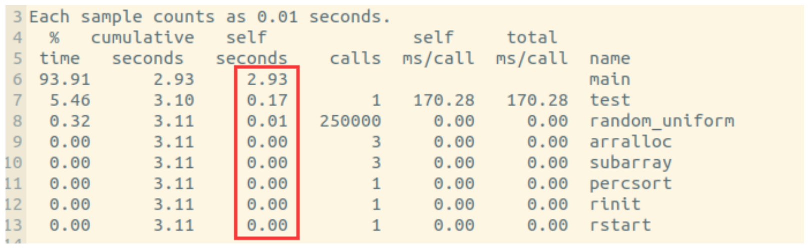
We use *clock()* function and *CLOCKS_PER_SEC* in *time.h* to get the running time in seconds of the core code. We also comment the *printf()* function to eliminate the influence of console output.

We can also use *gprof*, a profiling program which can collect timing information during program running. We install it by command *sudo apt-get install binutils*, then we run *gprof* we get *a.out: No such file or directory* which means it installed successful. When we compile our program, we add flag *-pg*, this means we will use *gprof* to analyze it. We run our program and get a file named *gmon.out*, that we can not read it directly so we convert it into information we can read by *gprof percolate gmon.out* > *out.txt*. We can get running time of all of the functions as shown in Figure 2. We encapsulate the core code into a function to get more accurate running time result.



```
int loop, nchange, old;
loop = 1;
nchange = 1;
double start_time = 1.0 * clock() / CLOCKS_PER_SEC;
while (nchange > 0) {
    nchange = 0;
    for (i = 1; i <= length; i++) {
        for (j = 1; j <= length; j++) {
            if (map[i][j] != 0) {
                old = map[i][j];
                if (map[i - 1][j] > map[i][j]) map[i][j] = map[i - 1][j];
                if (map[i + 1][j] > map[i][j]) map[i][j] = map[i + 1][j];
                if (map[i][j - 1] > map[i][j]) map[i][j] = map[i][j - 1];
                if (map[i][j + 1] > map[i][j]) map[i][j] = map[i][j + 1];
                if (map[i][j] != old) {
                    nchange++;
                }
            }
        }
    }
    //printf("Number of changes on loop %d is %d\n", loop, nchange);
    loop++;
}
double end_time = 1.0 * clock() / CLOCKS_PER_SEC;
printf("time cost: %.6lf s\n", end_time - start_time);
printf("number of loops: %d\n", loop - 1);
```

Figure 1: Modified Core Code.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 93.91      2.93     2.93                             main
  5.46      3.10     0.17        1   170.28   170.28  test
  0.32      3.11     0.01   250000     0.00     0.00  random_uniform
  0.00      3.11     0.00        3     0.00     0.00  arralloc
  0.00      3.11     0.00        3     0.00     0.00  subarray
  0.00      3.11     0.00        1     0.00     0.00  percsort
  0.00      3.11     0.00        1     0.00     0.00  rinit
  0.00      3.11     0.00        1     0.00     0.00  rstart
```

Figure 2: Gprof Profiling Result.

2.4 Number of Loops

As the description in Section 2.3, we get the results of running time and number of loops simultaneously.

3 Results

3.1 Memory Overhead

The result of the data that we get from experiment is shown in Table 3.1 and Figure 3.1.

Map Size	Memory Overhead(bytes)
20	**20,304**
40	**49,584**
80	**165,744**
160	**628,464**
320	**2,475,504**
640	**9,855,984**
1280	**39,362,544**

Table 1: Memory Overhead.

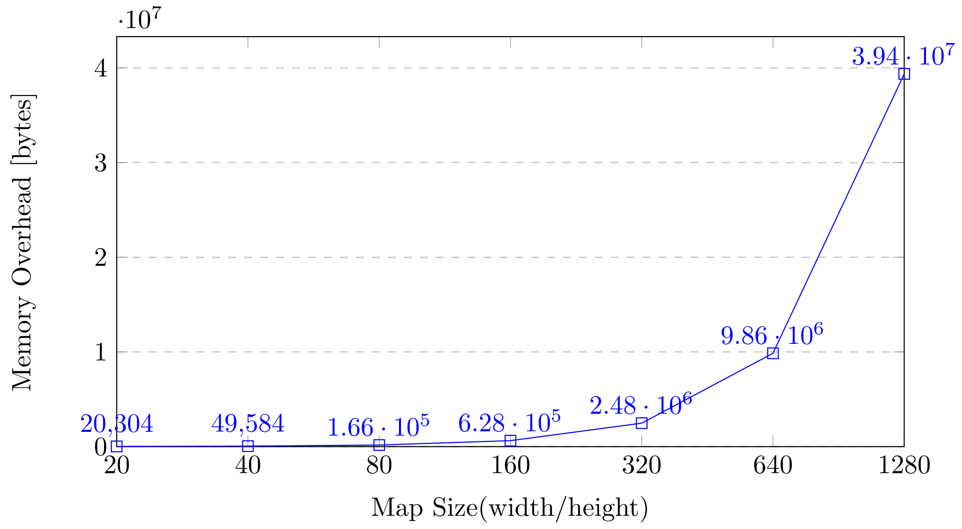
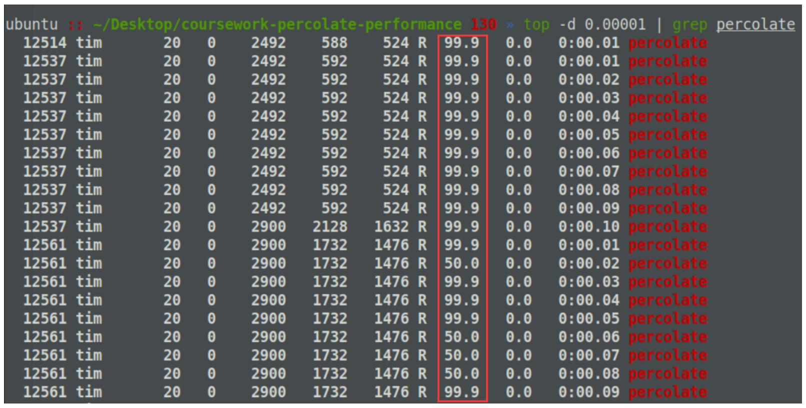


Figure 3.1: Memory Overhead.

3.2 CPU Overhead

We find out that *top* command can not capture the CPU usage if the program run too fast, e.g., map size is 20,40,80. We also find out that program use almost entire CPU resource in all cases, led to result of 99.9% CPU usage as shown in Figure 3 with red frame.



```
ubuntu :: ~/Desktop/coursework-percolate-performance 130 » top -d 0.00001 | grep percolate
  12514 tim       20   0    2492    588    524 R  99.9   0.0   0:00.01 percolate
  12537 tim       20   0    2492    592    524 R  99.9   0.0   0:00.01 percolate
  12537 tim       20   0    2492    592    524 R  99.9   0.0   0:00.02 percolate
  12537 tim       20   0    2492    592    524 R  99.9   0.0   0:00.03 percolate
  12537 tim       20   0    2492    592    524 R  99.9   0.0   0:00.04 percolate
  12537 tim       20   0    2492    592    524 R  99.9   0.0   0:00.05 percolate
  12537 tim       20   0    2492    592    524 R  99.9   0.0   0:00.06 percolate
  12537 tim       20   0    2492    592    524 R  99.9   0.0   0:00.07 percolate
  12537 tim       20   0    2492    592    524 R  99.9   0.0   0:00.08 percolate
  12537 tim       20   0    2492    592    524 R  99.9   0.0   0:00.09 percolate
  12537 tim       20   0    2900   2128   1632 R  99.9   0.0   0:00.10 percolate
  12561 tim       20   0    2900   1732   1476 R  99.9   0.0   0:00.01 percolate
  12561 tim       20   0    2900   1732   1476 R  50.0   0.0   0:00.02 percolate
  12561 tim       20   0    2900   1732   1476 R  99.9   0.0   0:00.03 percolate
  12561 tim       20   0    2900   1732   1476 R  99.9   0.0   0:00.04 percolate
  12561 tim       20   0    2900   1732   1476 R  99.9   0.0   0:00.05 percolate
  12561 tim       20   0    2900   1732   1476 R  50.0   0.0   0:00.06 percolate
  12561 tim       20   0    2900   1732   1476 R  50.0   0.0   0:00.07 percolate
  12561 tim       20   0    2900   1732   1476 R  50.0   0.0   0:00.08 percolate
  12561 tim       20   0    2900   1732   1476 R  99.9   0.0   0:00.09 percolate
```

Figure 3: CPU Overhead.

3.3 Running Time

The result of the data that we get from experiment is shown in Table 3.3 and Figure 3.3.

Map Size	1st Running	2nd Running	3rd Running	Average
20	0.000105	0.000104	0.000104	**0.000104**
40	0.001237	0.001255	0.001248	**0.001247**
80	0.011151	0.011267	0.011818	**0.011412**
160	0.100917	0.095535	0.095107	**0.097186**
320	0.759420	0.762362	0.756712	**0.759498**
640	6.120218	6.118672	6.146592	**6.128490**
1280	48.646746	48.865194	48.928090	**48.813300**

Table 2: Running Time Result in Second.

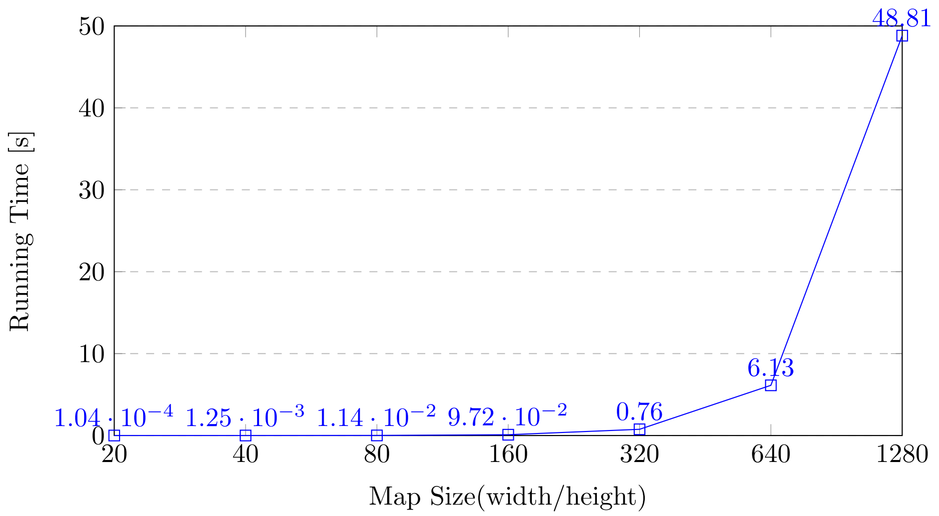


Figure 3.3: Running Time Result in Second.

3.4 Number of Loops

The result of the data we get from experiment is shown in Table 3.4 and Figure 3.4.

Map Size	Number of Loops
20	**28**
40	**77**
80	**159**
160	**313**
320	**633**
640	**1284**
1280	**2554**

Table 3: Number of Loops.

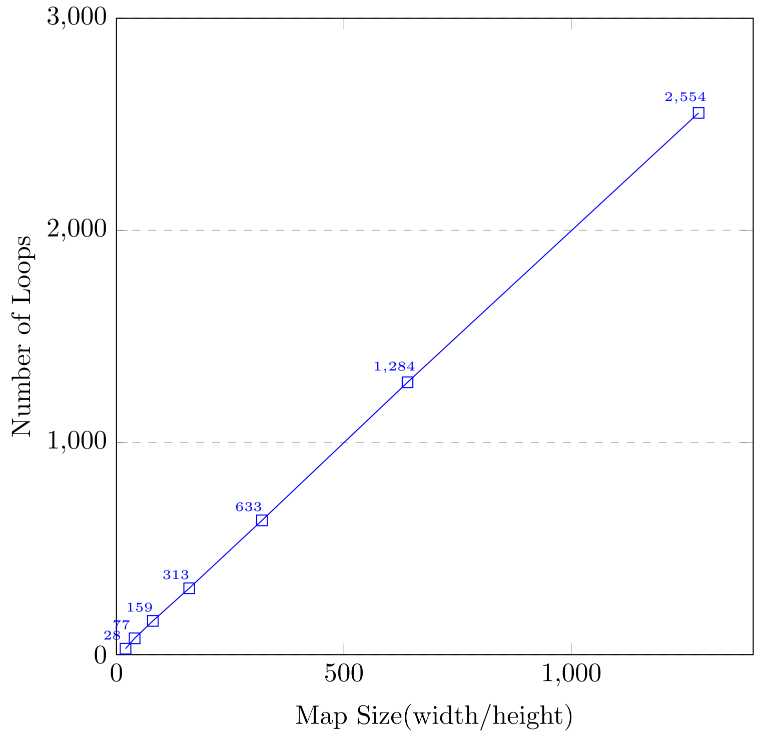


Figure 3.4: Number of Loops.

4 Discussion

All of our experiments show detailed data show in tables and figures in section 3 except monitoring in CPU overhead. This is because our computer allocates almost all CPU resources to run the program.

We can analyze from results of Figure 3.1 and 3.3 in memory overhead and running time that the curves of both have similar shapes. This shows that there is a close coupling between memory overhead and running time, because the size of the map is directly related to the size of the memory used and the number of *FOR* loop traversals. The larger the map size, the larger the memory required, and the more loops.

We can analyze from Figure 3.4 that the map size and number of loops have a proportional relationship. Assume map size be x and number of loops be f, then the relationship between them is $f \approx 2x$.

We once again analyze the result of running time only for the reason that running time and memory overhead have a close coupling. We divide one result by the previous result to get the ratio between two adjacent running time and the results are in Table 4 and Figure 4.1.

n	n_{th} running time/$n-1_{th}$ running time
2	**11.99**
3	**9.15**
4	**8.52**
5	**7.81**
6	**8.06**
7	**7.96**

Table 4: Number of Loops.

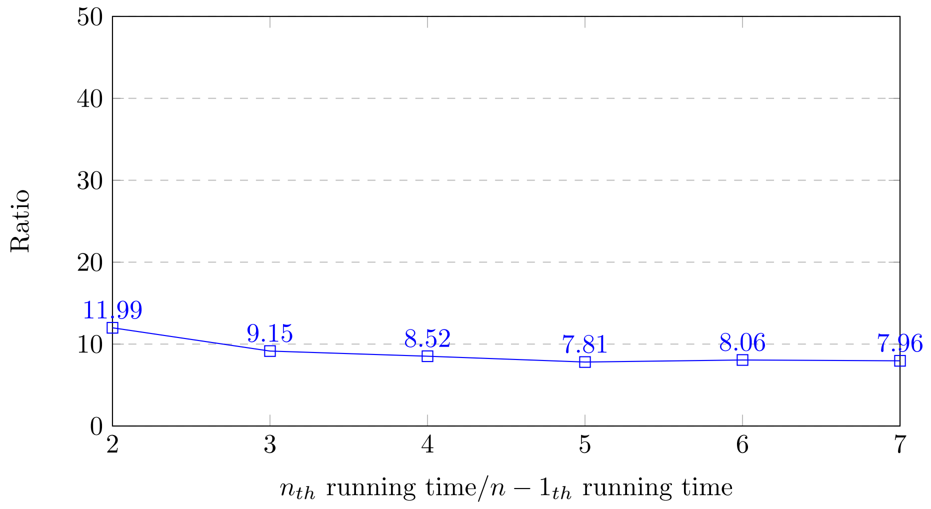


Figure 4.1: Ratio of Two Adjacent Running Time.

We can find that the ratio between two adjacent running time is similar, around 10 ± 2.

5 Conclusion

In this report, we designed and performed performance experiments for the existing complete program that aims to solve the *Percsort* problem. We designed methods to get results of the memory overhead, CPU overhead, running time and number of loops. We find some interesting results, e.g., the memory overhead and running time have a close coupling, the map size and number of loops have a proportional relationship, and the ratio between two adjacent running time is similar. In the future, we plan to use a higher performance computer to do the above experiments to get a more detailed analysis.