Lab3 性能分析

181220075 周其乐

命令行参数（./main perf -r rounds function）

通过 strcmp 条件判断，保证输入的命令参数的格式的规范性，若不符合格式要求直接返回。通过 strcpy 和 atoi，来转化得到参数运行轮数 rounds 和函数名称 function。

尽量精确统计运行时间

和 Lab1 中一样，在 gettime 函数中采用 clock 函数获得当前时刻的时间，并将时间的数据类型定为 double，以提高精度。这样基于的时间单位是**微秒**（Microsecond）。因为 multimod_p2，multimod_p3 的单次运行时间都极短，因此微秒是一个比较合适的统计时间的单位。

因为 multimod 的三个函数都需要 a，b，m 三个数进行计算，而要随机产生这三个数，一定要花费若干时间。所以，若要尽量精确统计 multimod 的运行时间，尽可能减少误差对最终统计信息的影响，就要**把产生随机数的过程，隔离于 multimod 之外**。

为达到隔离目的，先将 a，b，m 在 perf.h 中设置为全局变量，然后在 main.c 中加入多写的 **rand_64_bit 函数和 gen_rand_abm 函数**。rand_64_bit 函数生成一个 64 位的大数，gen_rand_abm 函数通过 srand((unsigned)clock())设置种子，然后分别调用 rand_64_bit 函数求出 a，b，m 三个随机数。最后在 run 函数里、运行 rounds 次 funtion 的循环体中，加入 is_multimod 变量。当判断得将要测试函数为 multimod 时，就调用 gen_rand_abm 生成一组新的（a，b，m）值，然后进行一轮测试。

此部分修改代码如下：

```
int is_multimod=0;
  if(strcmp(function,"multimod_p1")==0 ||\
     strcmp(function,"multimod_p2")==0 ||\
     strcmp(function,"multimod_p3")==0){
        is_multimod=1;
  }
  for (int round = 0; round < rounds; round++) {
    if(is_multimod){
      gen_rand_abm();
    }
    double st = gettime();
    func();
```

```
    double ed = gettime();
    elapsed[round] = ed - st;
  }
```

生成数据文件，并用图片初步表示运行时间统计信息

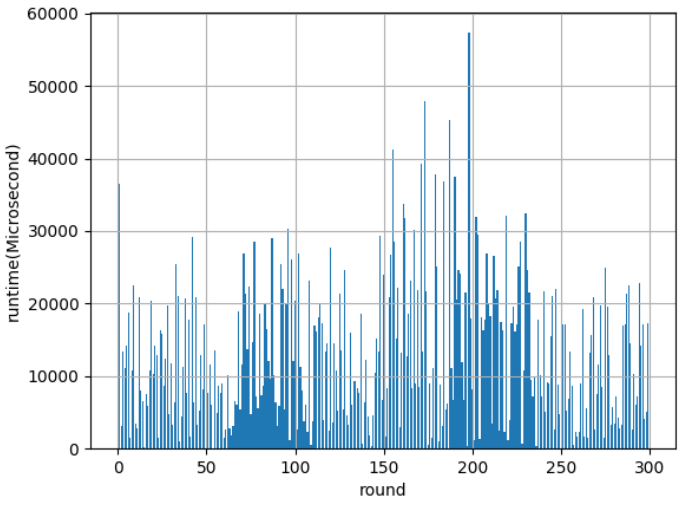
为了使统计信息以用户友好地形式更加直观地展现，通过 perform.py 代码，调用 pandas，numpy，matplotlib.pyplot 进行数据的可视化操作。

为了生成统计数据，首先在 run 函数中，把 elapsed 数组中存放的运行时间记录到 timedata 文件。随后执行 perform.py 进行 timedata 文件的处理。
此部分代码如下：

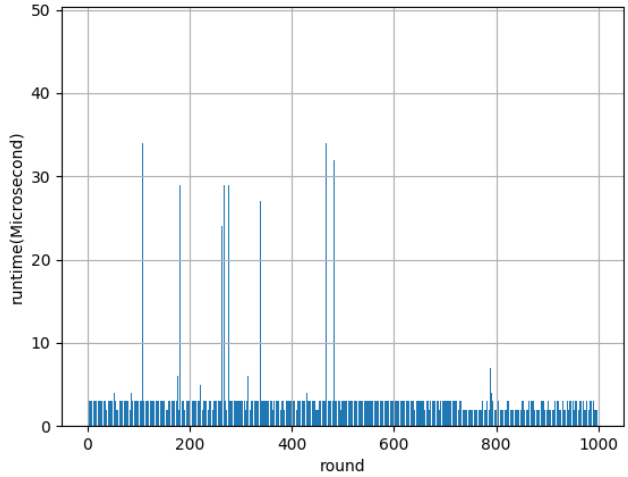
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

timedata = pd.read_csv("./timedata",header=None)
print(timedata.describe())
timedata.columns = ["runtime(Microsecond)"]
plt.bar(timedata.index, timedata["runtime(Microsecond)"])
plt.ylabel("runtime(Microsecond)")
plt.xlabel("round")
plt.grid()
plt.show()
```

初步测试结果展示

multimod_p1**(multimod_p1 运行时间过于长，为得到更多测试数据，使用较小测试数)**

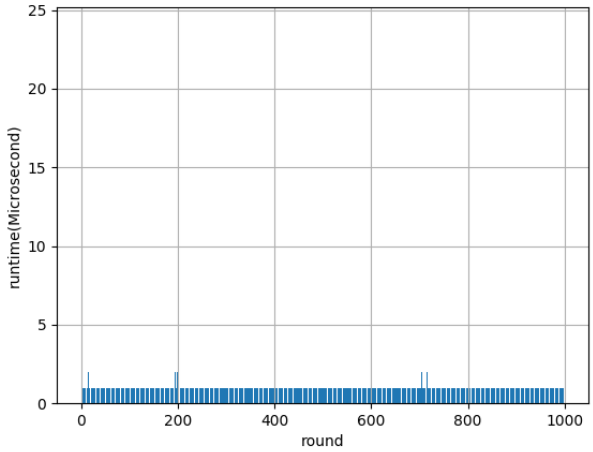


multimod_p2



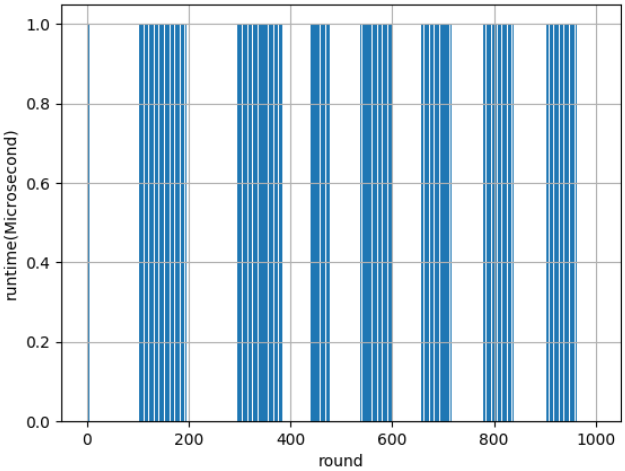
	0
count	1000.000000
mean	2.588000
std	4.044331
min	1.000000
25%	1.000000
50%	2.000000
75%	2.000000
max	55.000000

multimod_p3



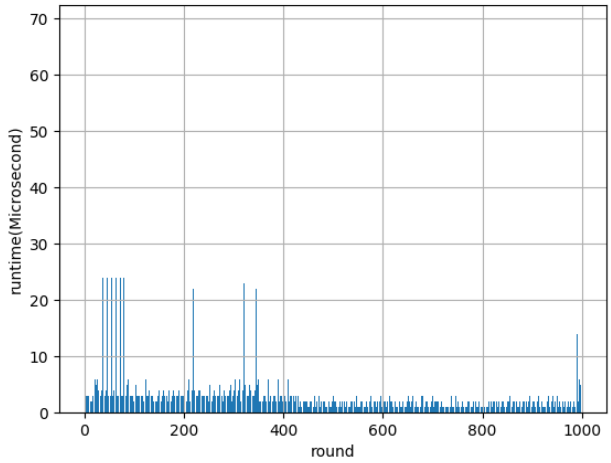
	0
count	1000.000000
mean	1.136000
std	1.704232
min	0.000000
25%	1.000000
50%	1.000000
75%	1.000000
max	30.000000

dummy



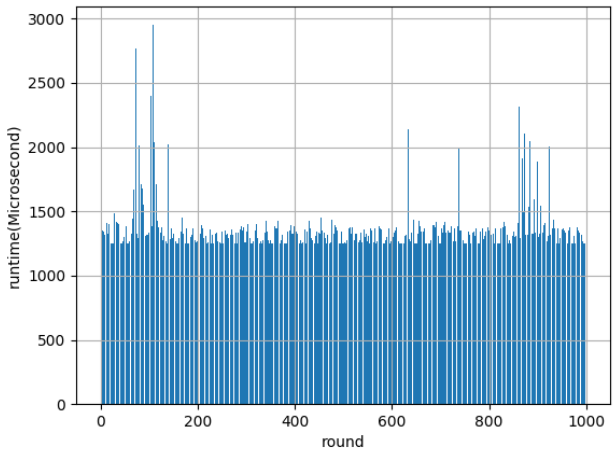
	0
count	1000.000000
mean	0.470000
std	0.499349
min	0.000000
25%	0.000000
50%	0.000000
75%	1.000000
max	1.000000

print_hello



	0
count	1000.000000
mean	3.045000
std	4.321997
min	1.000000
25%	2.000000
50%	2.000000
75%	3.000000
max	69.000000

simple_loop



	0
count	1000.000000
mean	1378.688000
std	227.410541
min	1253.000000
25%	1268.000000
50%	1326.000000
75%	1373.000000
max	2951.000000

正态分布检验（使用 Kolmogorov-Smirnov 检验方法）

Kolmogorov-Smirnov 检验方法通过比较频率分布 f(x)与理论分布 g(x), 来检验数据是否符合某种分布。当 pvalue 大于 0.05 时，一般可认为符合分布。

在 perform.py 中，通过调用 scipy 中的 stats 进行 KS 检验。
代码如下：

```
#Kolmogorov-Smirnov
u = timedata['runtime(Microsecond)'].mean()
std = timedata['runtime(Microsecond)'].std()
print(stats.kstest(timedata['runtime(Microsecond)'],\
      'norm', (u, std)))
# if p > 0.05, it is norm
```

最终计算测试结果为：

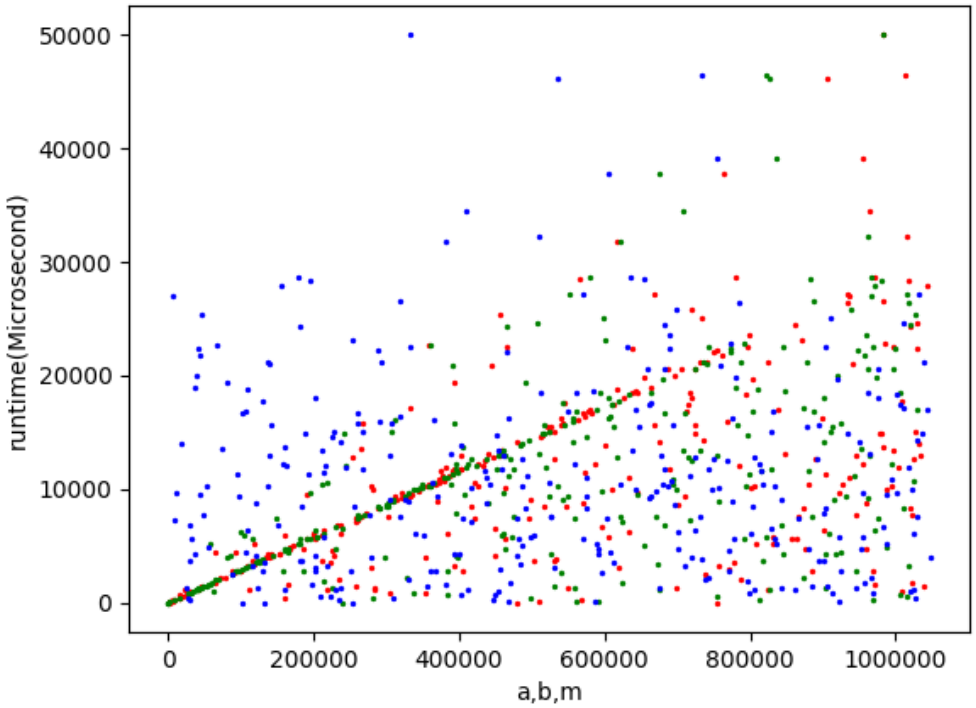
	multimod_p1	multimod_p2	multimod_p3	dummy	print_hello	simple_loop
pvalue	0.003823	1.21e-154	3.51e-118	2.15e-163	1.90e-96	6.19e-69
正态分布判断	较为接近正态分布	否	否	否	否	否

multimod 性能与各因素的关系

为了实现此功能，需要将 a，b，m 传入 perform.py 进行处理。因此在 run 函数的循环体中加入以下代码：

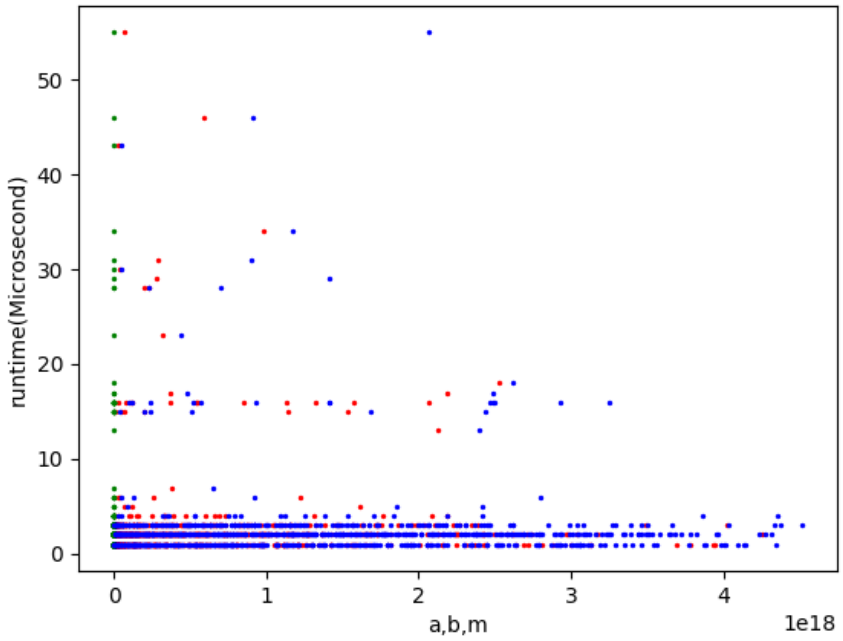
```
1.  if(is_multimod){
2.          fprintf(file,"%f %lld %lld %lld\n",elapsed[round],a,b,m);
3.      }
```

绘制(a，b，m)各数大小与 runtime 的关系的散点图。(a 为红色，b 为绿色，m 为蓝色)
multimod_p1(multimod_p1 运行时间过于长，为得到更多测试数据，使用较小测试数)

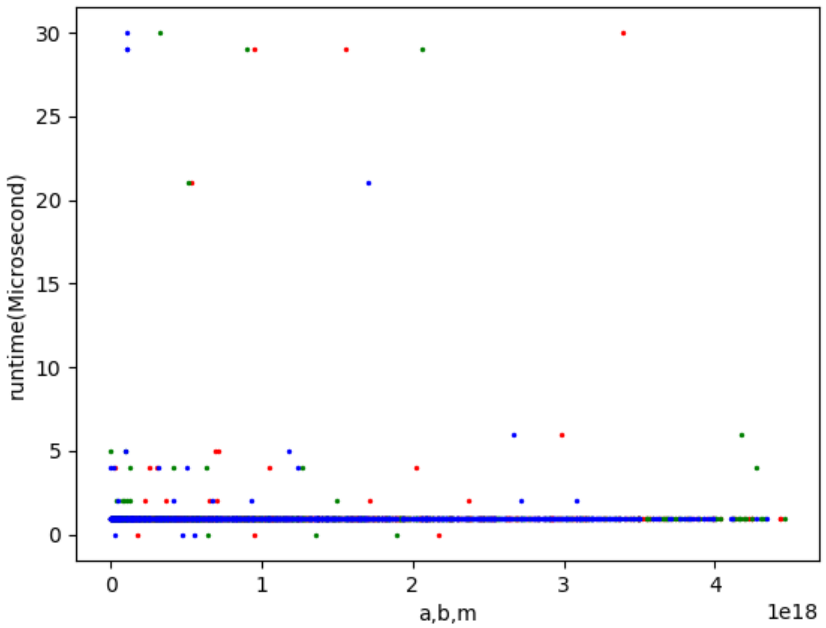


从散点图来看，multimod_p1 的运行时间与红点、绿点有比较强的线性关系。即，与 a 和 b 的大小有很强的线性关系。这与 p1 函数的实现方式相吻合。即先取 a，b 中小的那一个数，然后循环 min{a，b}次。时间复杂度确实与 a，b 有着线性关系。这与实验数据画得的散点图相互映衬。对于 m 来说，m 只是充当循环体中的一个除数，所以从图中也很难看出 m 的大小与运行时间的关系。

multimod_p2



multimod_p3

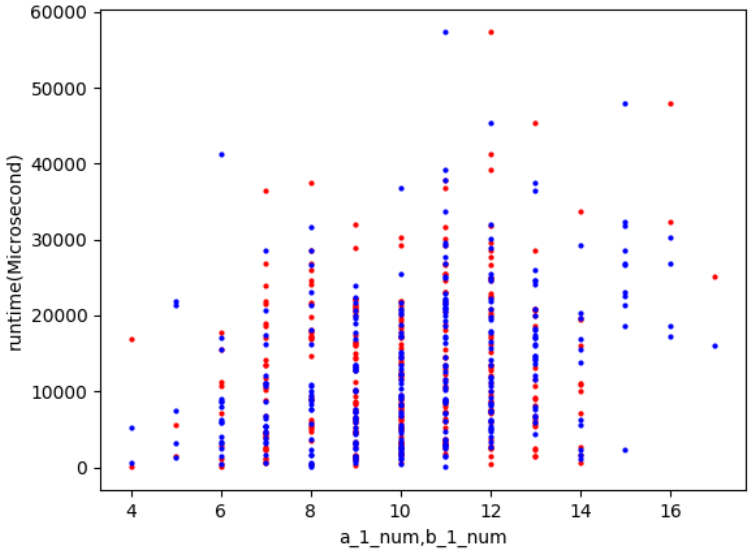


从散点图来看，multimod_p2 与 multimod_p3 的运行时间基本上都集中在某一水平上，与 a，b，m 的大小没有较为显著的关系。其中出现了一些点远离数据点聚集地的情况，虽然这一小部分点的 x 值较小，但这并不能说明规律，因为（a，b，m）三元组共同约束运行时间的长短，一个元素的较小值可能因另外两个元素的较大而影响受到削减。还有一个情况是在 x 值越大时，这些“离群点”出现得较少。原因分析是写的随机数函数产生的极大数密度不大，所以出现小概率离群点的概率会更加小，因此较为稀疏。

绘制（a，b）二进制数中 1 的数量与 runtime 的关系的散点图。(a 为红色，b 为蓝色)

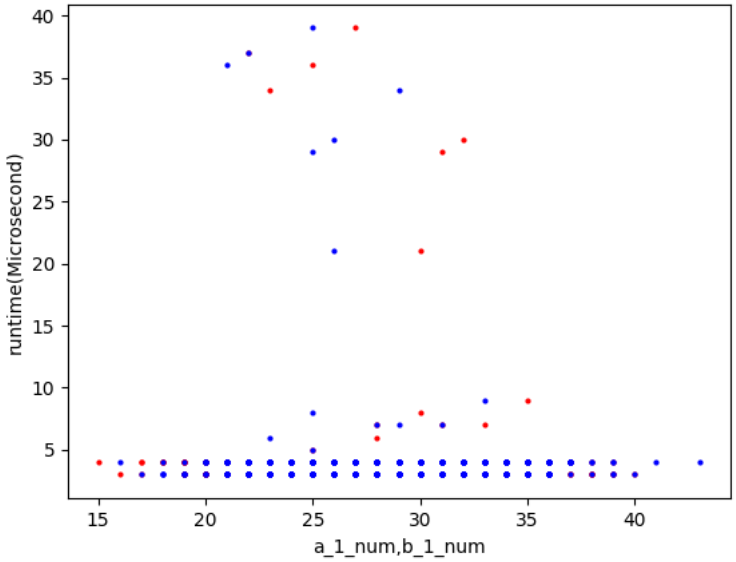
在计算 b_1_num 时，一开始发现如果计算 multimod_p3 的 b_1_num 就一切正常，但是如果计算 multimod_p2 的 b_1_num，就会全部都是零。一开始很懵，后来想到既然 p3 正常，那就说明只和 multimod 函数本身有关。而且把 a，b，m 写入文件的操作，都是在运行完函数之后，所以就是 p2 中对 b 的右移操作使 b 最后是 0。最后只需在运行函数前用 tempb 记录 b 的原值即可。

multimod_p1(multimod_p1 运行时间过于长，为得到更多测试数据，使用较小测试数)

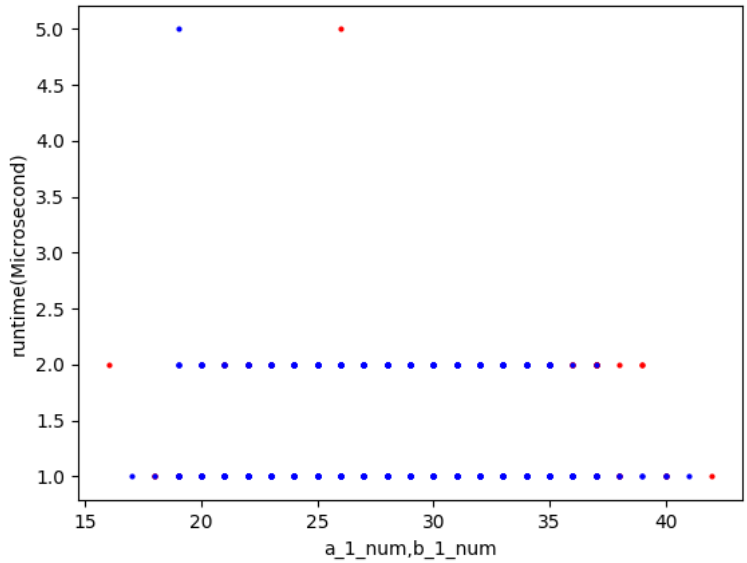


从散点图来看，multimod_p1 中 a，b 中 1 的个数与运行时间，在总体上呈正比关系。但是并不明显。从 p1 的函数实现思考，p1 的复杂度与 a，b 的大小有关。而 a，b 中 1 的数目越多，a，b 的大小有较大概率更大。但是还是有很大可能性出现这种情况：1 数目少的 a，b 本身大小却更大。所以，a，b 中 1 的个数与运行时间并没有呈现出很明显的关系，但是总体来看是呈正比的。

multimod_p2



multimod_p3



在 multimod_p2 和 p3 中，a，b 中 1 的个数与运行时间基本没有关系。原因分析，在 p2 中，循环次数的多少只与二进制数的位数相关，而不是和 1 的个数相关。循环内的操作也不会因为 1 的个数的变化而受明显影响。在 p3 中，算法的复杂度 O（1），自然不会受到二进制数中 1 的个数的影响。