Lab3实验报告

181240035
刘春旭
2019.12.23

> 在搜索`perf`工具的时候发现了这篇文章，感觉写的不错。

1. 命令行工具

1. 解析命令行参数，使得运行 `perf fn` 能运行统计 `fn` 函数的运行时间，例如 `perf dummy` 统计 `dummy` 函数的运行时间。

```
//解析命令行参数需要使用到 main 函数传进来的 argc 和 argv 参数;
//写一个switch来进行判断:
//...
case 4: //./perf -r 10000 dummy
    memcpy(funcname, argv[3], sizeof(funcname));
    rounds = atoi(argv[2]);
    break;
//...
```

2. 可以用 `-r` 命令行选项指定运行次数，例如 `perf -r 10000 simple_loop` 会将 `simple_loop` 运行 10,000 次。默认只运行一次。

```
这一步可以通过判断 argc 来做;
```

对于时间函数的选择（获取精确时间）：

1. 一开始用了clock()函数，但后来发现它定义的所谓'clock tick'是C/C++的一个基本计时单位，硬件滴答1000下是1s，也就是说，每过1ms，调用这个函数返回的值加一 DESCRIPTION The clock() function returns an approximation of processor time used by the program.
2. 故切换成：gettimeofday(), 它可以精确到微秒，在lab2中测试用的就是它；但是精度仍然不够测试出dummy的运行时间，只有偶尔它才会输出一个1us的数据；

```
CPU timeused: 0.000000
CPU timeused: 0.000000
CPU timeused: 0.000000
CPU timeused: 0.000000
CPU timeused: 0.000000
CPU timeused: 0.000001
CPU timeused: 0.000000
CPU timeused: 0.000000
```

（使用gettimeofday()函数）

3. 然后再切换成能够显示纳秒的函数：`clock_gettime()`，它的函数中定义的用来表示时间的结构体相比于`gettimeofday()`，两个`long`型变量分别是`tv_sec`和`tv_nsec`，所以它能够精确到纳秒；

```
CPU timeused: 37.000000 ns
CPU timeused: 38.000000 ns
CPU timeused: 38.000000 ns
CPU timeused: 37.000000 ns
CPU timeused: 37.000000 ns
CPU timeused: 38.000000 ns
CPU timeused: 38.000000 ns
```

（使用clock_gettime()函数）

4. 如果时间还不够精确的话，其实还可以用一个内联汇编的指令`rdtsc`，它可获取一个64位的CPU时钟周期数，经过交流，我的一个朋友就用了这个这条指令，比较有特征，为了避免抄袭嫌疑（其实是自己懒得写内联汇编了qwq），就不进行尝试了，需要用到的时候再查吧；

3. 把收集到的运行时间信息以用户友好的形式表示出来——这可以是一个生成的图片 (png, svg 等)，也可以直接输出到终端上。

> 等有时间一定想办法改成图片显示（`flag`）

2. 评估`multimod`实现的性能差异

因为我们要保证结果正确，所以由lab1的结果，生成数据的范围控制在52位以内；

1. 首先比较1的个数对计算时间的影响：

   生成测试文件的方法：`python`的字符串拼接；

   生成10w组如下的计算表达式：

```
6   fffffffff ffffff fffffff
7   fffff fffffff fffffffffffff
8   ffffffff ff fffffffffffff
9   fffffffff fffff fffff
10  ffffffffff fffffffff fffffff
11  fffffffffffffff ff fffffffffffff
12  fffffffffffff fffffffffff ffff
13  ffff fffffffff fffffffff
14  ffffff ffffffffff ffffffff
15  fffffffffff fffffffff ff
16  ffffffffff fffffffffffff fffffffff
```

   测量5次取平均：

> `multimod_p1`: 29254935ns
>
> `multimod_p2`:6084496ns
>
> `multimod_p3:`2647314ns

2. 再比较a, b, m的大小对时间的影响：

2.1 当a, b, m的大小都控制在2^{51}到2^{52}时（以下均为8次平均结果）：

```
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p1
Total time used: 44123423.400000ns
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p2
Total time used: 6059741.400000ns
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p3
Total time used: 2191711.000000ns
```

2.2 当a在2^{51}到2^{52}之间，b，m控制在$0-100$之间时:

```
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p1
Total time used: 19406636.000000ns
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p2
Total time used: 6205553.400000ns
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p3
Total time used: 2265719.600000ns
```

发现：`p2`,`p3`基本没变，`p1`优化了一倍

2.3 当m在2^{51}到2^{52}之间，a, b控制在$0-100$之间时:

```
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p1
Total time used: 9392915.800000ns
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p2
Total time used: 2852667.600000ns
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p3
Total time used: 2275428.400000ns
```

发现：`p1`时间增加一倍，`p2`时间减少一倍，`p3`基本不变

2.4 当a, b在2^{51}到2^{52}之间，m控制在$0-100$之间时:

```
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p1
Total time used: 41825748.000000ns
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p2
Total time used: 5898486.800000ns
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p3
Total time used: 2245833.400000ns
```

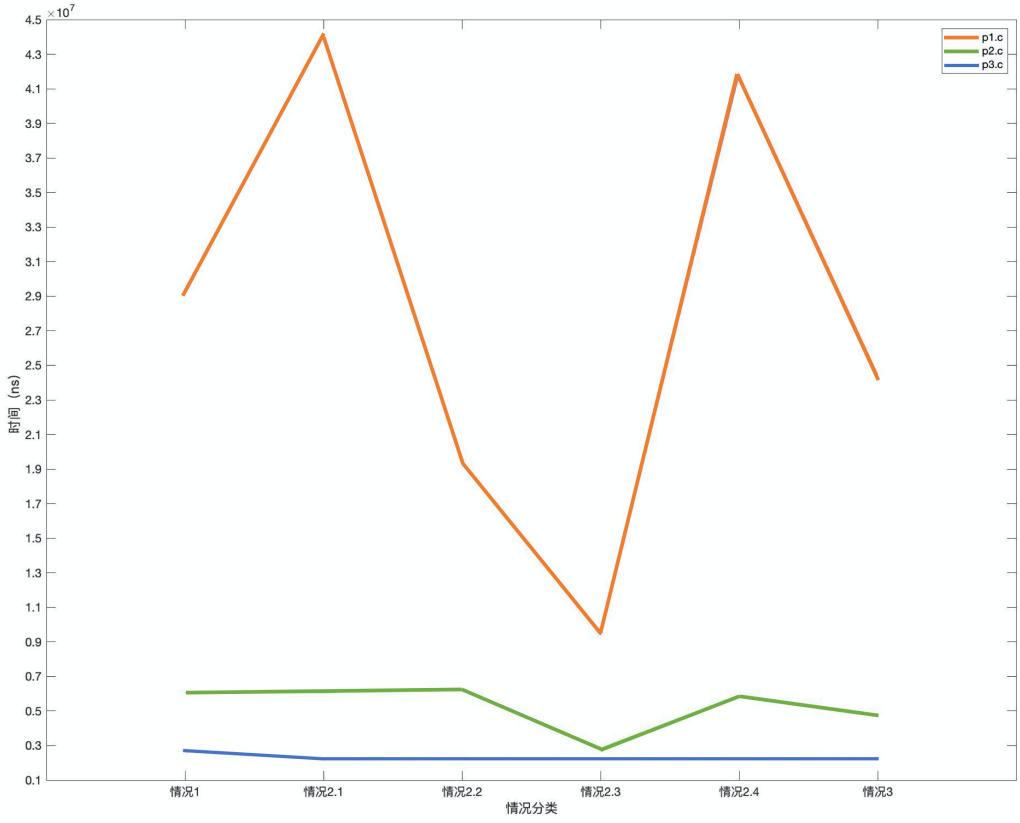
发现：基本和2.1中情况类似

3. 其他情况：0的个数对计算时间的影响，数据范围与1中一致：

```
317    1000000000000000 10 1000000000000
318    10000000000 10000 100000000
319    1000000 100 100000000000
320    1000000000000000 100000000000 10000000000
321    1000000000000000 10000000000000 10000000
322    10000000 100000000000 100000000000
323    100000000000 10000 10000000000
324    100 100000 1000000000
325    10000 100000 1000000000000
326    10000000000000 1000000000000 10
327    1000 100000 10000000
328    10000000000 100000000000000 10000000000
329    1000 10000000000000 1000
330    10000000000 1000000000000 1000000000
```

```
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p1
Total time used: 24484082.000000ns
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p2
Total time used: 4770793.400000ns
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100000 multimod_p3
Total time used: 2294457.800000ns
```

4. 总结:



可以发现，无论数据如何变化，对`p3.c`的影响比较微弱，毕竟O(1)程序，简单运算确实没什么可受影响的，而对`p2.c`影响比较大的只有情况2.3，即m较大，a, b较小时，初步推测：因为m较大，取模操作相对节省时间了，所以此时`p1`,`p2`运行时间均有很大改善；然后对于`p1.c`，出了刚才说过的情况2.3，在情况2.1和情况2.4的时候运行时间明显很长，即a, b较大时，运行时间很长，初步推测：因为`p1.c`是用高精度实现的，所以做运算的两个数长度越长，分解的时候越费时间，所以运行时间较长。

附：发现了有趣的事情：

运行dummy的时间越来越少：

刚开始：

```
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100 dummy
CPU time used: 0 s 165 ns
CPU time used: 0 s 44 ns
CPU time used: 0 s 44 ns
CPU time used: 0 s 44 ns
CPU time used: 0 s 44 ns
CPU time used: 0 s 43 ns
CPU time used: 0 s 43 ns
CPU time used: 0 s 43 ns
CPU time used: 0 s 43 ns
CPU time used: 0 s 44 ns
CPU time used: 0 s 44 ns
CPU time used: 0 s 42 ns
CPU time used: 0 s 44 ns
```

后面几次：

```
CPU time used: 0 s 36 ns
CPU time used: 0 s 35 ns
CPU time used: 0 s 35 ns
CPU time used: 0 s 35 ns
CPU time used: 0 s 37 ns
CPU time used: 0 s 36 ns
CPU time used: 0 s 36 ns
CPU time used: 0 s 36 ns
CPU time used: 0 s 36 ns
CPU time used: 0 s 37 ns
chauncey@debian:~/ics-workbench/perf$
```

但是运行hello的时候除了第一次以外，好像是运行时间周期性变长，再变短

```
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100 print hello
hello
CPU time used: 0 s 28962 ns
hello
CPU time used: 0 s 1453 ns
hello
CPU time used: 0 s 1338 ns
hello
CPU time used: 0 s 1315 ns
hello
CPU time used: 0 s 1304 ns
hello
CPU time used: 0 s 1296 ns
hello
CPU time used: 0 s 1291 ns
hello
CPU time used: 0 s 1292 ns
hello
CPU time used: 0 s 1292 ns
hello
CPU time used: 0 s 1292 ns
```

```
CPU time used: 0 s 1379 ns
hello
CPU time used: 0 s 1400 ns
hello
CPU time used: 0 s 3927 ns
hello
CPU time used: 0 s 4007 ns
hello
CPU time used: 0 s 3714 ns
hello
CPU time used: 0 s 3642 ns
hello
CPU time used: 0 s 3634 ns
hello
CPU time used: 0 s 3668 ns
hello
CPU time used: 0 s 3618 ns
hello
CPU time used: 0 s 3647 ns
hello
CPU time used: 0 s 3742 ns
```

```
CPU time used: 0 s 1356 ns
hello
CPU time used: 0 s 1360 ns
hello
CPU time used: 0 s 1365 ns
hello
CPU time used: 0 s 1299 ns
hello
CPU time used: 0 s 1360 ns
hello
CPU time used: 0 s 3662 ns
hello
CPU time used: 0 s 3796 ns
hello
CPU time used: 0 s 3689 ns
hello
CPU time used: 0 s 3733 ns
chauncey@debian:~/ics-workbench/perf$
```

但是切换到simple_loop的话，这种规律似乎又变了，第一次加载程序的时间不是最长的，然后总的时间变化趋势也是逐渐变短：

```
chauncey@debian:~/ics-workbench/perf$ ./perf-64 -r 100 simple loop
CPU time used: 0 s 2025364 ns
CPU time used: 0 s 1930892 ns
CPU time used: 0 s 3543018 ns
CPU time used: 0 s 3803212 ns
CPU time used: 0 s 2709996 ns
CPU time used: 0 s 1719830 ns
CPU time used: 0 s 1759218 ns
CPU time used: 0 s 1752402 ns
CPU time used: 0 s 2061306 ns
CPU time used: 0 s 1833855 ns
CPU time used: 0 s 1998061 ns
CPU time used: 0 s 1880970 ns
CPU time used: 0 s 2091248 ns
CPU time used: 0 s 1724221 ns
```

```
CPU time used: 0 s 1468828 ns
CPU time used: 0 s 1385611 ns
CPU time used: 0 s 1370763 ns
CPU time used: 0 s 1367023 ns
CPU time used: 0 s 1404770 ns
CPU time used: 0 s 1467880 ns
chauncey@debian:~/ics-workbench/perf$
```

推测：可能是CPU的频率控制系统的问题，在集中进行大量计算的时候，CPU的频率被调高，所以第一次运行时，需要的时间会比较慢，而后面越来越快