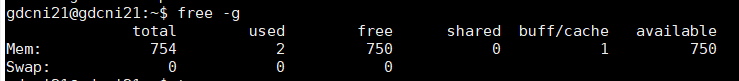
**【实验环境】**

操作系统：Ubuntu 16.04

CPU型号：Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz

物理CPU个数：4，每个物理CPU中12 个core (即核数)

内存：754G



【**实验结果**】FIB中只插入实表项

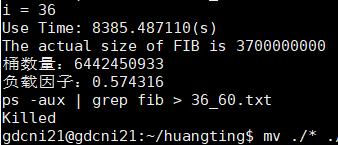
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| FIB表规模 | 1亿 | 2亿 | 3亿 | 4亿 | 5亿 | 6亿 | 7亿 | 8亿 | 9亿 | 10亿 |
| 运行时间  (s) | 187.58 | 333.40 | 485.30 | 641.97 | 798.94 | 964.09 | 1128.84 | 1298.95 | 1472.73 | 1649.75 |
| 实际生成  FIB表大小 | 1亿 | 2亿 | 3亿 | 4亿 | 5亿 | 6亿 | 7亿 | 8亿 | 9亿 | 10亿 |
| 内存消耗(%MEM) | 10.3% | 12.7% | 15.1% | 17.5% | 20.0% | 22.4% | 24.8% | 27.2% | 29.7% | 32.1% |
| CPU占用  (%CPU) | 99.5% | 99.2% | 99.0% | 100% | 98.9% | 98.7% | 98.7% | 100% | 98.6% | 100% |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| FIB表规模 | 11亿 | 12亿 | 13亿 | 14亿 | 15亿 | 16亿 | 17亿 | 18亿 | 19亿 | 20亿 |
| 运行时间  (s) | 1835.58 | 2026.89 | 2217.44 | 2419.82 | 2621.55 | 2832.78 | 3048.07 | 3264.83 | 3491.04 | 3723.98 |
| 实际生成  FIB表大小 | 11亿 | 12亿 | 13亿 | 14亿 | 15亿 | 16亿 | 17亿 | 18亿 | 19亿 | 20亿 |
| 内存消耗(%MEM) | 34.5% | 36.9% | 39.4% | 41.8% | 44.2% | 46.6% | 49.1% | 51.5% | 53.9% | 56.3% |
| CPU占用  (%CPU) | 98.5% | 100% | 100% | 100% | 98.4% | 98.4% | 98.3% | 98.4% | 98.3% | 100% |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| FIB表规模 | 21亿 | 22亿 | 23亿 | 24亿 | 25亿 | 26亿 | 27亿 | 28亿 | 29亿 | 30亿 |
| 运行时间  (s) | 3953.18 | 4186.63 | 4424.74 | 4670.63 | 4925.64 | 5182.74 | 5445.06 | 5714.59 | 5990.76 | 6271.49 |
| 实际生成  FIB表大小 | 21亿 | 22亿 | 23亿 | 24亿 | 25亿 | 26亿 | 27亿 | 28亿 | 29亿 | 30亿 |
| 内存消耗(%MEM) | 58.8% | 61.2% | 63.6% | 66.0% | 68.5% | 70.9% | 73.3% | 75.7% | 78.2% | 80.6% |
| CPU占用  (%CPU) | 100% | 98.3% | 98.2% | 98.2% | 98.2% | 98.2% | 98.2% | 98.1% | 98.1% | 98.1% |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| FIB表规模 | 31亿 | 32亿 | 33亿 | 34亿 | 35亿 | 36亿 | 37亿 | 38亿 | 39亿 | 40亿 |
| 运行时间  (s) | 6557.86 | 6850.01 | 7147.81 | 7451.45 | 7760.69 | 8077.09 | 8385.48 |  |  |  |
| 实际生成  FIB表大小 | 31亿 | 32亿 | 33亿 | 34亿 | 35亿 | 36亿 | 37亿 |  |  |  |
| 内存消耗(%MEM) | 83.0% | 85.4% | 87.9% | 90.3% | 92.7% | 95.5% | 99.4% |  |  |  |
| CPU占用  (%CPU) | 98.1% | 98.1% | 98.1% | 98.1% | 98.0% | 67.4% | 48.1% |  |  |  |

【**实验分析**】



当FIB标识规模扩大到37亿时，内存占用达到99.4%，由于剩余内存不够用，程序被killed，结束执行。

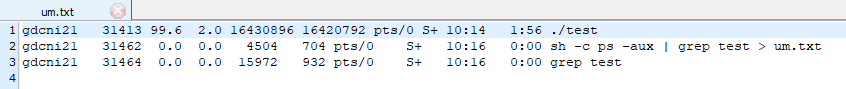
实验发现每新增1亿个FIB表项，平均增加2.45%的内存占用，也就是750\*0.0245=18.375G，则每新增一个表项的内存占用为183.75B，除了计算得出的hash表的每个表项的字节数为120B， 额外的字节数为unordered\_map内部结构的内存占用。unordered\_map通过链表解决哈希冲突问题，每个bucket都需要存储一个链表头指针，链表中每一项都需要存储一个next指针。

由于每新增1亿个FIB表项，平均增加2.45%的内存占用，因此该服务器理论上支持100% / 2.45% = 40.81亿规模的FIB表。但在生成FIB哈希表之前，base\_FIB的生成和程序其他变量或数据结构所占用内存为8%，剩余92%的内存可用于存储FIB表前缀名，因此能支持92% / 2.45% = 37.55亿FIB表规模。

**测试unordered\_map：**

typedef unordered\_map<std::string,std::string> umap;

键值对为string类型，规模为1亿时的内存占用如下所示：



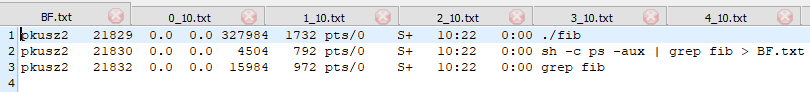
750 \* 2.0% = 15G，则每个hash表项的实际内存占用为150B，而每个表项的key和value所占字节数分别为32+17，32+17，共98B，额外的内存占用52B为unordered\_map内部数据结构。这与上面的实验结果是相符的。

**内存占用测试：**

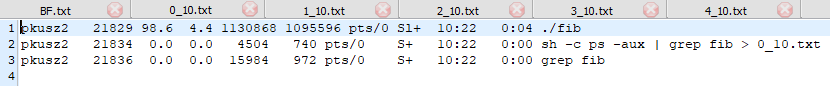
**服务器pkusz2 MAX=1e7 base=1e6 M=10**

BinarySearch（哈希表+前缀树+普通二分查询）：每新增1百万占用0.7%的内存

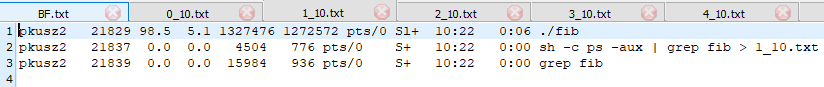
BF



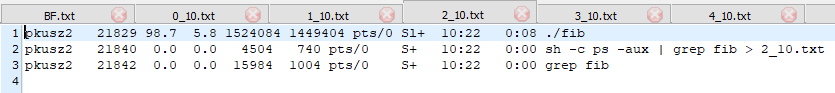
1百万



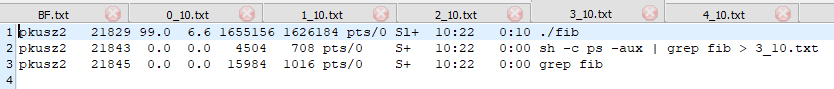
2百万



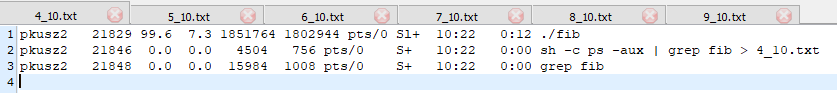
3百万



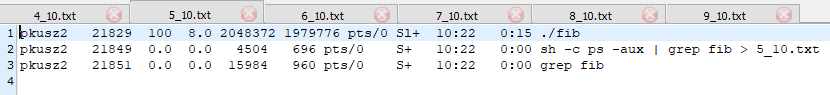
4百万



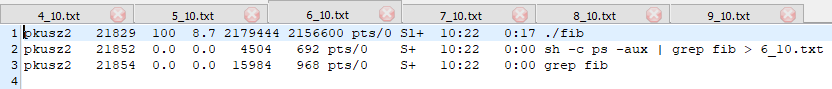
5百万



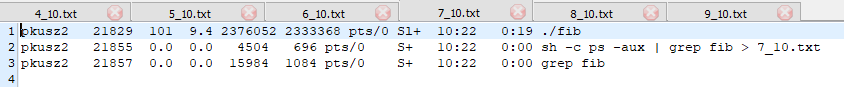
6百万



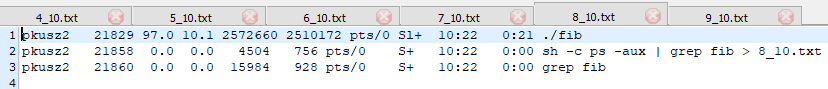
7百万



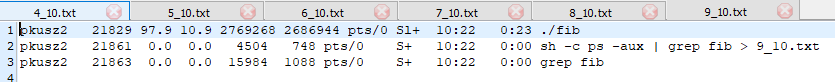
8百万

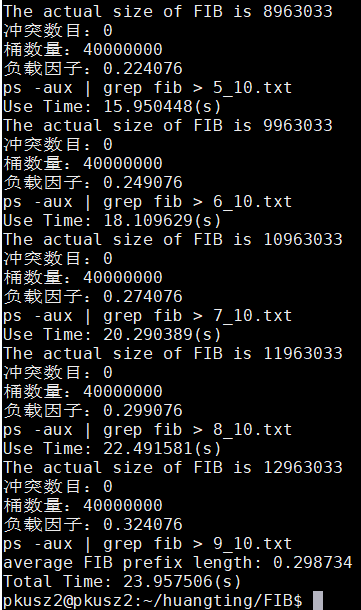
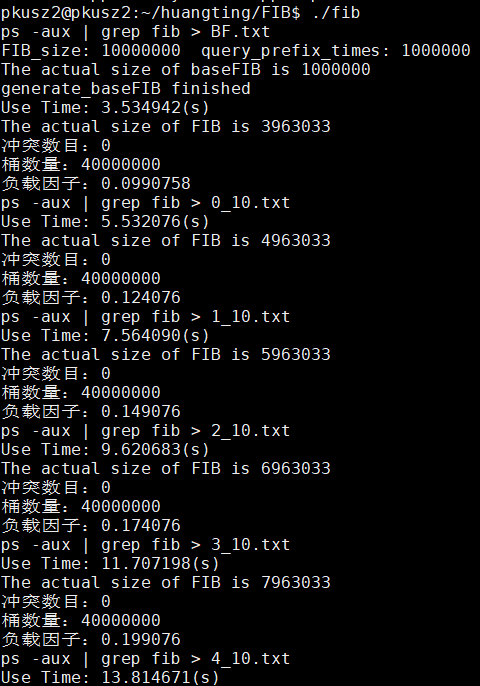


9百万



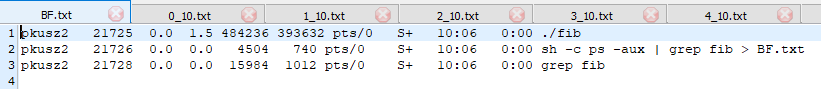
10百万



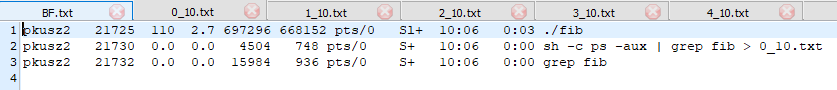


CBF-HT（布隆过滤器+哈希表+内存优化二分查询）：每新增1百万占用0.3%的内存

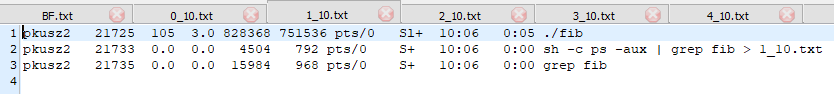
BF



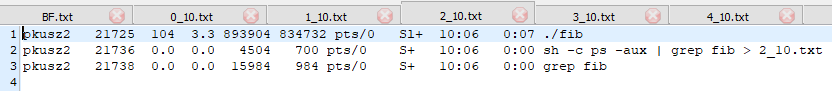
1百万



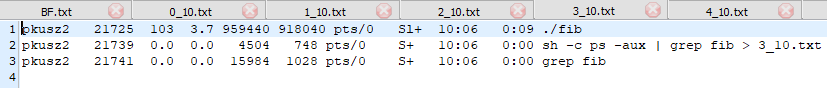
2百万



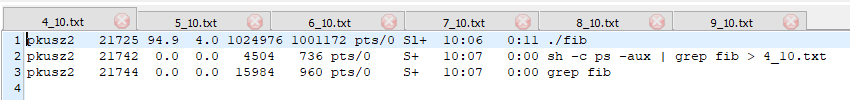
3百万



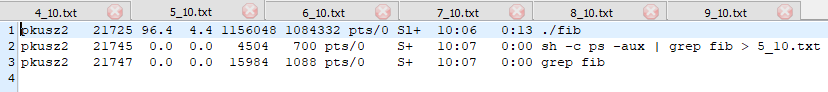
4百万



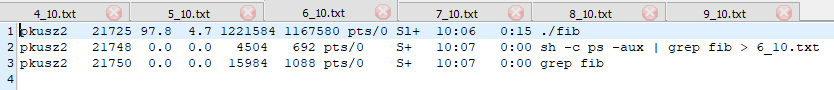
5百万



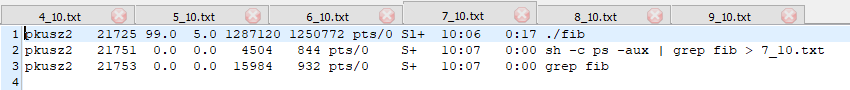
6百万



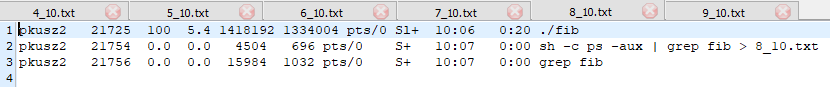
7百万



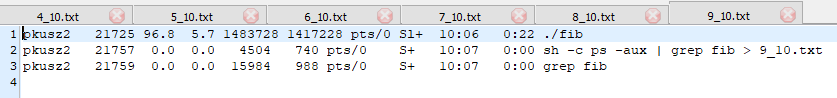
8百万

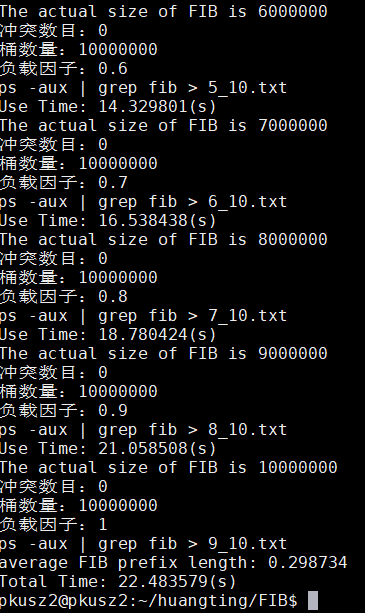
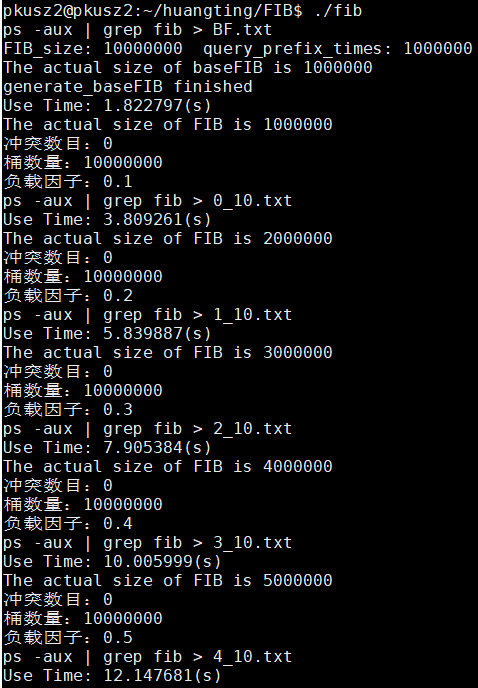


9百万



10百万

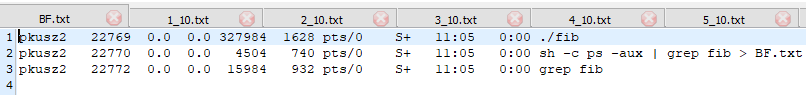




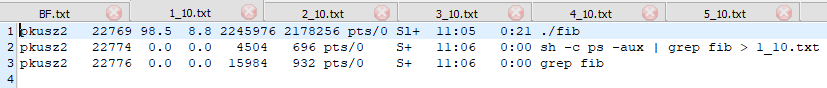
**服务器pkusz2 MAX=1e7 base=1e7 M=1**

BinarySearch（哈希表+前缀树+普通二分查询）：每新增1百万占用2.65%的内存

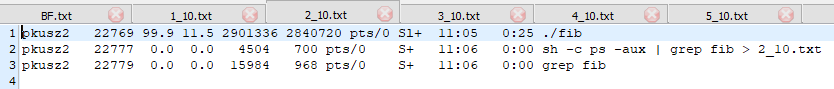
BF



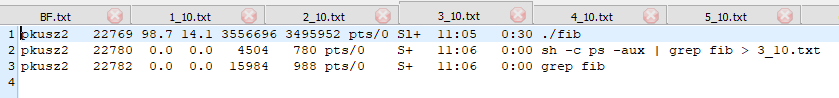
1百万



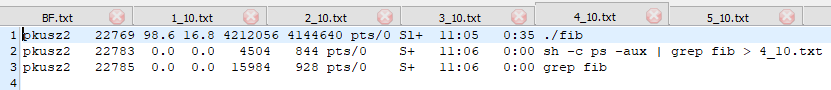
2百万



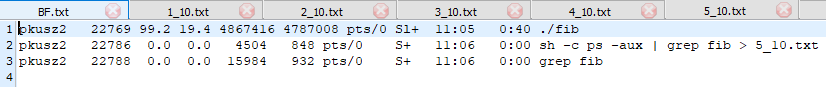
3百万



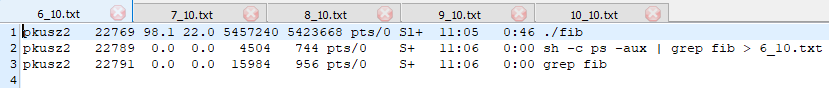
4百万



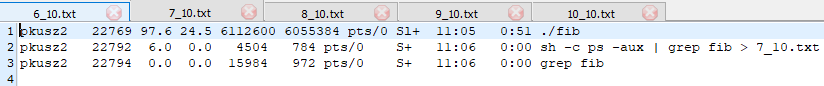
5百万



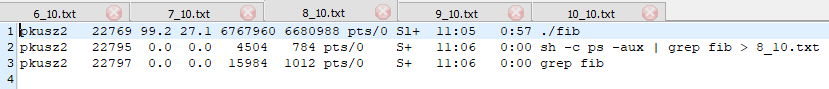
6百万



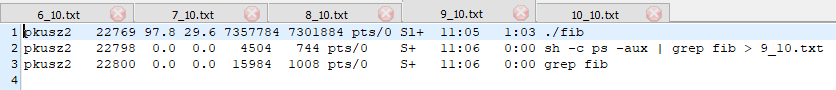
7百万



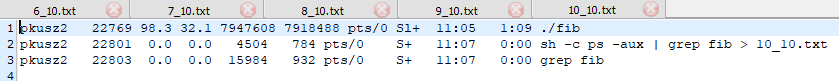
8百万

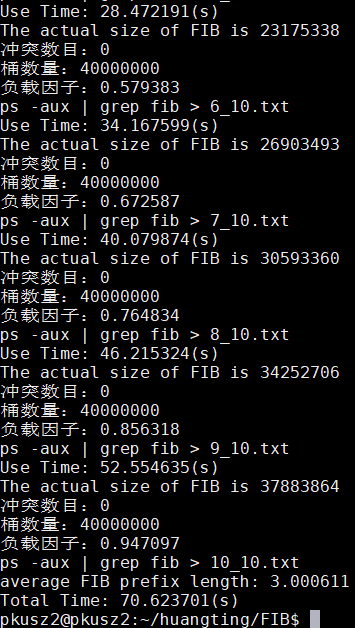
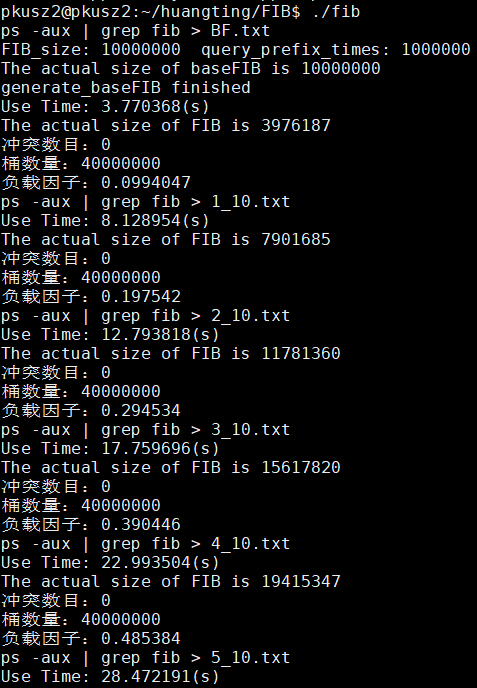


9百万



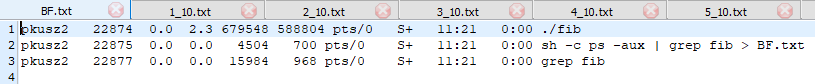
10百万



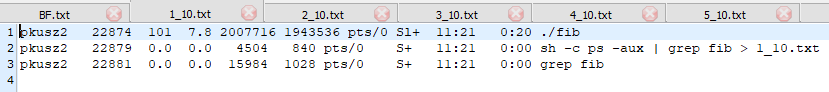


CBF-HT（布隆过滤器+哈希表+内存优化二分查询）：每新增1百万占用0.35%的内存

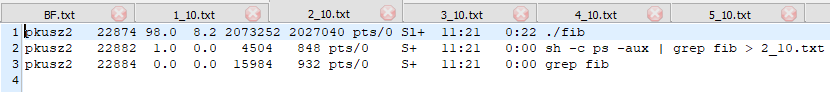
BF



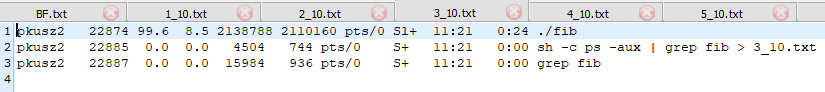
1百万



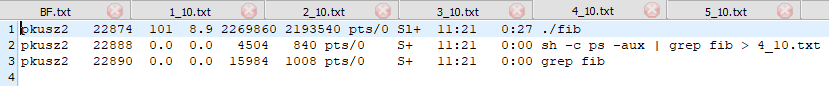
2百万



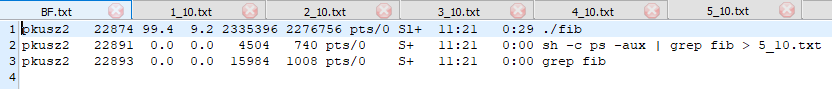
3百万



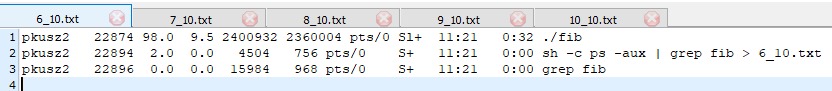
4百万



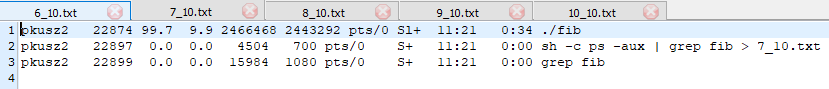
5百万



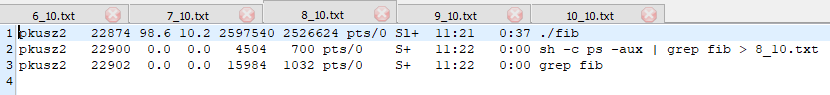
6百万



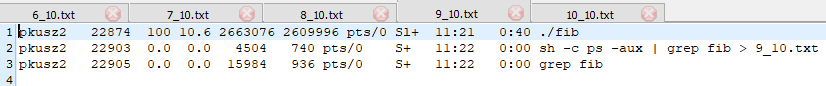
7百万



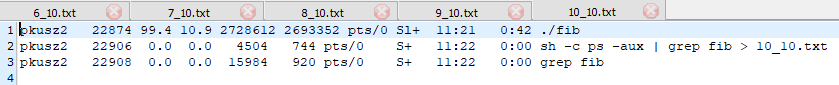
8百万

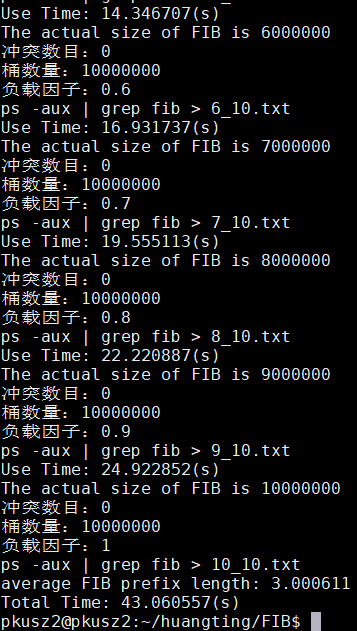
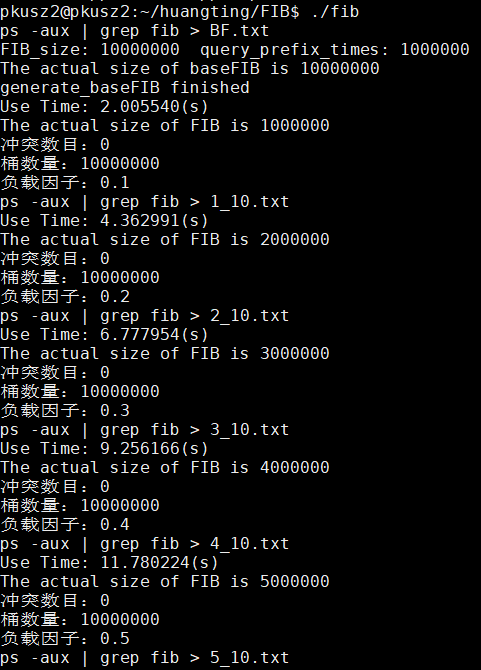


9百万



10百万





**在服务器环境下HPTvsCBF-HT能支持的FIB规模测试：**

关于在给定服务器环境下，进行HPT、CBF-HT能支持的最大FIB规模测试，主要分两步进行，第一步先生成大量名字标识供算法使用；第二步再具体测试两种算法分别能支持多大规模的FIB。

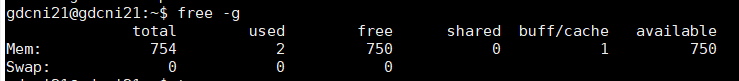
【实验环境】

操作系统：Ubuntu 16.04

CPU型号：Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz

物理CPU个数：4，每个物理CPU中12 个core (即核数)

可用内存：750G



**一、名字标识的生成**

生成名字有两种方法：

1）循环随机生成

·优点：可保证随机性

·缺点：生成速度慢，效率低。当生成的名字规模越来越大时，每新生成的一个名字与当前已存在的名字冲突的概率就越大，为了保证生成的名字与现有的名字不相同，就需要循环重复生成，导致生成速度很慢。

2）批量生成

·优点：生成速度快，效率高

·缺点：无法保证随机性。

5个名字：{/s1/s2/s3/s4、/s2/s3/s4/s5、/s3/s4/s5/s6、/s4/s5/s6/s7、/s5/s6/s7/s8}

ht0 ht1 … ht8 ht9

/ht0/s1/s2/s3/s4 /ht1/s1/s2/s3/s4 … /ht8/s1/s2/s3/s4 /ht9/s1/s2/s3/s4

/ht0/s2/s3/s4/s5 /ht1/s2/s3/s4/s5 … /ht8/s2/s3/s4/s5 /ht9/s2/s3/s4/s5

/ht0/s3/s4/s5/s6 /ht1/s3/s4/s5/s6 … /ht8/s3/s4/s5/s6 /ht9/s3/s4/s5/s6

/ht0/s4/s5/s6/s7 /ht1/s4/s5/s6/s7 … /ht8/s4/s5/s6/s7 /ht9/s4/s5/s6/s7

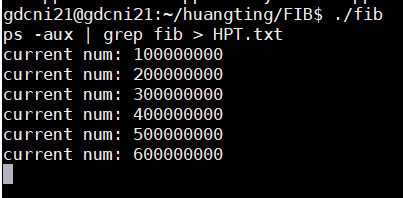
/ht0/s5/s6/s7/s8 /ht1/s5/s6/s7/s8 … /ht8/s5/s6/s7/s8 /ht9/s5/s6/s7/s8

共50个名字。

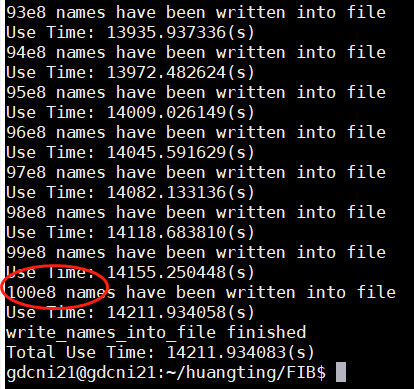
先使用循环随机生成方法生成一定基数的base\_dataset，基于基数数据集，通过添加不同前缀或后缀的方式来批量生成数据集。至于是前缀还是后缀，需要考虑一下。由于添加后缀会导致批量生成的名字间的前缀相同，这会导致实验结果的不准确性。而添加前缀能避免这个问题。

附关键代码及实验结果：

**循环随机生成**

****

**批量生成并写入文件**

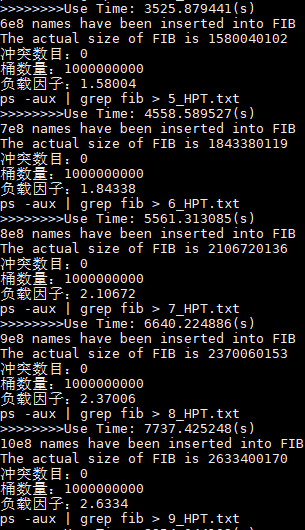
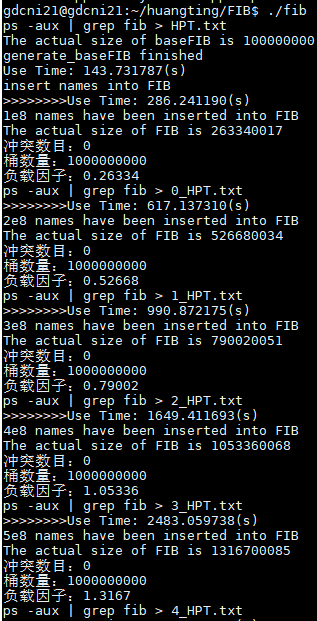
****

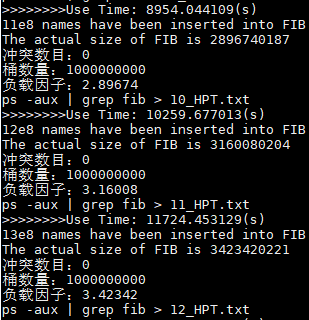
****

**二、测试HPTvsCBF-HT在服务器环境下能支持的最大规模FIB**

>> HPT

当插入16亿规模名字时，内存占用达到99.6%，由于剩余内存不够用，程序被killed，结束执行。





>> CBF-HT

当插入79亿规模名字时，内存占用达到99.7%，由于剩余内存不够用，程序被killed，结束执行。

