南開大學
Nankai University

计算机学院

并行程序设计第四次实验报告

口令猜测 Generate() 函数多线程

姓名：葛明宇

学号：2312388

专业：计算机科学与技术

2025 年 5 月 29 日

目录

1 基础要求

1.1 问题阐述

论文 [1] 设计了 PCFG 并行化算法，如图1.1所示。

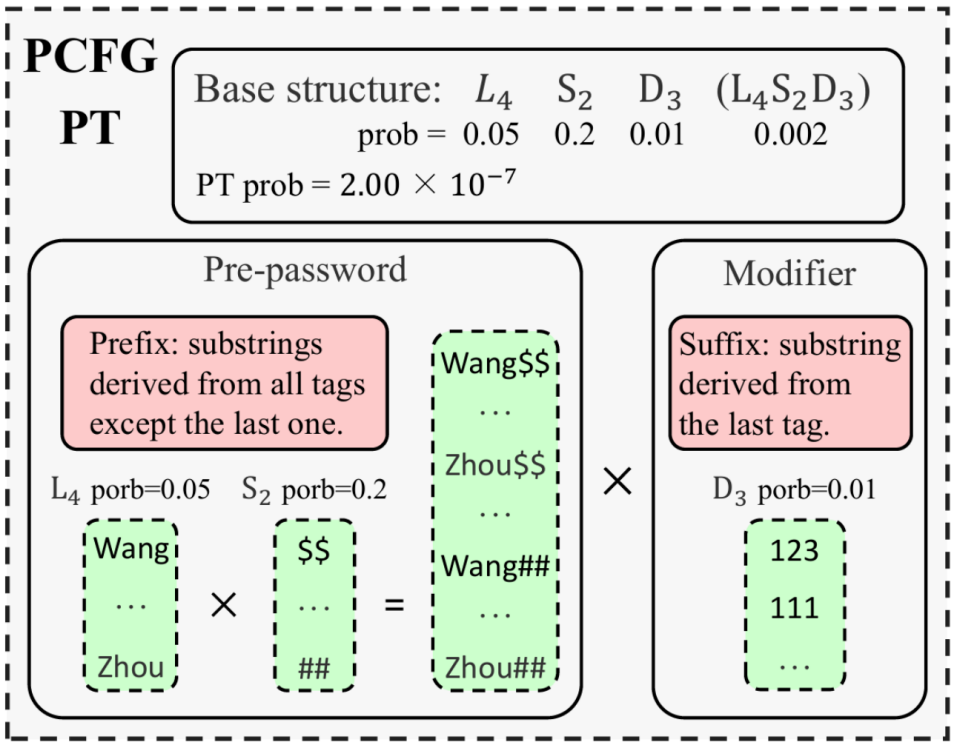

图 1.1: PCFG 并行化算法

并行思路为对 preterminal 中最后一个 segment 不进行初始化和改变。在 preterminal 从优先队列中弹出的时候，直接一次性将所有可能的 value 赋予这个 preterminal。在我们的代码框架中虽然使用了并行思路，但还是使用着串行的代码写法。本次实验目标即为对 Generate() 函数进行多线程并行化改造。原函数通过循环生成口令猜测，核心操作为 emplace_back() 向全局容器添加元素以及总猜测数自增操作。以下为关键代码片段：

```
//guessing.cpp
for (int i = 0; i < pt.max_indices[0]; i += 1)
    {
        string guess = a->ordered_values[i];
        guesses.emplace_back(guess);
        total_guesses += 1;
    }
```

```
//guessing.cpp
for (int i = 0; i < pt.max_indices[pt.content.size() - 1]; i += 1)
    {
        string temp = guess + a->ordered_values[i];
        guesses.emplace_back(temp);
```

```
            total_guesses += 1;
        }
```

1.2 并行设计与实现

1.2.1 并行分析

在处理从优先队列弹出的 ‘preterminal‘时，‘Generate()‘ 函数需将所有猜测口令的取值（value）“赋予”一个全局的 guesses 容器。这一过程在代码中体现为通过 ‘emplace_back()‘ 操作向容器添加元素，是性能优化的关键目标。为提升该操作的执行效率，我们尝试对其循环结构进行多线程改造，将 preterminal 生成的 value 均分为 num_thread 块，每一块单独使用一个线程执行。但初始测试结果未达预期。性能瓶颈主要源于两方面：其一，线程的频繁创建与销毁带来显著开销；其二，并行生成的局部结果需要进行全局整合，而这一过程存在线程安全隐患。

针对全局整合问题，我们评估了两种主要解决方案：(1) 同步整合策略。等待所有线程执行完毕后再统一合并局部结果。此方案虽能保证数据一致性，但会引入较长的线程等待时间。(2) 互斥锁机制。通过加锁保护共享资源，避免数据竞争。然而，锁的频繁获取与释放会消耗额外的系统资源，性能退化明显。

上述两种方案均会引入不可忽视的额外开销，因此我们提出以下优化策略：

(1) 线程资源池化：在 ‘PriorityQueue‘ 类中预先创建并维护固定数量（‘num_thread‘）的线程句柄（‘threadId‘）及对应的参数结构体（‘threadargs‘），避免重复创建销毁线程参数带来的开销。

(2) 局部结果隔离：为每个线程分配独立的局部结果容器（‘local_guesses‘），替代直接操作全局容器。同时，将统计计数操作从线程内部移至外部，通过直接累加区间长度（如 ‘pt.max_indices[0]‘）的方式替代逐次递增（‘total_guesses += 1‘），彻底消除自增操作的线程同步需求。

1.2.2 pthread 多线程实现

有了以上的分析，不难编写出 pthread 多线程的代码。

(1)PCFG.h

定义了 ThreadArgs 结构体,用于保存线程参数,特别是维护了 vector<string> 类型的 local_guesses。除此之外还宏定义了 num_thread 的值，2、4、8 线程版本的实现只需在这里修改线程数即可。

```
//PCFG.h
#define num_thread 4

struct ThreadArgs {
    segment* seg;           // 最后一个segment的指针
    string guess;          // 已实例化的前缀
    int start;          // 起始索引
    int end;            // 结束索引
    vector<string> local_guesses;
};
```

同时，在 Priority 类中定义了 num_thread 个 pthread_t 类型的 threads 和 ThreadArgs 类型的 threadargs。

```
//PCFG.h
```

```
    class PriorityQueue
    {
    public:
        ...

        pthread_t threads[num_thread];
        ThreadArgs threadargs[num_thread];

};
```

(2)guessing.cpp

定义了 guess_thread 函数,用于生成一个线程的任务。注意到只需要对每个线程中的 local_guesses 进行操作，也不需要最后再将每一个线程结果赋值给 guesses。

```
    //guessing_pthread.cpp
    void* guess_thread(void* arg) {

        ThreadArgs* args = static_cast<ThreadArgs*>(arg);
        for (int i = args->start; i < args->end; ++i) {
            args->local_guesses.emplace_back(args->guess+args->seg->ordered_values[i]);
        }
        return nullptr;
}
```

对 Generate() 函数内部两条循环具体的并行化实现。包括线程创建前的准备，线程的创建和等待线程的部分。使用 reminder 变量解决生成 value 的数量不被整除 num_thread 的情况，使得最终每一个线程的 local_guesses 中的元素数更加均衡。

```
    //guessing_pthread.cpp
    int index=pt.max_indices[0];
    int chunk_size = index / num_thread;
    int remainder = index % num_thread;

    int start = 0;
    int end=0;

    for (int i = 0; i < num_thread; ++i) {
        end = start + chunk_size + (i < remainder ? 1 : 0);

        threadargs[i].seg = a;
        threadargs[i].guess = "";
        threadargs[i].start = start;
        threadargs[i].end= end;

        pthread_create(&threads[i], nullptr, guess_thread, &threadargs[i]);

        start=end;
    }

```

```
    // 等待线程完成
    for (int i = 0; i < num_thread; ++i) {
        pthread_join(threads[i], nullptr);
    }
    total_guesses+=pt.max_indices[0];
```

```
    //guessing_pthread.cpp
    //一个segment多线程版本代码
    int index=pt.max_indices[pt.content.size() - 1];
    int chunk_size = index / num_thread;
    int remainder = index % num_thread;

    int start = 0;
    int end=0;

    for (int i = 0; i < num_thread; ++i) {
        end = start + chunk_size + (i < remainder ? 1 : 0);

        threadargs[i].seg = a;
        threadargs[i].guess = "";
        threadargs[i].start = start;
        threadargs[i].end= end;

        pthread_create(&threads[i], nullptr, guess_thread, &threadargs[i]);

        start=end;
    }

    // 等待线程完成
    for (int i = 0; i < num_thread; ++i) {
        pthread_join(threads[i], nullptr);
    }

    total_guesses+=pt.max_indices[pt.content.size() - 1];
```

(3)main.cpp

用 local_guesses 代替 guesses 的使用。

```
    //main_pthread.cpp
    //多个segment多线程版本代码

    //MD5 Hash处理部分
    bit32 state[4];
    for (string pw : q.local_guesses)
    {
        MD5Hash(pw, state);
    }
```

```
//main_pthread.cpp
//多个segment多线程版本代码

//local_guesses代替guesses的使用
for (int i = 0; i < num_thread; i++) {
    q.threadargs[i].local_guesses.clear();
}
```

1.2.3 openMP 多线程实现

(1) 由于先完成了 pthread 版本的代码，openMP 版本的实现只需要对 guessing_thread.cpp 函数进行修改即可。当然，PCFG.h 中的全局 threads、ThreadArgs 结构体中 start 和 end、guessing_thread.cpp 中的 guess_thread() 函数在 openMP 版本中均是不需要的。

```
//guessing_openmp.cpp
//一个segment多线程版本代码
#pragma omp parallel num_threads(num_thread)
{
    int tid = omp_get_thread_num();
    #pragma omp for schedule(static)
    for(int i=0; i< index; ++i) {
        threadargs[tid].local_guesses.emplace_back(a->ordered_values[i]);
    }
}
total_guesses+=pt.max_indices[0];
```

```
//guessing_openmp.cpp
//多个segment多线程版本代码
#pragma omp parallel num_threads(num_thread)
{
    int tid = omp_get_thread_num();
    #pragma omp for schedule(static)
    for(int i=0; i< index; ++i) {
        threadargs[tid].local_guesses.emplace_back(guess + a->ordered_values[i]);
    }
}
total_guesses+=pt.max_indices[pt.content.size() - 1];
```

1.3 正确性验证

编写了多线程对应版本的 correctness_guess.cpp，用于验证多线程版本的正确性，输出的 Cracked 数量与串行版本的输出数量一致，均为 358217 条口令。将 pthread 版本与串行版本生成的这 358217 条口令分别输出到 output 文件夹下的 result_guess.txt 和 result_pthread.txt 中，并行版本头尾部分的口令位置与串行版本一致。但是中间的口令位置不同，是因为并行版本的生成顺序并不是按照优先队列的顺序生成的。如果想要在输出时也完全按照概率降序输出需要构建索引。

output > results_pthread.txt

200488	charlmae
200489	neuronas
200490	galajyan
200491	Florinda
200492	jlfkjlfk
200493	ladyjoan
200494	parishil
200495	jadelist
200496	lanalove
200497	yopisari
200498	tigulang
200499	samocapo
200500	waddells
200501	ilovekau
200502	montawat
200503	FOODTOWN
200504	wandapot
200505	clubbabe
200506	lokeloke
200507	larstama
200508	SPICEBOY
200509	realista
200510	rockface
200511	pagliuca
200512	petarina
200513	gbgopher
200514	mihakora
200515	mijailos

output > results_guess.txt

200488	bapartite
200489	rentianle
200490	gallagirl
200491	kuyamarlo
200492	spiderone
200493	PENADILLO
200494	addasweet
200495	patrasche
200496	andreacbz
200497	pacharebe
200498	YOURDADDY
200499	leondejud
200500	BONAPARTE
200501	oliviaisa
200502	dongjames
200503	akachable
200504	thienbinh
200505	victoriaa
200506	walesbest
200507	gigglegal
200508	babybonny
200509	iamfatfat
200510	JACKFLASH
200511	naulanono
200512	carlberal
200513	takeapeek
200514	jimaylwin
200515	bluedames

图 1.2: pthread 版本部分 Cracked 口令

当然，数量一致并不能说明结果的正确性。使用 compare.py 文件进一步进行 result_guess.txt 和 result_pthread.txt 的一致的验证。

compare.py > ...

```python
from collections import Counter

def compare_files(file1_path, file2_path):

    with open(file1_path, 'r') as file1:
        lines1 = [line.rstrip('\n') for line in file1]
    counter1 = Counter(lines1)

    with open(file2_path, 'r') as file2:
        lines2 = [line.rstrip('\n') for line in file2]
    counter2 = Counter(lines2)


    return counter1 == counter2

file_a = './output/results_guess.txt'
file_b = './output/results_pthread.txt'
result = compare_files(file_a, file_b)
print("文件内容一致" if result else "文件内容不一致")
```

问题 输出 调试控制台 终端 端口

```
PS D:\PCFG_framework> & D:/ProgramTools/Python/python.exe d:/PCFG_framework/
compare.py
文件内容一致
PS D:\PCFG_framework>
```

图 1.3: 进一步验证 pthread 多线程正确性

结果说明 pthread 版本与串行版本生成的这 358217 条口令相同，验证了 pthread 版本的正确性。

同样的操作验证了 openMP 版本的正确性。

2 实现加速

在基础实现中，尽管通过线程参数复用和局部结果隔离策略，解决了线程频繁构造销毁及全局数据整合的开销问题，但线程创建操作本身仍存在不可忽视的性能瓶颈。即使引入线程池机制减少线程生命周期管理成本，任务分配过程中的额外开销依然显著。由此引发对多线程适用场景的深层思考：多线程的性能优势源于并行处理任务的能力，但其资源消耗（如线程调度、同步机制等）可能抵消理论上的加速收益。当资源消耗超过并行处理带来的收益时，多线程反而会导致性能下降。因此，寻找资源消耗与并行收益的平衡点成为关键，这一思路推动了后续探索。

2.1 阈值机制

为攻克性能瓶颈，需引入阈值机制以实现多线程与串行算法的切换。该阈值本质上是资源消耗与多线程理论收益的平衡点——当 PT 生成的口令猜测数量超过此阈值时，多线程并行处理的收益将覆盖其资源开销，此时启用多线程；反之，当猜测数量低于阈值时，串行算法因避免了线程管理开销而更具效率。

2.1.1 代码实现

在 PCFG.h 的 Priority 类中，定义一个阈值 threshold。

```
    //PCFG.h
    class PriorityQueue
    {
    public:
        ...

        pthread_t threads[num_thread];
        ThreadArgs threadargs[num_thread];

        //新添的阈值
        const int threshold=10500;  //8线程:10500    4线程: 4000      2线程:2500
};
```

在 guessing_openmp.cpp 或 guessing_pthread.cpp 中，通过生成猜测口令数与阈值大小的判断控制执行逻辑。

```
    //guessing_openmp.cpp或guessing_pthread.cpp
    int index=pt.max_indices[0];

    if(index<=threshold){
        //一个segment串行版本代码
        ...
    }
    else{
        //一个segment多线程版本代码
        ...
```

```
    }
    total_guesses+=index;
```

```
    //guessing_openmp.cpp或guessing_pthread.cpp
    int index=pt.max_indices[pt.content.size() - 1];

    if(index<=threshold){
        //多个segment串行版本代码
        ...
    }
    else{
        //多个segment多线程版本代码
        ...
    }
    total_guesses+=index;
```

2.1.2 阈值影响因素

阈值的具体取值受多重因素影响：

(1) 多线程实现方式：底层线程库（如 pthread）与高层并行框架（如 OpenMP）的线程创建、同步机制不同，导致资源开销差异显著。(2) 编程接口特性：不同 API 对线程生命周期管理、负载均衡的支持程度会直接影响阈值边界。(3) 硬件平台架构：CPU 核心数、缓存容量、内存带宽等硬件参数决定了并行任务的实际执行效率与资源消耗上限。

2.1.3 测试方法论

首先通过测试阈值为 10，100，1000 •••••• 测试得到阈值的数量级。后面采用采用二分法动态缩小阈值范围，通过持续迭代逼近最优解。由于线程数与资源开销呈正相关，不同线程数的阈值理论上呈线性比例关系，可利用已知线程数的阈值（如 8 线程）推导其他线程数（如 2、4 线程）的阈值范围，减少测试复杂度。

2.1.4 阈值取值

经过实验测定，不同框架与线程数的阈值设定如下：

pthread：2 线程:2500，4 线程:4000，8 线程:10500

OpenMP：2 线程:1200，4 线程:1700，8 线程:2500

以 8 线程的 pthread 版本为例，以折线图展示阈值设定为 10^4 量级的过程。

图 2.4: 总猜测口令数为 1000 万的阈值测试

上述折线图确定阈值为 1000-100000 之间，之后又测试了阈值为 6500 和 65000 时的 Guess 时间，也比 10000 时的 Guess 时间高，初步认为阈值会在 10000 左右，量级为 10^4。

2.1.5 结果局限性

环境波动性：服务器执行时间存在波动，难以获得绝对精确的阈值。样本偏差：PT 分布在理论阈值的分布稀疏时将难以确定阈值。

2.2 样本分布与阈值关联分析

2.2.1 数据分布

统计了猜测口令数量的分布，每 1000 条口令作为一个区间，统计此区间上的 PT 个数。分布图如图1.3所示。

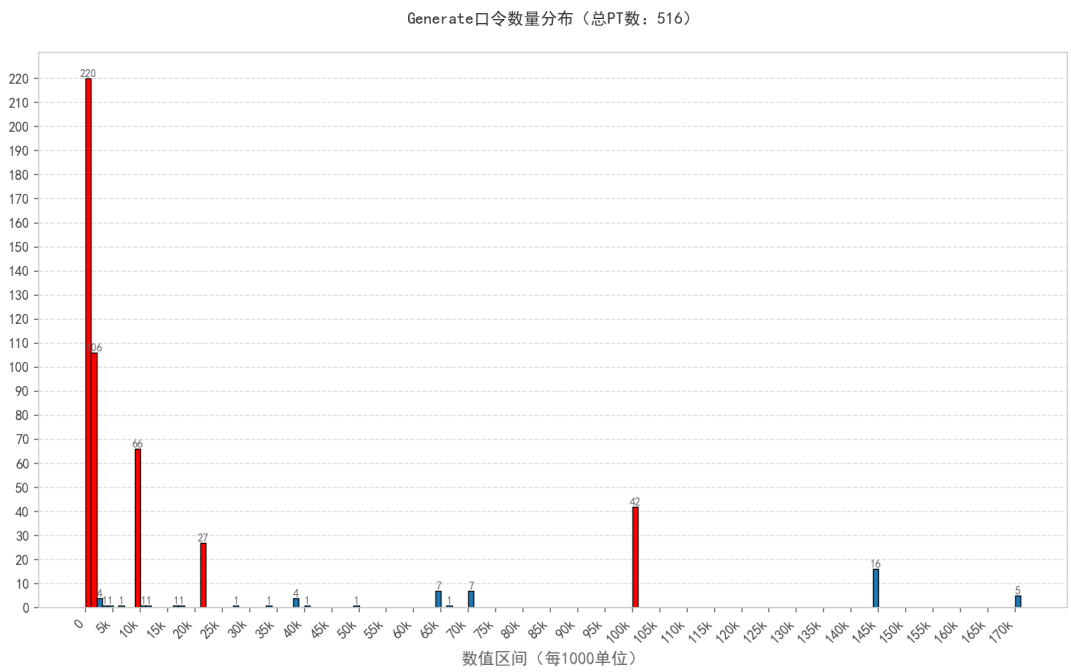

图 2.5: 总猜测口令数为 1000 万的 PT 分布

其中，红色的柱形是占总 PT 个数的比例超过 5% 的区间。

2.2.2 阈值设定的必要性

可以看到如果不进行阈值设置，对于多线程的资源开销由轻量级任务主导。[0,2000) 区间 PT 数占比 63%，多线程资源开销占比可估计为 63%，但总猜测数仅占 3%，串行处理耗时可忽略。花费了大量的多线程资源，加速占比极小的部分。而阈值的设定正好可以解决这一问题。

2.2.3 阈值设定的难度

数据分布具有部分区间稀疏性。[2000,10000) 和 [11000,21000) 区间 PT 数占比约 2%，异常的空旷，估计的 pthread 阈值又恰巧在这一区间，导致 pthread 阈值在该范围难以精确校准。OpenMP 因线程池等机制，阈值会更接近于 [0,2000) 的样本集中值，阈值也更好确定，而 pthread 需更严格的阈值控制。不过由于服务器测试时间波动，openMP 版本阈值也并不容易确定。

想要检测目前设定的阈值的合理性需要对总猜测数更大的情况进行测试，以此减小服务器波动影响。这一过程在总猜测数对性能影响的分析中会有所体现。

2.3 性能测试

2 线程、4 线程和 8 线程均能实现加速，这里只展示 8 线程的性能结果。在“总线程数对加速比影响”的部分会全部进行展示。

总猜测数为 1000 万、1 亿、10 亿的情况下，也均能实现加速，这里只展示 1000 万时的性能结果。在“总猜测数对加速比的影响”部分会全部进行展示。

注：表中不进行编译优化的数据只进行了一次，这是因为执行时间较长，测试时间波动的影响也比较小。O1 和 O2 优化均是在 5-10 次结果中取的平均值 (会舍去一些偏离中心值的结果，减小测试波动对结果的影响)。

pthread 多线程结果			
编译优化选项	串行 Guess 时间 (s)	多线程 Guess 时间 (s)	加速比
不编译优化	7.693	7.198	1.068
O1 优化	0.616	0.438	1.406
O2 优化	0.585	0.397	1.473
openMP 多线程结果			
编译优化选项	串行 Guess 时间 (s)	多线程 Guess 时间 (s)	加速比
不编译优化	7.693	7.059	1.090
O1 优化	0.616	0.357	1.725
O2 优化	0.585	0.301	1.943

表 1: 多线程实验测试结果

从上述结果中能够清晰地看到无论是在 pthread 多线程版本还是 openMP 多线程版本，无论是不进行编译优化、O1 优化还是 O2 优化，均能实现加速。且在 O1 和 O2 优化下的加速比超过了 1.4，在 openmp 版本的 O2 优化下甚至接近 2 的加速比。

2.4 结果分析

上述测试结果说明设定的阈值能够很好地将执行串行算法的部分与执行多线程算法的部分分开，在猜测口令数多时进行多线程加速，在猜测口令数少时使用串行算法起到了很好的作用。

从结果上验证了阈值解决 [0,2000) 区间，PT 数占比极高，但串行处理时间占比极小的问题 (特别是 pthread 版本，每个 PT 均需要创建线程和等待线程结束)。体现了设定阈值的必要性和优越性。

3 编译优化选项对加速比的影响

3.1 性能测试

多线程版本	不编译优化	O1 优化	O2 优化
pthread	1.068	1.406	1.473
openmp	1.090	1.725	1.943

表 2: 不同编译选项加速比结果

在数据表中，无论是 pthread 版本还是 openmp 版本，O2 的加速比最高，加速效果非常明显；O1 的加速比次之；不进行编译优化能够实现加速，但是效果不明显。

3.2 结果分析

为了解释这一现象，我构建了以下模型：阈值给定的同时，定义 s 和 t 均是关于编译优化选项 (O1、O2 等) 的函数，分别表示不同编译选项下 guess 时间缩减因子，简记为 s 和 t。s, s_1, s_2 分别表示串行不进行编译优化、O1 优化和 O2 优化下的 guess 时间缩减因子;t, t_1, t_2 分别表示并行不进行编译优化、O1 优化、O2 优化下的 guess 时间缩减因子。a、b、b_1 均是常数，表示 guess 时间。

对于串行算法，阈值左侧是串行 guess 时间 sa，阈值右侧是串行 guess 时间 sb。隐含了对于不同编译选项，阈值左右侧的比例一直是恒定的，也是符合直觉的。

对于并行算法，阈值左侧是串行 guess 时间 sa，阈值右侧是多线程 guess 时间 tb_1。

于是加速比可以由公式给出:

$$\frac{s(a+b)}{sa+tb_1}$$

接下来就可以解释为什么不进行编译优化，O1 优化和 O2 优化的加速比随着优化程度的增大而增大的原因。

O 优化下相对于 O 优化的加速比为：

$$\frac{\frac{s_2(a+b)}{s_2a+t_2b_1}}{\frac{s_1(a+b)}{s_1a+t_1b_1}} = \frac{a+\frac{t_1}{s_1}b_1}{a+\frac{t_2}{s_2}b_1}$$

由于相对加速比大于 1，故可得：

$$\frac{t_2}{s_2} < \frac{t_1}{s_1} \quad \Rightarrow \quad \frac{t_1}{t_2} > \frac{s_1}{s_2}$$

上述推导说明从 O1 优化到 O2 优化，对于多线程部分的优化比对串行部分的优化更明显。O1 优化相对于不进行编译优化也是同理。

既然知道了是由于对多线程的优化效果比对串行部分的优化效果更明显，那么接下来进行具体的分析。

本次需要改进的代码就是一个循环以及内部不断的 emplace_back() 操作。

对于串行代码，O1 优化时，编译器会将循环条件外移，对于 emplace_back() 的对象直接构造。O2 优化时，编译器会进行循环展开，可仍然是对 guesses 动态数组进行操作，存在着很大的数据依赖。

对于并行代码，O1 优化时，编译器会将循环条件外移，对于 emplace_back() 的对象直接构造。同时将各线程的地址缓存在寄存器中，避免重复计算线程位置的偏移。O2 优化时，编译器会进行循环展开，可这时每一个线程和参数都是独立的，能够更好的适配循环展开的优化。

简言之，多线程算法下，O1 优化不仅进行了串行同样的优化，还减小了并行部分线程需要频繁访问线程局部变量的开销；O2 优化进行循环展开，且并行部分是无数据依赖的，而串行的循环展开是有数据依赖的。所以不进行编译优化，O1 优化和 O2 优化的加速比随着优化程度的增大而增大。

4 pthread 与 openMP 的性能对比

4.1 性能测试

从表一中整合 pthread 与 openMP 的加速比信息，可以看到 openMP 的性能明显优于 pthread。

编译优化选项	pthread 加速比	openMP 加速比
不编译优化	1.068	1.090
O1 优化	1.406	1.725
O2 优化	1.473	1.943

表 3: pthread 和 openMP 加速比对比结果

在不进行编译优化、O1 优化和 O2 优化的情况下，openMP 的优化均比 pthread 的优化效果更

好。且在 O1 和 O2 优化下的优势十分明显。

4.2 profiling

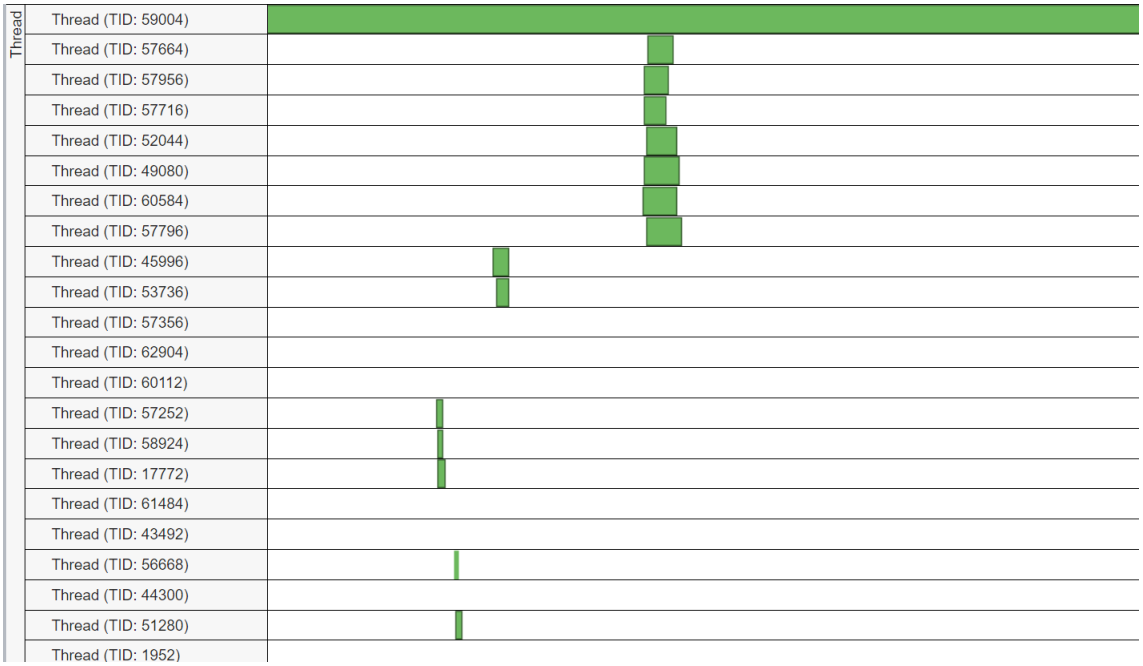

图 4.6: pthread 多线程 vTune 结果

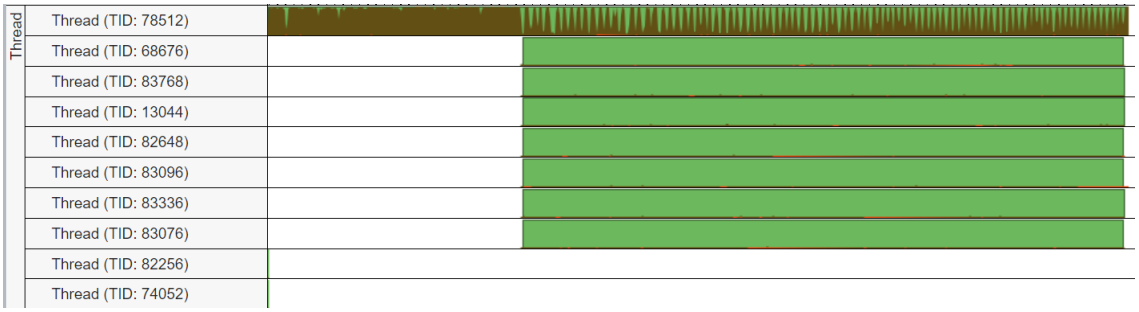

图 4.7: openMP 多线程 vTune 结果

Grouping: Thread / Function / Call Stack

Thread / Function / Call Stack	CPU Time ▼	Instructions Retired	Microar… CPI Rate	Module
Thread (TID: 78512)	38.867s	272,704,800,000	0.567	
Thread (TID: 68676)	0.474s	2,484,000,000	0.709	
Thread (TID: 83768)	0.460s	2,388,000,000	0.724	
Thread (TID: 13044)	0.446s	2,380,800,000	0.703	
Thread (TID: 82648)	0.418s	2,428,800,000	0.641	
Thread (TID: 83096)	0.418s	2,371,200,000	0.653	
Thread (TID: 83336)	0.415s	2,392,800,000	0.647	
Thread (TID: 83076)	0.413s	2,320,800,000	0.663	
Thread (TID: 82256)	0.002s	7,200,000	1.045	
func@0x18012e140	0.001s	0		ntdll.dll
func@0x14023dda7	0.001s	0		ntoskr…
func@0x1c00093c0	0s	2,400,000	0.000	ntfs.sys
func@0x1800a79ff	0s	2,400,000	0.000	ntdll.dll
func@0x1402b7270	0s	2,400,000	0.000	ntoskr…
Thread (TID: 74052)	0.001s	4,800,000	0.784	
func@0x1c00055e0	0.001s	0		fltmgr.sys
RtlAnsiCharToUnicodeChar	0s	2,400,000	0.000	ntdll.dll
KeReleaseSpinLock	0s	2,400,000	0.000	ntoskr…

图 4.8: openMP 各线程执行时间

其中 openMP 多线程的版本中，在执行 generate() 中的并行处理部分，主线程也在 8 条线程中，至于其他两条线程 (82256、74052) 是进程中存在少量系统自动创建的线程（如内存管理、信号处理线程），如 RtlAnsiCharToUnicodeChar 函数和 KeReleaseSpinLock 函数分别属于字符处理和内核同步领域，与 openMP 任务线程无关。

多线程版本	L1 命中率	L2 命中率	L3 命中率
pthread	97.93%	85.63%	47.46%
openMP	98.17%	83.14%	57.22%

表 4: pthread 和 openMP 缓存命中率

4.3 结果分析

OpenMP 是一种基于指令的高层并行编程模型，通过编译器指令（如 #pragma omp parallel）自动管理线程的创建、销毁和任务分配。

pthread 是底层的线程库，需要开发者显式管理线程生命周期、同步机制（如互斥锁、条件变量）和负载均衡。

本次实验中主要是以下方面体现出 openMP 的效果更优：

(1) 线程池复用

从 vTune 结果中可以看到以下信息：

OpenMP 默认使用线程池，在多次并行区域中复用线程，避免重复创建和销毁线程的开销。并且如图4.7所示，每一个线程的负载比较均衡。

Pthread 每次执行需显式调用 pthread_create，最终导致创建和销毁了大量的线程，，线程创建和销毁的系统调用开销较高。

(2) 循环展开

OpenMP：编译器能直接识别 #pragma omp for 的并行语义，对循环进行激进优化。

Pthread：线程函数（guess_thread）内的循环是通过手动分块的方式实现的，难以被编译器优化。

当然，初始时认为 openMP 的缓存优化会优秀很多，但是 profiling 中的数据显示，pthread 和 openMP 缓存命中率差距不大。

OpenMP 中 schedule(static) 采用块划分（Block Partitioning），将连续迭代块分配给线程，保证内存访问连续性，提高缓存命中率。Pthread 中手动划分的连续块（如 start 到 end）。二者的操作对缓存都很友好，此部分对性能影响不明显。

此外，其实在 pthread 的代码中的 PCFG.h 文件定义了全局的 pthread_t 类型的 8 个 threadId。但其实在我们的 pthread 写法中，使用这 8 个 threadId 的线程执行一次任务后就被销毁了，无法真正意义上的复用线程。只有通过线程池的方式才能真正复用线程。

5 Guess 和 Hash 同时加速

上一次实验进行了 Md5 Hash 的 SIMD 加速，接下来将 Guess 的加速与 Hash 的加速部分合并(未对 Md5 Hash 按照概率降序处理口令)，Md5 Hash 部分使用的是 4 条口令并发处理的 NEON 版本，Guess 部分使用的是 8 条口令并发处理的 pthread 版本。理论上来说这两个版本并不存在操作上的冲突，所以可以同时进行优化，但是 NEON 版本的多口令并发处理在不进行编译优化时是不能实现加速的，所以接下来检测这两个模块能否同时做到加速，哪种编译选项下可以实现同时加速。

5.1 代码实现

```
//main_pthread.cpp或main_openmp.cpp
//多个segment多线程版本代码

//MD5 Hash处理部分
for (int k = 0; k < num_thread; k++) {

    size_t total =  q.threadargs[k].local_guesses.size();
    size_t int_total=(total/4)*4;
    size_t left=total-int_total;
    // 每次处理 4 个口令
    for (size_t i = 0; i < int_total; i += 4)
    {
        batch_buffer.clear();
        // 收集 4 个口令
        for (int j = 0; j < 4; ++j)
        {
            batch_buffer.push_back( q.threadargs[k].local_guesses[i+j]);
        }
            MD5Hash_NEON(batch_buffer, batch_states);
    }

    bit32 state[4];
    for(int i=0;i<left;i++){
        MD5Hash(q.threadargs[k].local_guesses[int_total+i],state);
    }
}
```

5.2 性能测试

对 pthread 和 openMP 版本均进行了 Guess 时间和 Hash 时间的测试，测试结果也是对 10 次左右测试中的数据进行筛选后取的平均值。

pthread 多线程结果				
编译优化选项	串行 Hash 时间	并行 Hash 时间	Hash 时间加速比	Guess 时间加速比
不编译优化	9.454	12.091	0.782	1.068
O1 优化	3.077	1.745	1.763	1.406
O2 优化	2.917	1.660	1.757	1.473
openMP 多线程结果				
编译优化选项	串行 Hash 时间	并行 Hash 时间	Hash 时间加速比	Guess 时间加速比
不编译优化	9.454	12.028	0.786	1.090
O1 优化	3.077	1.757	1.751	1.725
O2 优化	2.917	1.689	1.727	1.943

表 5: Guess 和 Hash 优化合并加速比结果

从表 4 中可以看到，pthread 和 openMP 多线程版本的结果相似，在无编译优化时，Hash 时间不能得到优化，在 O1 和 O2 优化下，Hash 时间和 Guess 时间均能得到不同程度的优化。

5.3 结果分析

将 Hash 和 Guess 的加速部分合并后，Hash 时间得到的优化和 Guess 时间得到的优化与未合并时的结果相同，也符合理论上的这两个过程的独立性。只是在实现时，我们并未按照概率降序进行 Md5 Hash 处理，如果加上这一步骤，则 Guess 时间和 Hash 时间的优化均会有所降低。

6 总线程数对加速比的影响

6.1 性能测试

这里为了保证结果的准确性且减少测试的次数，将总猜测数调整为 1 亿。

pthread 多线程结果			
总线程数	不编译优化	O1 优化	O2 优化
2	1.030	1.238	1.300
4	1.054	1.297	1.326
8	1.057	1.457	1.517
openMP 多线程结果			
总线程数	不编译优化	O1 优化	O2 优化
2	1.050	1.388	1.402
4	1.089	1.630	1.809
8	1.106	1.858	1.872

表 6: 不同线程数加速比结果

pthread 多线程和 openMP 多线程下，无论是不编译优化，O1 优化还是 O2 优化，总线程数越大，加速比越大。

6.2 结果分析

在阈值取值合理的情况下，理论上线程数越高，加速比越大。这是由于同时进行的任务越多，并行度越高。和我们的测试结果也是一致的。

但是在这个问题上，不一定会出现线程数越高，加速比越大的现象。虽然阈值不随样本数据的改变而改变，但可能会出现样本全部在 8 线程阈值内等情况。此时 4 线程必然是比 8 线程快的。所以对于不同分布的数据应该采取不同的线程数的优化。

7 总猜测数对性能的影响

7.1 性能测试

对于总猜测数为 10 亿的情况下，执行时间会很长。尤其对于串行算法来说，10 亿条口令的结果无法在 10 分钟内完成，而由于服务器无法测试超过 10 分钟的代码。所以这里就不进行 10 亿条口令的加速比的计算。但是会记录并行算法在 10 亿条口令时的执行时间。

pthread 多线程加速比结果				
总线程数	总猜测数	不编译优化	O1 优化	O2 优化
2	1000 万	1.054	1.158	1.289
	1 亿	1.030	1.238	1.300
4	1000 万	1.041	1.384	1.455
	1 亿	1.054	1.297	1.326
8	1000 万	1.068	1.406	1.473
	1 亿	1.057	1.457	1.517
openMP 多线程加速比结果				
总线程数	总猜测数	不编译优化	O1 优化	O2 优化
2	1000 万	1.038	1.502	1.539
	1 亿	1.050	1.388	1.402
4	1000 万	1.040	1.625	1.630
	1 亿	1.089	1.630	1.809
8	1000 万	1.090	1.725	1.943
	1 亿	1.106	1.858	1.872

表 7: 不同总猜测数加速比结果

从数据上看，调整了总猜测数后，O2 优化下，pthread 版本中，8 线程数的加速比在 1000 万和 1 亿的情况下均是最大的，不过 2，8 线程数版本在总猜测数 1 亿时的加速比均比 1000 万时的加速比大，而 4 线程版本的加速比在总猜测数 1 亿时比 1000 万时更小；openMP 版本中，8 线程数的加速比在 1000 万和 1 亿的情况下均是最大的，不过 2，8 线程数版本在总猜测数 1 亿时的加速比均比 1000 万时的加速比小，但是 4 线程版本的加速比在总猜测数 1 亿时比 1000 万时更大。

pthread 多线程 Guess 时间结果			
总线程数	不编译优化	O1 优化	O2 优化
2	/	133.828	123.348
4	/	133.035	120.366
8	/	133.846	123.357
openMP 多线程 Guess 时间 (s) 结果			
总线程数	不编译优化	O1 优化	O2 优化
2	/	143.255	121.996
4	/	122.205	120.384
8	/	120.025	112.441

表 8: 总猜测数 10 亿时 Guess 时间结果

当猜测总数达到了 10 亿，不进行编译优化是无法在服务器上得到结果的，表中便用“/”代替。这里只关注 Guess 的时间，我们可以看到 pthread 版本中，各线程版本的时间是差不多的，openMP 版本中线程数愈大，Guess 时间越短。

7.2 结果分析

绘制 1 亿和 10 亿时的样本分布情况图。

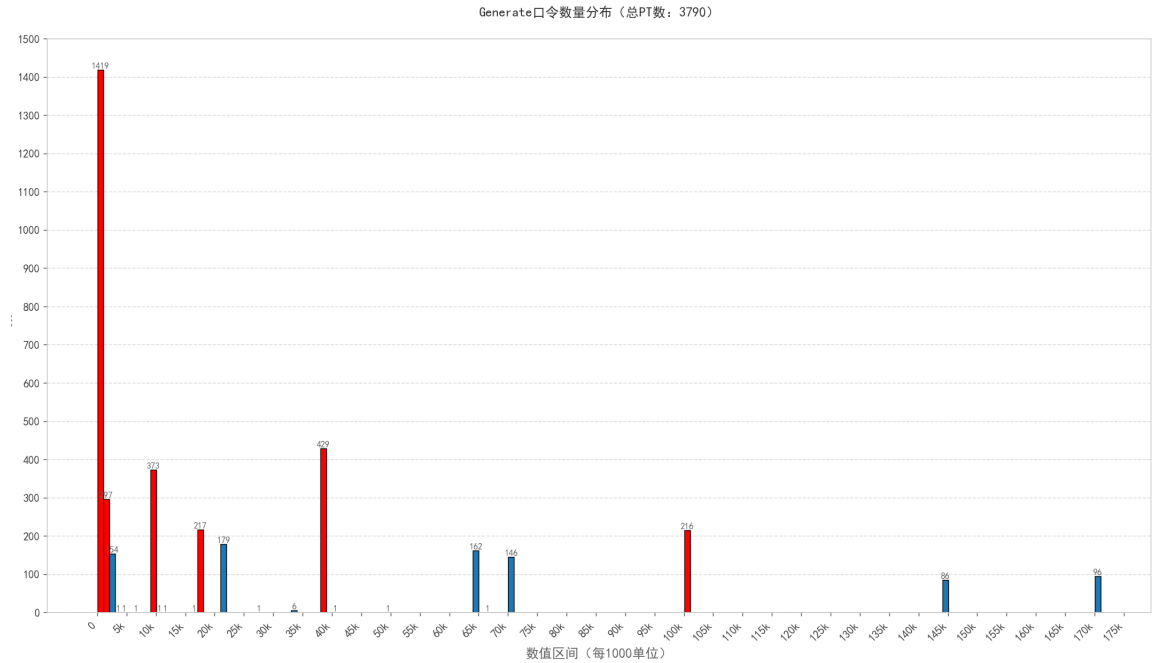

图 7.9: 总猜测数 1 亿时的 PT 分布

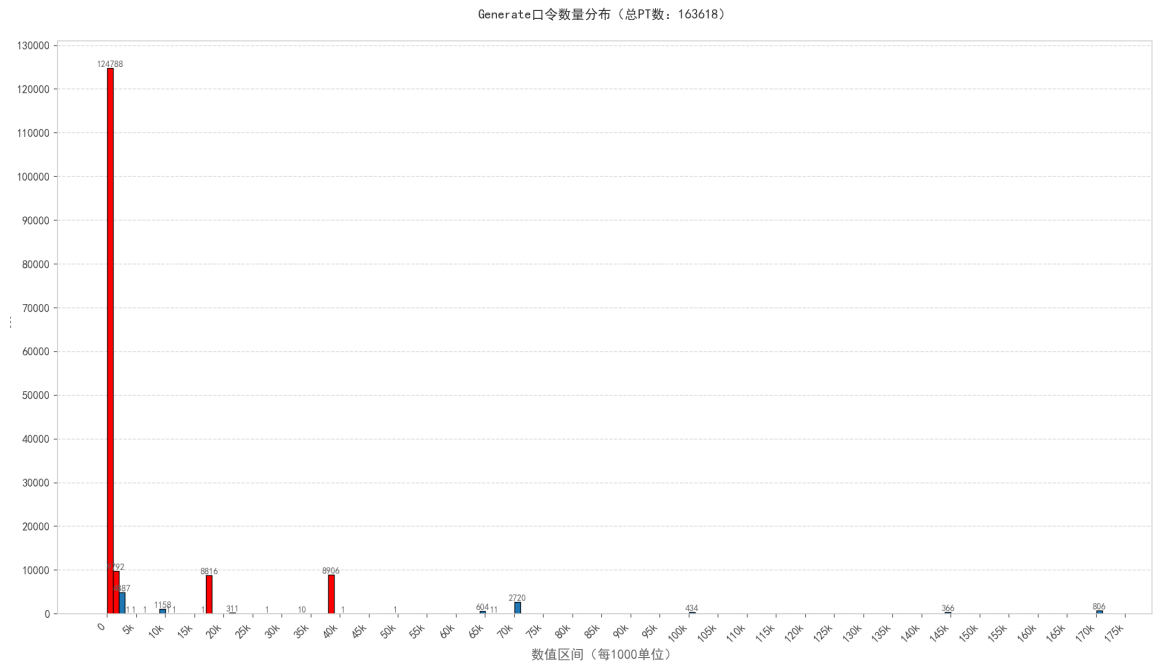

图 7.10: 总猜测数 10 亿的 PT 分布

可以看到随着总猜测数量的增加，[0,1000) 的 PT 数占比一直在增大，到了总猜测数 10 亿时，占比已经高达 76.3%。但是 Guess 时间占比仍然不超过 10%。所以只要阈值设定比 1000 大，实现方式不太糟糕，多线程版本几乎就是可以实现加速的。但是想要达到最优的性能是需要更精细的筛选的。

7.2.1 阈值调整

当总猜测数为 10 亿时，pthread 实现的多线程的阈值取值就不是很合理，2，4，8 线程下，Guess 时间差距不大，甚至 8 线程版本最慢。

总猜测数 1 亿时，[10000,11000)PT 数大约时 [20000,21000) 的 PT 数的两倍，可是当总猜测数为 10 亿时，[20000,21000)PT 数几乎达到了 [10000,11000)PT 数的 8 倍，而且 [20000,21000) 区间的 PT 占比也达到 6%。我认为 [20000,21000) 是 8 线程的 pthread 版本在总猜测数 10 亿时，性能最差的原因。所以推测 8 线程的 pthread 版本阈值会超过 20000。在将 8 线程的阈值调整到 22000 时，测试结果如下：

总线程数	不编译优化 (s)	O1 优化 (s)	O2 优化 (s)
8	/	132.025	116.744

表 9: 阈值调整后 8 线程 pthread 版本 Guess 时间结果

能够看到相对于阈值为 10500 来说 22000 的阈值更适合 8 线程的 pthread 版本。以 O2 的结果来看，性能优化了 6.613s，占原来 Guess 时间的 5.36%，上面已经得到 [20000,21000) 区间的 PT 占比约为 6%，也能反映是阈值选取不当的原因。

而对于 openMP 版本我们设定的阈值还是很有效果的。无论总猜测数是 1000 万、1 亿还是 10 亿，编译优化选项如何，openMP 的加速比均随着线程数的增加而增加。

8 总结

本次实验围绕概率上下文无关文法（PCFG）的并行化口令猜测展开，通过 Pthread 和 OpenMP 多线程框架的深度优化，系统性探索了并行策略在 PCFG 中的应用。

8.1 多线程适用场景

并行算法的效率高是源于同步进行任务执行，可是劣势是带来线程开销和互斥锁等资源的消耗。在特定问题中，不能直接使用全局的多线程算法，需要找到多线程算法并行优化和资源消耗之间的平衡点。在本问题中便是阈值的选取。

8.2 加速实质

在选取阈值后能够大幅度提升性能的根本原因是对产生的猜测数很少，但是数量很多的 PT 不消耗过多的资源。从样本看，这样的 PT 极多，全局使用多线程，就是在消耗不必要的资源。

8.3 讨论阈值

理论上，忽略不同字符串 emplace_back() 处理时间不同等问题，本实验中的阈值是与样本选取无关的，但是确定准确的阈值又必须依赖于样本数据。所以如果想找到阈值，需要预先准备各猜测口令数的 PT 数均匀分布的样本。

8.4 总猜测数对性能影响的重新解读

本意上是对总猜测数进行讨论，实际上从最后结果的分析来看，是在对样本数据以及选取的阈值进行讨论。所以这部分的结果与总猜测数没有太大相关性。这也是没有在 1000 万到 1 亿之间继续细分进行测试的原因。不过总猜测数也确实间接影响了分布，能帮助我们继续缩小阈值的范围。

8.5 优化方向

(1)pthread 版本可以采用线程池优化。在 pthread 的 vTune 性能测试中看到，虽然使用全局 threadId，但是在创建线程时仍然是无法实现线程复用的。而 openMP 线程池也确实带来了更高效的性能，于是 pthread 版本也可以利用线程池来进行优化。

(2) 目前的阈值机制的场景适应性不足。这是由于无法精准确定阈值导致的。当前阈值为静态设定（如 Pthread 8 线程固定为 10500），依赖特定样本分布（如猜测上限 1 千万时 [0,2000) 区间占比 63%），但实际应用中口令分布动态变化，可能导致阈值失效。例如，当猜测上限提升至 10 亿时，[20000,21000) 区间 PT 数占比突增至 6%，原阈值导致 Pthread 性能下降。引入动态阈值调整算法，基于实时任务规模统计（如滑动窗口）自动切换串行/并行策略。

9 代码仓库

2 线程、4 线程和 8 线程的 pthread 版本代码以及 openmp 版本代码已经上传至 Gitee

参考文献

[1] Ziyi Huang, Ding Wang, and Yunkai Zou. Prob-hashcat: Accelerating proba-bilistic password guessing with hashcat by hundreds of times. In Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses, 2024.