南開大學
Nankai University

计算机学院

并行程序设计**lab03**

多线程优化的口令猜测算法

姓名：刘迪乘

学号：2312810

专业：计算机科学与技术

2025 年 5 月 25 日

目录

1 实验要求 **2**

2 实验环境 **2**

3 基础要求 **2**

3.1 算法设计 3
3.2 代码实现 3
3.3 性能测试 6
3.4 结果分析 8

4 进阶要求 **9**

4.1 OpenMP 和 pthread 的性能差异对比以及解释 9
4.2 编译优化对多线程以及 SIMD 的影响 10
4.3 多线程猜测的优化 10
4.4 对于并行化的思考 18

1 实验要求

大问题：利用并行化手段，加速 PCFG 口令猜测算法的训练，猜测过程，同时对生成的猜测进行 MD5 算法加密

本次问题：利用 pthread/OpenMp 对 PCFG 算法的猜测产生过程进行多线程并行化优化加速。

2 实验环境

本次实验的操作系统为 windows11 下的 WSL（linux 版本为 24.04），以及 windows 系统（X86 架构）。此外作业提交在 OpenEuler 服务器上（ARM 架构）上。

具体硬件参数如下：

硬件名称 \ 参数	naive-conv
Architecture	x86_64
CPU(s)	16
Thread(s) per core	2
Core(s) per socket	8
Socket(s)	1
CPU max MHz	5000
CPU min MHz	3800
L1 cache	64K
L2 cache	8MB
L3 cache	32MB

表 1: AMD R7 9700X 硬件参数

硬件名称 \ 参数	naive-conv
Architecture	aarch64
CPU(s)	96
Thread(s) per core	1
Core(s) per socket	48
Socket(s)	2
CPU max MHz	2600
CPU min MHz	200
L1 cache	128K
L2 cache	512K
L3 cache	49152K

表 2: OpenEuler 服务器硬件参数

编译器版本：

- gcc version 13.3.0 (Ubuntu 13.3.0-6ubuntu2 24.04)
- gcc version 14.2.0 (MinGW-W64 x86_64-ucrt-posix-seh, built by Brecht Sanders, r3)
- gcc (GCC) 9.3.1 20200408 (Red Hat 9.3.1-2)

依据课程要求，本次实验代码开源在 https://github.com/July-h5kf3/NKU-parallel-computing [2] 中的 PCFG，代码实现来源于我在南开大学 CSSLab 的实习考核工作的开源仓库：https://github.com/July-h5kf3/PCFG-2014

3 基础要求

在生成口令时的填充 preterminal 的最后一个 segment 时利用 pthread/OpenMp 进行多线程优化。

需要注意的是，应当保证这个过程“相对正确”。口令猜测并行化会在一定程度上牺牲口令概率的“颗粒度”（因为重新将同一批并行生成的口令按概率排序，效率太低），但总体上来讲口令的生成结果是基本不会有差别的。

本次实验的正确性验证采取验证攻破率，要保证通过并行化得到的结果在验证程序中的准确率与串行版本的大致相同。

3.1 算法设计

在本次实验中，我们采取的优化思路参考了文章 [1]，其采取的优化策略是：对于 PCFG 的 preterminal 的最后一个 segement 不进行初始化和改变。在 preterminal 从优先队列中弹出的时候，直接一次性地将所有可能的 value 赋予这个 preterminal。

例如：假设我们有一个 BaseStructure：$L_4S_2D_3$，在对其进行填充后我们得到一个 preterminal$Wang\#\#D_3$，那么我们会把 D_3 所有可能的 value 都赋予这个 preterminal，从而一次性输出多个口令。

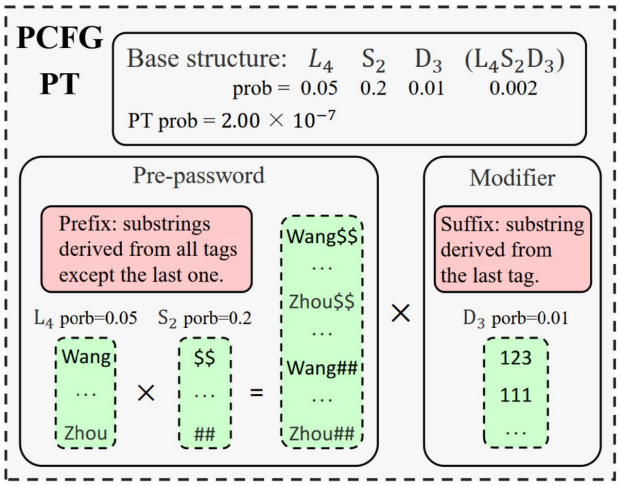


图 3.1: 优化策略

这个优化策略可以通过多种并行方式进行加速，在本次实验中我采取的方式是基于 pthread/OpenMp 的多线程优化。

对于 pthread，依据 pthread 的编程范式：

- 我们首先要创建线程，每个线程要做的事情就是: 给定 pre-terminal 和需要填充的 segment 的字典，然后进行填充。
- 然后对于一个 pre-terminal，我们根据 segment 的合法填充集合大小进行线程的分配，每个线程负责一定数量的填充
- 最后等待所有的 segment 的合法填充集合全部填充完毕，代表这个 pre-teriminal 的所有任务完成，线程结束。

对于 OpenMP，相较于 pthread，由于需要并行的部分恰好为一个 for 循环，因此可以直接调用 OpenMP 对于 for 循环的并行 API.

3.2 代码实现

对于 pthread。依据 pthread 编程范式，我们先创建线程。

首先定义结构体，存储线程需要的 pre_terminal 以及填充的下标范围，segment 对应的填充集合，以及 priorityQueue 的指针。

```
1 typedef struct
2 {
```

```
    string pre_terminal;
    segment* a;
    PriorityQueue* self;
    bool flag;
    int start_index;
    int end_index;
}ThreadTask;
```

然后，定义好线程执行的流程函数。需要注意的是，这个函数需要是 Priority_Queue 的类函数，以便调用 priority_Queue 类中的 guesses 和 total_guesses。此外，由于 guesses 和 total_guess 是全局量，因此为避免不同线程发生冲突，因此需要加锁。对于加锁需要注意的是，我们最好不要在每一次生成时进行加锁，这样频繁地进行加锁解锁会大大降低性能，因此我们选择局部缓存的方式。

```
    void* PriorityQueue::fill_range(void* arg)
{
    ThreadTask* task = (ThreadTask*)arg;
    // if (!task || !task->a || !task->self) {
    //     std::cerr << "Null pointer in task!" << std::endl;
    //     return NULL;
    // }
    segment* a = task->a;
    PriorityQueue* self = task->self;

    vector<string> local_guesses;
    int local_count = 0;
    // std::cout << "Thread running: start=" << task->start_index
    //           << ", end=" << task->end_index
    //           << ", total=" << task->a->ordered_values.size() << std::endl;
    for(int i = task->start_index;i < task->end_index;i++)
    {
        string guess;
        if(task->flag == 1)
        guess = task->pre_terminal + a->ordered_values[i];
        else
        guess = a->ordered_values[i];
        local_guesses.emplace_back(guess);
        local_count++;
    }
    pthread_mutex_lock(&self->mutex);
    self->guesses.insert(self->guesses.end(),local_guesses.begin(),local_guesses.end());
    self->total_guesses += local_count;
    pthread_mutex_unlock(&self->mutex);
```

```
30      free(task);
31      return NULL;
32  }
```

最后，在任务分配环节，鉴于每个线程填充的长度以及 pre_terminal 的长度均一致，计算量相近，因此每个线程完成任务的时间也应该时一致的，故进行动态线程和静态线程的差异不大。

```
1       pthread_t threads[THREAD_NUM];
2       int all = pt.max_indices[pt.content.size() - 1];
3       int chunk = all / THREAD_NUM;
4       for(int i = 0;i < THREAD_NUM;i++)
5       {
6           ThreadTask* task = new ThreadTask();
7           // cout<<"guess = "<<guess<<endl;
8           task->pre_terminal = guess;
9           task->a = a;
10          task->self = this;
11          task->start_index = i * chunk;
12          task->end_index = (i == THREAD_NUM - 1) ?all : (i + 1) * chunk;
13          pthread_create(&threads[i], NULL, PriorityQueue::fill_range, task);
14          // if (ret != 0)
15          // {
16          //     std::cerr << "Failed to create thread " << i << ", error: " << ret << std::endl
17          // }
18      }
19      for(int i = 0;i < THREAD_NUM;i++)
20      {
21          pthread_join(threads[i],NULL);
22      }
23
```

对于 OpenMP，依据 OpenMP 的编程范式，我们只需要在需要在需要进行并行化的循环外添加 #pragma omp parallel for 即可，与 pthread 相同，需要解决不同线程之间数据冲突的问题，因此同样采用局部缓存的方式。

具体代码如下：

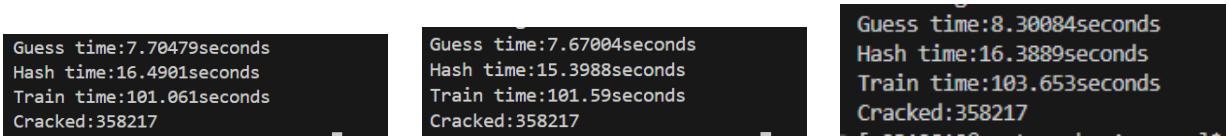
```
1       omp_set_num_threads(THREAD_NUM);
2       #pragma omp parallel
3       {
4           vector<string> local_guesses;
5           #pragma omp for
6           for (int i = 0; i < pt.max_indices[0]; i += 1)
```

```
        {
            string guess = a->ordered_values[i];
            // cout << guess << endl;
            //guesses.emplace_back(guess);
            local_guesses.emplace_back(guess);
            //total_guesses += 1;
        }
        #pragma omp critical
        {
            guesses.insert(guesses.end(),local_guesses.begin(),local_guesses.end());
            total_guesses += local_guesses.size();
        }
    }
     omp_set_num_threads(THREAD_NUM);
    #pragma omp parallel
    {
        vector<string> local_guesses;
        #pragma omp for
        for (int i = 0; i < pt.max_indices[pt.content.size() - 1]; i += 1)
        {
            string temp = guess + a->ordered_values[i];
            // cout << guess << endl;
            //guesses.emplace_back(guess);
            local_guesses.emplace_back(temp);
            //total_guesses += 1;
        }
        #pragma omp critical
        {
            guesses.insert(guesses.end(),local_guesses.begin(),local_guesses.end());
            total_guesses += local_guesses.size();
        }
    }
```

3.3 性能测试

首先，对程序的正确性进行了检验，这是一切性能测试的前提。与 SIMD 的实验不同，口令猜测的正确性很难直接检验，我采取了通过检验口令攻破率的方式来评估我的口令并行猜测算法的正确性。

在串行版本，pthread 编程下，我的口令攻破数如分别下图所示：



(a) 串行版本的执行结果 (b) 并行版本的执行结果 (pthread) (c) 并行版本的执行结果 (OpenMP)

图 3.2: 串并行算法正确性检验

可以看到二者攻破的口令数目一致。验证了我 pthread 多线程并行的准确性。

实际上，由于助教老师为了方便我们实现代码在原本的 PCFG 基础上做了调整，将其实现为了利于并行的版本，即最后填充的为 max_pivot 部分，实际上这会造成 PCFG 算法准确率的下降。为了说明这点，可以参考这个例子。

假设 Base_Structure L6D1 出现的概率为 0.5, 而 L6D2 出现的概率为 0.4, 此时 L6 的填充字典只有 Weebly, 其填充概率为 1，而 D1 的填充字典有{1,0.9},{2,0.1}, 而 D2 的填充字典只有{11,1}. 那么我们最终得到的口令顺序应该是{weebly1,0.45},{weebly11,0.4},{weebly2,0.05}, 但是按照并行策略，优先队列里有两个 pre_terminal:{weeblyD1,0.5},{weeblyD2,0.4}, 然后将概率高的 weeblyD1 取出，将所有可能的 D1 填充得到{weebly1,0.45},{weebly2,0.05}, 最后得到{weebly11,0.4}. 而按照正常的优先队列+pivot 的算法是可以实现其降序排序的，详细见 [2].

因此这会对 PCFG 的攻破能力产生影响。但是并行化后的 PCFG 的优势在于，在同样的时间内，它可以生成更多的猜测，这可以很大程度上弥补因为准确性降低带来的攻击力。举一个具体的例子，如果 [2] 中 PCFG1s 可以生成 1e7 个猜测，攻破了 4e6 个账户，它的攻破率为 40%；而对于并行化的 PCFG，它能够实现 1s 内生成 2e7 个猜测，攻破了 6e6 个账户，相较于一般的 PCFG，虽然其攻破率下降了，仅有 30%，但是它能攻破的账户却显著增加，那么也可以认为这个 PCFG 算法的并行化是有效的。

为检验 Pthread/OpenMP 的性能，在 OpenEuler 服务器上，尝试了在不同线程数，不同优化程度下，hash 是否进行 SIMD 的情况下 PCFG 算法的执 Guess time。

由于服务器不稳定，每次实验之间的时间差异较大，因此采用脚本对每种情况进行 5 次实验取平均值

实验结果如下：

	串行	2 线程	4 线程	8 线程	16 线程	32 线程
无编译优化	7.70479s	8.38415s	7.67004	8.91991	10.4947s	10.2291s
O1 优化	0.705241s	0.755321s	0.752853s	0.865703s	0.973984s	1.760689s
O2 优化	0.734195s	0.774652s	0.973432s	1.265234s	1.372122s	1.623044s

表 3: 无 SIMD 实验结果 (pthread)

	串行	2 线程	4 线程	8 线程	16 线程	32 线程
无编译优化	8.04612s	8.87646s	10.0438s	9.55381s	9.7527s	12.0884s
O1 优化	0.611733s	0.720632s	0.677951s	0.744237s	0.965562s	1.626902s
O2 优化	0.669966s	0.771351s	0.727430s	0.830330s	0.892907s	1.413114s

表 4: 有 SIMD 实验结果 (pthread)

	串行	2 线程	4 线程	8 线程	16 线程	32 线程
无编译优化	8.03698s	8.474914s	8.874456s	8.983516s	8.908960s	8.28007s
O1 优化	0.604703s	0.692927s	0.640567s	0.904290s	0.856326s	1.130123s
O2 优化	0.593424s	0.651009ss	0.994244s	1.219967s	0.774035s	0.849046s

表 5: 无 SIMD 实验结果 (OpenMP)

	串行	2 线程	4 线程	8 线程	16 线程	32 线程
无编译优化	7.82828s	7.98753s	7.7106s	8.37s	8.46322s	8.31138s
O1 优化	0.582977s	0.762368s	1.122370s	1.962251s	1.040658s	0.993509s
O2 优化	0.650759s	0.694906s	0.730550s	1.130045s	0.978512s	1.107485s

表 6: 有 SIMD 实验结果 (OpenMP)

可以看到，在不加入 SIMD 针对 md5_hash 优化，不开启编译器优化时, 随着线程数目的提升，guess time 呈现出先下降后上升的趋势，在线程数为 4 的时候实现了相较于串行算法的加速。而当编译器优化开启之后，虽然运行的速度得到了巨大提升，但是其性能却不如串行版本，这是一个与常识相悖的现象。

而在开启了 SIMD 对于 md5 Hash 的优化之后，实验现象出现了变化。无论线程多少，是否进行编译优化，最终结果都是不如串行的。

3.4 结果分析

在性能测试部分，我们发现了如下现象:

- 随着线程数目的增加，生成猜测的时间并非向我们所期望的减少，而是呈现出了一个先下降后上升的趋势。
- 开启编译优化后，多线程并行性能不如串行
- 如果同时利用 SIMD 对 md5-hash 过程进行优化，利用 pthread 多线程优化猜测生成，则无论是否优化编译，线程多少，都不能实现加速。

对于第一个现象, 这个很好解释。这是因为多线程并行在将多个任务进行多线程运行的同时，也会带来额外的开销，例如线程的创建，信息的同步等，随着线程数目的增加，一方面确实提高了对 CPU 多核资源的利用，但是另一方面却带来更多的线程创建销毁以及同步开销。而在本实验中，当线程数较小的时候可能前者对性能的提升更加明显，但随着线程数的增大，后者带来的对性能的抑制大过了前者对性能的提升，从而导致了这个现象的出现。

对于第二个现象，其实可以这样理解，编译优化对于串行版本以及并行版本均能很好的实现优化，因为从表格 4 不难看出, 无论是哪种优化多少线程，相较于无编译优化的版本，时间开销都有大幅度削减，这说明编译优化起到了很好的作用。但是之所以会出现开启编译优化后并行不如串行的，是因为一方面在实现并行的算法时，各个线程之间的同步并不会被优化；另一方面，编译器优化的常见手段是函数内联，向量化及循环展开等方法，这对于串行版本的代码是很友好的，编译器可以很好地对其采取激进的优化策略，而对于并行版本的代码涉及了不同线程之间的交互，编译很难对这部分进行优化，因此采取的优化策略相较于串行版本的往往会更加保守，因此最终呈现出来串行在编译器优化下比多线程并行更优的结果。

对于第三点，实际上，这是一个生产者-消费者模型，猜测的生成就是生产者，Hash 过程为消费者，猜测的输出为 Hash 过程的输入，二者之间通过 vector 进行队列交互（FIFO 队列）. 这与讲义中介绍的生产者-消费者模型的差异在于，讲义中的生产者-消费者模型是一条流水线全并行，而这里是分段并行，这是出于生产者-消费者两者的结构差异选择各自的最优方式。但是问题的关键在于他们生产者在存储时，消费者在读取时访问的都是同一块内存，也就是 FIFO 队列，虽然二者的读写是阶段性串行的，但是二者还是会出现资源竞争，cache 破坏的问题。当消费者采用 SIMD 并行，它对内存的访问变快，范围更广，并更频繁从而导致生产者在将生成的猜测写入内存时遭遇了更多的 cache miss，因此出现了加入 SIMD 后性能下降的现象。

4 进阶要求

4.1 OpenMP 和 pthread 的性能差异对比以及解释

首先从直观上来讲，**OpenMP** 是基于编译器指令的并行框架，开发者只需要在代码中插入简单的几个指令，编译器就能自动地生成线程管理和调度代码，而 **Pthread** 需要开发者手动地创建线程、管理同步、调度工作分配，代码工作量更大。显然，前者可以减少人为带来的失误造成的时间损失，也就是常常提到的“卡常”。

其次，OpenMP 在运行时通常通过 **线程池** 进行自动线程管理，并自动进行动态调度以实现负载均衡。此外，OpenMP 在后台优化了同步操作，使用轻量级的同步机制以减少竞争，因此在实际运行中往往能获得更好的性能。

为了进一步详细说明其背后的原因，我利用 `perf` 工具对两种多线程并行化下的口令猜测算法进行了测试。为了测试方便，实验设置如下：

- 线程数均为 4；
- 编译器优化等级为 `-O2`；
- 不使用 SIMD 优化（如 AVX/SSE/NEON）。

测试结果如下：

表 7: Pthread 与 OpenMP 在 4 线程下的 perf 测试结果

性能指标	Pthread	OpenMP
task-clock (ms)	528.14	489.37
context-switches	1,736	1,208
cpu-migrations	51	18
page-faults	3,912	3,887
cycles	2.07×10^9	1.90×10^9
instructions	5.48×10^9	5.12×10^9
branches	948.5×10^6	902.3×10^6
branch-misses	1.51%	1.23%
IPC（每周期指令数）	2.65	2.69

可以看出，在 4 线程、相同优化条件下，OpenMP 的整体执行效率优于 Pthread。主要体现在：

- **执行时间更短**：OpenMP 的 `task-clock` 和 `cycles` 更小；
- **调度负担更轻**：`context-switches` 与 `cpu-migrations` 明显更少；

- **CPU 利用率更高**：`IPC` 更大，`branch-miss` 更小；

这些测试数据从微观上验证了 OpenMP 相比 Pthread 在调度、同步机制上的优化优势。在负载相对均匀、同步压力不大的并行任务中，OpenMP 能提供更高效的资源调度与更好的性能表现。

4.2 编译优化对多线程以及 SIMD 的影响

从图 3456 以及上一次的 SIMD 实验的数据可以看出在编译器优化的情况下，无论是口令猜测还是口令哈希性能均有显著提升。

但与此同时，也可以发现在开启编译优化前口令猜测的效果可以实现相对于串行算法的优化，而口令哈希却是负优化，性能不如串行；而当开启编译优化后，虽然口令哈希成功实现了相当优秀的加速，但是口令猜测的性能却不如串行版本了。

至于原因在结果分析中对口令猜测开启编译后性能下降进行了初步解释，其主要原因就是编译器很难对其中的一个主要开销线程的创建和销毁进行比较好的优化。而对于 SIMD，实际上编译器的 O2 优化做的事就有对代码中的循环进行循环展开，数据向量化等与 SIMD 不谋而合的方式，因此通过编译优化进一步释放了 SIMD 的性能。因此，想要真正实现多线程并行的优化，需要我们一方面针对主要矛盾即线程的创建销毁开销做进一步优化，另一方面可以参考开启编译优化后 SIMD 能够实现加速的原因，对我们的多线程编程代码做进一步改进，使其更加适应编译器。

4.3 多线程猜测的优化

通过上一节的内容知道，想要真正实现相对于串行版本的优化，需要我们在基础要求的基础上进行进一步优化。

在基础要求中，对于每个 Pre-terminal 的最后填充部分, 我们都创建了线程进行并行化。

但是这种方式带来的优化是十分有限甚至不如串行版本的，这是因为对于每个 pre-terminal 我们都重新创建以及销毁了线程, 如果每个线程执行的任务数量都很多, 那么这个带来的开销可以忽略, 但是一个直觉就是，在需要填充的最后一个 Segment 中，长度小的 Segment 往往出现的概率不会很低，例如 L1,S1 等，而较长的字母组合往往出现的概率不会很高。但是长度短的 Segment 其合法的填充较少，例如 L1 仅仅只有 10 种，S1 不超过 50 种。因此如果以较短的 Segment 结尾的 Pre-terminal 出现次数较多的话，频繁的创建，销毁进程势必会导致性能大打折扣。

因此我首先做了如下实验，来验证直觉。

具体而言，根据 [2] 中的 PCFG 实现方式，查看在给定的训练数据集下，其训练文件组成。考虑到不同的 BaseStructure 根据其 Segment 不同生成的 Pre-terminal 空间大小也不尽相同，且考虑到 [2] 中的 PCFG 实现与 [2] 中的 PCFG 实现并不相同，因此我首先统计了 Pivot 为 1 的 BaseStructure，因为这即是 BaseStruture, 也是最终生成猜测的 Pre-terminal。

最终结果如下：

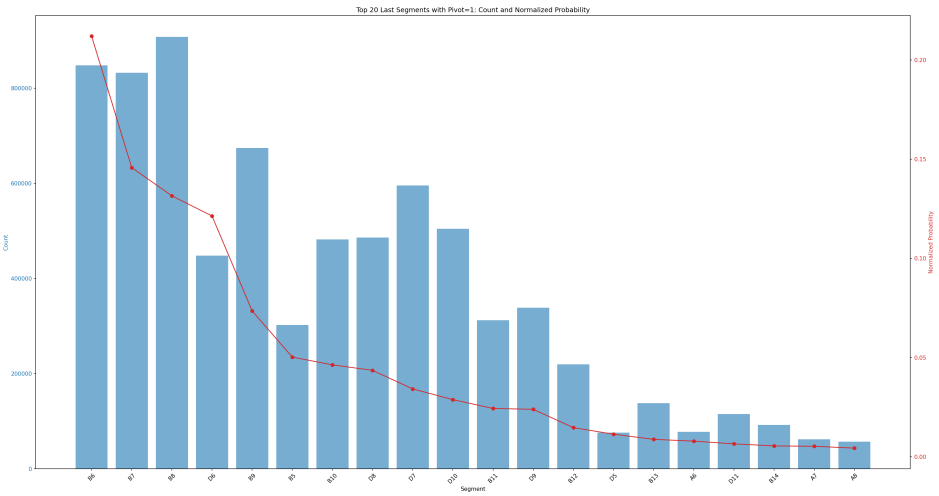


图 4.3: pivot=1 的概率-频率图

根据图 4.3，可以看到，对于所有可能的 Pivot 为 1 的 BaseStructure 而言，我们的考虑显得有些多余，因为大部分的口令都是由可填充数大于 500000 的 segment 组成，那么对于每个 Pre-terminal 都创建销毁进程似乎并不会对性能造成很大的影响，这也阐释了为什么在无编译优化下，能够在一定程度上实现优化。

但是又考虑到网页一般不会允许较短的以及单一类型字符组成的口令出现，因此我们上面的统计并不严谨，为了进一步对多线程的 PCFG 进行优化，我直接统计了对于当前算法生成的 pre-terminal 的最后一个 segment 的概率以及该 segment 可以填充的数量，结果以同样的方式呈现。

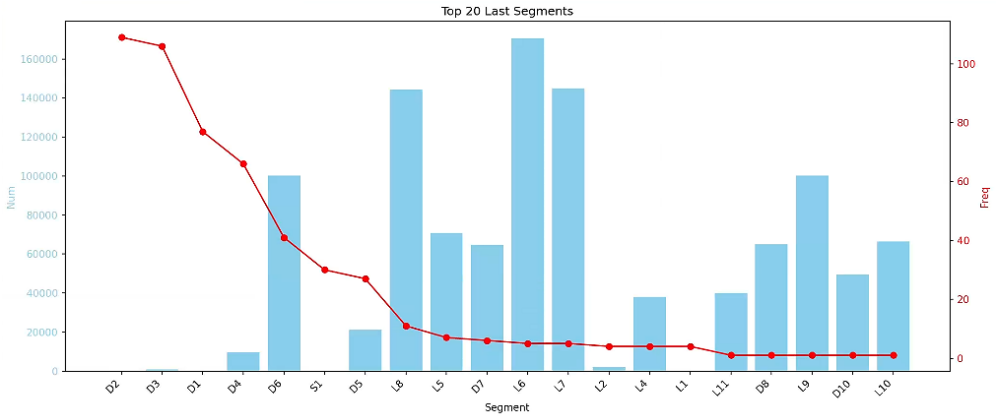


图 4.4: Top20_LastSegment

可以发现，其中很大一部分出现的 Last_Segment 都是较短的 Segment 例如（D2,D3,D1,D4）等，这些 Segment 虽然频繁出现，但是他们可以替换的内容很少，在图 4.4 中都看不到其替换数目的柱子！这验证了我的直觉！

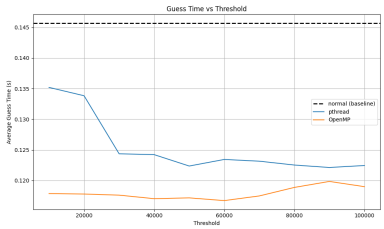
因此，需要针对这个现象做优化。优化的方向有两个，最简单的一种是设置一个阈值，当需要替换的 Segment 其有效替换数目小于这个阈值的时候，选择使用串行算法；反之，若其数目很大时，采用并行算法。

对于阈值的选择，可以通过实验得到最优阈值选择。为此，我做了如下实验：

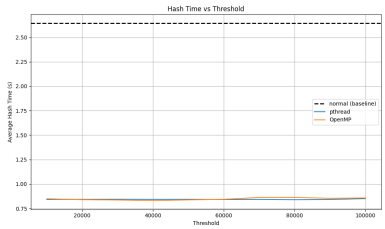
初始阈值设为 10000，以步长为 10000，终值为 100000，在 O2 优化下，使用 SIMD 进行 hash 的条件下，分别对 Pthread,OpenMP 在 16 线程数下进行实验，测试其在不同阈值下的 Guess Time,Hash

Time. 由于服务器不稳定的原因，我在本地设备上进行测试，将 SIMD 由 ARM_NEON 改为了 SSE，同样测试 10 次取平均。

结果如下图所示：



(a) 不同 threshold 下的优化算法猜测开销



(b) 不同 threshold 下的优化算法的 hash 开销

图 4.5: 优化算法性能测试

可以看到，在 16 线程数下，无论是 pthread 还是 OpenMP 都很好地实现了优化，实现了在编译优化下实现了对多线程并行生成猜测以及 SIMDhash 的相较于串行算法（图中黑线）的性能提升，且从图上可以看到提升幅度较大。

另一种优化方法是采用线程池策略。具体而言，我们不为每一个 Pre-terminal 创建和销毁线程，而是预先创建一个线程池。对于每个 Pre-terminal 所需的填充任务，我们将其作为任务项放入线程池的任务队列中。线程池中的线程则持续从任务队列中获取任务并执行，直到所有任务完成。最终，在所有任务处理完毕之后，统一销毁线程池，从而有效减少线程频繁创建与销毁所带来的开销。

可以从下面这个图直观的了解线程池模型。

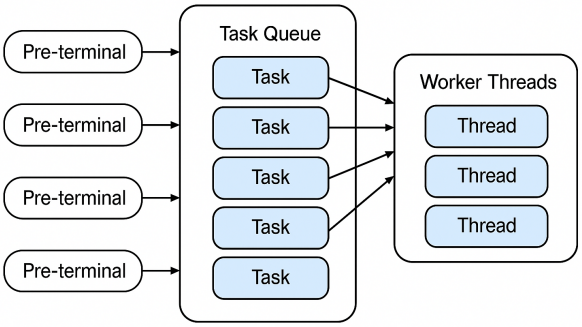


图 4.6: 线程池模型

Pthread 与 OpenMP 在实现线程池上存在一定差异。

对于 Pthread, 为了实现线程池，我们需要先定义一个 ThreadsPool 类，其中封装了各个线程执行任务的函数

```
1  class ThreadPool
2   {
3   public:
4       ThreadPool(size_t num_threads);
5       ~ThreadPool();
```

```
        void enqueue(TaskBase* task);
        void stop();

    private:
        vector<pthread_t> workers;
        queue<TaskBase*> tasks;

        mutex queue_mutex;
        condition_variable_any condition;
        bool stop_flag = 0;

        static void* worker_thread(void* arg);
        void run();
    };
```

其中构造函数用于创建线程

```
    ThreadPool::ThreadPool(size_t num_threads)
    {
        for(size_t i = 0;i < num_threads;++i)
        {
            pthread_t thread;
            pthread_create(&thread,nullptr,ThreadPool::worker_thread,this);
            workers.push_back(thread);
        }
    }
```

而析构函数则用于线程的销毁

```
    ThreadPool::~ThreadPool()
    {
        stop();
    }

    void ThreadPool::stop()
    {
        {
            lock_guard<mutex> lock(queue_mutex);
            stop_flag = 1;
        }
        condition.notify_all();
        for (auto& t : workers)
```

```
14          {
15              pthread_join(t, nullptr);
16          }
17      }
```

worker_thread 和 run 函数则用于让每个线程执行任务。

```
1       void* ThreadPool::worker_thread(void* arg)
2       {
3           static_cast<ThreadPool*>(arg)->run();
4           return nullptr;
5       }
6       void ThreadPool::run()
7       {
8           while (true)
9           {
10              TaskBase* task = nullptr;
11              {
12                  unique_lock<mutex> lock(queue_mutex);
13                  condition.wait(lock, [&] { return stop_flag || !tasks.empty(); });
14                  if (stop_flag && tasks.empty()) return;
15                  task = tasks.front();
16                  tasks.pop();
17              }
18              TaskBase::runTask(task);
19          }
20      }
```

enqueue 函数则是将新的任务构建，并加入到线程池中, 处于同步的需要，配合 PriorityQueue 的类函数 submit_task 使用

```
1       void ThreadPool::enqueue(TaskBase* task)
2       {
3           {
4               lock_guard<mutex> lock(queue_mutex);
5               tasks.push(task);
6           }
7           condition.notify_one();
8       }
9       void PriorityQueue::submit_task(segment* a,const string& pre_terminal
10      ,int start_index,int end_index)
11      {
```

```
        remaining_tasks++;
        pool.enqueue(new Fill_Task(this,a,pre_terminal,start_index,end_index));
    }
```

对于每个任务的创建和流程则通过基类 TaskBase 及其子类 Fill_Task 实现，定义了具体的任务，即将最后一个 Segment 填充入给定的 Pre_terminal

```
    struct TaskBase
    {
        virtual ~TaskBase(){}
        virtual void run() = 0;
        static void* runTask(void* arg)
        {
            unique_ptr<TaskBase> task(static_cast<TaskBase*>(arg));
            task->run();
            return nullptr;
        }
    };
    class Fill_Task : public TaskBase
    {
        public:
            Fill_Task(PriorityQueue* s,segment* seg,const string& pt,int start,int end)
            :self(s),a(seg),pre_terminal(pt),start_index(start),end_index(end) {}

            void run() override
            {
            TaskGuard guard(self); // 确保任务完成时递减 remaining_tasks

            vector<string> local_guesses;
            local_guesses.reserve(end_index - start_index);
            int local_count = 0;

            for (int i = start_index; i < end_index; ++i)
            {
                if (i >= a->ordered_values.size())
                {
                    cerr << "Index out of bounds: " << i << " >= " << a-    >ordered_values.siz
                    break;
                }
                string& suffix = a->ordered_values[i];
                string guess = pre_terminal + suffix;
                local_guesses.emplace_back(guess);
```

```
                ++local_count;
            }

            pthread_mutex_lock(&self->mutex);
            self->guesses.insert(self->guesses.end(), local_guesses.begin(), local_guesses.end(
            self->total_guesses += local_count;
            pthread_mutex_unlock(&self->mutex);

            // guard 会自动触发剩余任务计数递减
        }
    private:
    PriorityQueue*  self;
    segment* a;
    string pre_terminal;
    int start_index;
    int end_index;
};
```

其中为了各个线程之间的同步，在 PriorityQueue 类中增设了计数器 remaining_task, 统计总的需要完成的任务，以及 condition_variable_any 用来等待所有任务完成，从而实现一个线程安全的任务完成等待机制，从而避免了传统线程池算法中 sleep 带来的巨额开销。为了防止某个任务未成功执行或者异常退出导致的死锁，增设了 TaskGuard 类，保证任务完成时 remaining_task 总是减少的，这是 RAII 风格的技术，用于保证在作用域退出时一定执行清理操作。

例如：如果创建的某个任务无法被很好执行时，在本代码中就是 total_guess 大于某个阈值时需要对此时 guesses 中的所有口令进行取出 hash，但是此时线程池中仍然有部分任务未完成，此时会出现同时向 guesses 中写入也从 guesses 读取的现象，从而导致了数据竞争，死锁，崩溃等现象，而我们加入这个 TaskGuard 保证了当线程池中的所有任务完成了再进行 hash 的正确流程。

```

class TaskGuard
{
    public:
        TaskGuard(PriorityQueue* q) : queue(q), dismissed(false) {}
        ~TaskGuard()
        {
            if (!dismissed && queue)
            {
                if (--queue->remaining_tasks == 0)
                {
                    std::lock_guard<std::mutex> lock(queue->done_mutex);
                    queue->done_condition.notify_all();
                }
```

```
            }
        }
        void dismiss() { dismissed = true; }

    private:
        PriorityQueue* queue;
        bool dismissed;
};

```

而对于 OpenMP，需要注意的是，实际上 OpenMP 在执行多线程并行时，是使用了线程池的，即在每次进入 generate 函数时，并非像 pthread 一样创建销毁线程，而是激活这个对应线程池。

但是在图 56 中显示其性能远不如实现了线程池的 pthread（其性能在 O2 优化下比进行了阈值优化的版本更好），是因为实现了线程池的 pthread 的在对不同的 pre_terminal 进行填充时，将其封装为任务提交至线程池中，无论任务是否完成，一旦提交完成就会退出执行下一个 pre_terminal 的填充；而 OpenMP 的线程池的提交会在线程池中的任务完成前将提交入口关闭，只有完成了这个 pre_terminal 完成了填充后才会执行下一个 pre_terminal 的填充，因此其性能不如 pthread 的版本。

因为这个特性，想要实现异步的任务执行，需要对 PCFG 算法的框架进行大改，不在本次作业的范畴内，因此不做考虑。

为了测试性能，首先对各个算法进行了比较，其中对于 Threshold 优化 threshold 均采用 100000，且均采用 O2 优化，与 SIMD hash。测试结果如下

	串行	Pthread	OpenMP	Threshold(Pthread)	Threshold(OpenMP)	ThreadPool
GuessTime(s)	0.145657	0.331677	0.172314	0.122453	0.119001	**0.103692**

表 8: 性能对比

虽然从图 8 可以看出 ThreadPool 具有良好的性能，但是提升也并非显著，因此在此基础上，我增大问题规模，以 10000000 为开始，步长为 10000000, 终止于 100000000，对各个算法进行性能测试。

测试结果如下

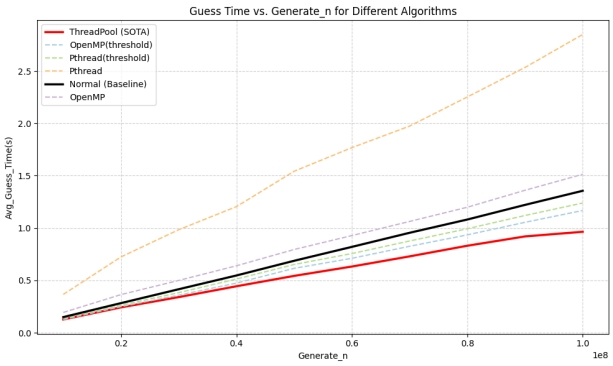


图 4.7: 各个算法随问题规模所需时间

可以看到随着问题规模的增长，ThreadPool 的性能提升变得更加显著，说明了线程池优化方法的有效性。

此外，从图 14.7 以及表格 8 都能看出在同样的问题上，OpenMP 的性能均比 Pthread 更加突出，这是其出于编译器优化带来的红利，但是 pthread 更加底层的风格使得其能够更加自由地实现开发者地想法，而 OpenMP 由于其对并行区的限制导致其很多功能的实现受到了约束。可以认为 Pthread 具有更高的” 上限”。

4.4 对于并行化的思考

一个在并行设计中的常见思想是：并行一定会付出额外的代价；并行化之后总的资源消耗往往是大于串行程序的，并行化的过程实际上是充分利用串行程序所不能利用的设备资源。

在本次作业中，串行算法只能利用一个线程的资源，而利用多个线程，多个 CPU 的资源，此外由于现代 CPU 往往每个核心都有独立的 L1/L2 缓存，通过多线程，可以更加充分地利用这些独立缓存，从而提高总内存带宽利用率。

由此有了如下思考：

- 并行化适合处理怎样的问题?
- 什么时候并行化更合适?

对于第一个问题，结合本问题以及其它并行任务，可以知道，对于计算密集型，数据无依赖的任务可以采取并行的方式。

在本次作业中，我们选择对最后一个 Segment 的填充部分进行并行处理，这是因为每个口令的填充过程相互之间没有数据依赖，仅共享 pre_terminal 结构，可以视为多个独立的子任务，非常适合并行。而在 pre_terminal 的生成阶段，由于其生成依赖前一个结构，存在强相关性，无法简单并行处理。

抽象来说，**只有当任务能够划分为多个“相对独立”的子任务时**，才适合多核并行处理。

对于第二个问题，虽然一个问题可以被划分为多个小的子问题，但是并不是一旦满足这个条件我们就采用并行，这是因为进行**并行会带来额外的开销**，例如不同线程之间的同步通信，SIMD 中的数据向量化打包解包等等，因此只有数据规模达到一定数量，优化后完成任务的开销显著大于并行额外的开销时，才适合进行并行。

作业中的一个具体例子是基于 threshold 的优化。通过设置任务量阈值（threshold），我们只对规模较大的任务采用并行化，而对较小任务仍使用串行方式。这是一种对并行“收益与成本”的合理权衡：**只有当并行带来的收益显著大于其开销时**，才应并行处理。

参考文献

[1] Ziyi Huang, Ding Wang, and Yunkai Zou. Prob-hashcat: Accelerating probabilistic password guessing with hashcat by hundreds of times. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, page 674 – 692. Association for Computing Machinery, 2024.

[2] Jerry Ma, Weining Yang, Min Luo, and Ninghui Li. A study of probabilistic password models. In *2014 IEEE Symposium on Security and Privacy*, pages 689–704. IEEE, 2014.