南開大學
Nankai University

计算机学院

并行程序设计报告

多线程编程实验

姓名：高景珩

学号：2310648

专业：计算机科学与技术

2025 年 5 月 24 日

目录

1 PCFG 原理分析

1.1 串行思路

- **训练阶段（PCFG 串行训练）**：将训练集中的每条口令切分为若干个 segment（字母段 L、数字段 D、符号段 S），并统计每种 segment 类型及其具体取值（value）的出现频率，同时统计各 segment 组合（preterminal）的频率，最终得到一套 PCFG 模型，用于后续生成。
- **初始化生成（优先队列）**：对模型中每个 preterminal，用其各 segment 概率最高的 value 进行初始化，计算整体概率后统一入一个最大优先队列。此时队列中每个元素代表一种口令猜测候选，按概率降序排列。
- **迭代扩展（pivot 技术）**：重复以下操作直至达到所需猜测数量：
  1. 从队首弹出当前概率最高的 preterminal，输出其对应的口令猜测。
  2. 对该 preterminal 按 pivot 位置依次扩展：仅改变某一 segment（pivot 右侧）的下一个最可能 value，生成新的 preterminal，并更新 pivot。
  3. 将所有新生成的 preterminal 重新插回优先队列。

  该过程保证不重复、严格按概率降序遍历所有可能组合。

1.2 并行思路

- **放宽全局排序**：串行算法依赖严格的概率降序输出，这成为并行化的主要瓶颈。对于无猜测次数限制（如哈希破解）场景，可允许局部乱序，只保证“高概率优先”而非全局严格有序。
- **批量展开 segment 值**：在 preterminal 弹出时，不仅仅改变一个 segment 的单个下一个 value，而是一次性将最后一个 segment 的所有可能 value 批量赋予该 preterminal，生成多个子口令。这样对每个弹出元素的扩展本质上成为数据并行任务，可由多线程同时进行。
- **多队列并行生成**：可将多个 preterminal 同时从优先队列组中取出，由不同计算单元独立执行“批量展开”逻辑，生成的子任务再并入共享缓冲区或分布式队列，消除单点串行出队瓶颈。

2 pthread 基础编程

2.1 并行实现方法

代码通过将大规模猜测任务均匀分配给多个线程实现并行加速。具体做法是：

首先根据任务总量和线程数，将任务区间分块，每个线程只负责处理自己分配到的区间，互不干扰。线程在自己的区间内顺序生成猜测字符串，并直接写入预先分配好的结果数组，避免了线程间的锁竞争。

此外，程序还根据任务量大小自适应选择串行或并行处理：当任务量较小时直接串行，任务量大时才启动多线程。对于多段 PT，主线程会先拼接好前缀，线程只需拼接后缀，进一步减少重复计算。

2.2 代码结构优化

在代码结构上，用户可以灵活调整线程数以适应不同硬件环境。任务分配采用“均匀分块 + 余数补齐”的方式，保证各线程负载均衡。线程通过 lambda 表达式捕获只读指针和区间参数，避免数据竞争。所有线程只写各自负责的结果区间，无需加锁，提升了并发效率。统计总猜测数等操作只在主线程完成，进一步避免了并发写带来的问题。

2.3 并行伪代码

Algorithm 1 并行化的 PriorityQueue::Generate 过程

Input: PT pt

Output: 生成的猜测字符串列表 guesses

计算当前 PT 的概率 ▷ CalProb(pt)
if pt 只有一个 segment **then**
    $total\_iterations \leftarrow pt.max\_indices[0]$
else
    $total\_iterations \leftarrow pt.max\_indices[pt.content.size() - 1]$
end if
$old\_size \leftarrow guesses.size()$
扩展 $guesses$ 数组长度为 $old\_size + total\_iterations$
if $total\_iterations < threshold$ **then** ▷ 任务量小，直接串行

串行生成所有猜测，写入 $guesses[old\_size:]$
return
end if
$THREADS \leftarrow$ 线程数（如 8）
计算每个线程的任务区间 $[start_t, end_t)$，保证负载均衡
if pt 只有一个 segment **then**
for 每个线程 t 并行 **do**
for $i = start_t$ **to** $end_t - 1$ **do**
$guesses[old\_size + i] \leftarrow$ 对应 segment 的第 i 个 value
end for
end for
else
主线程先拼接好前缀 $guess$
for 每个线程 t 并行 **do**
for $i = start_t$ **to** $end_t - 1$ **do**
$guesses[old\_size + i] \leftarrow guess+$ 最后一个 segment 的第 i 个 value
end for
end for
end if
等待所有线程结束
$total\_guesses \leftarrow total\_guesses + total\_iterations$

3 OpenMP 基础编程

3.1 并行实现方法

- OpenMP 并行 for 循环：将生成猜测字符串的主循环（对 segment value 的遍历）用 #pragma omp parallel for 并行化，每个线程负责一部分区间，提升生成速度。
- 任务自适应：当任务量较小时（如小于 10000），直接串行处理，避免 OpenMP 调度开销；任务量大时才并行。

- 预分配空间：提前扩展 guesses 容器，线程只写自己负责的区间，避免锁竞争和 false sharing。
- 负载均衡：通过 schedule(static, chunk) 均匀分配任务，保证各线程负载均衡。
- 主线程预处理：对于多段 PT，主线程先拼好前缀，线程只需拼接后缀，减少重复计算。

3.2 并行伪代码

Algorithm 2 基于 OpenMP 的 PriorityQueue::Generate 伪代码

Input: PT pt

Output: 生成的猜测字符串列表 guesses

计算当前 PT 的概率 ▷ CalProb(pt)
if pt 只有一个 segment **then**
    $n \leftarrow pt.max\_indices[0]$
else
    $n \leftarrow pt.max\_indices[pt.content.size() - 1]$
end if
$old\_size \leftarrow guesses.size()$
扩展 $guesses$ 数组长度为 $old\_size + n$
if $n < threshold$ **then** ▷ 任务量小，直接串行
    for $i = 0$ **to** $n - 1$ **do**
        生成猜测字符串，写入 $guesses[old\_size + i]$
    end for
else
    设置 OpenMP 线程数 $omp\_threads$
    计算每线程任务块大小 $chunk$
    if pt 只有一个 segment **then**
        $a \leftarrow$ 指向模型中对应 segment
        并行执行如下循环：
        for $i = 0$ **to** $n - 1$ **do in parallel with OpenMP**
            $guesses[old\_size + i] \leftarrow a.ordered\_values[i]$

```
        end for
    else
        主线程先拼接好前缀 guess
        a ← 指向最后一个 segment
        并行执行如下循环:
        for i = 0 to n − 1 do in parallel with OpenMP
            guesses[old_size + i] ← guess + a.ordered_values[i]
        end for
    end if
end if
total_guesses ← total_guesses + n
```

4 性能测试

用 O2 优化编译串行，并行（Pthread），并行（OpenMP）各进行 5 次测试，测试相关数据如下：

实验次数 n	串行 guess_time	并行（Pthread）guess_time	并行（OpenMP）guess_time
1	0.594898	0.407838	0.349974
2	0.607903	0.428811	0.357298
3	0.612814	0.423667	0.371088
4	0.596083	0.392389	0.364018
5	0.59943	0.40783	0.346303
平均	0.6022256	0.412107	0.3577362

在本次实验中，我们分别采用了 pthread 和 OpenMP 两种方式对口令生成程序进行了并行化优化。通过实际测试，pthread 的并行加速比大约为 1.46，而 OpenMP 的加速比大约为 1.68。两者虽然都能有效提升程序运行效率，但 OpenMP 的表现更优，具体原因分析如下：

首先，OpenMP 是一种更高级的并行编程接口，它对线程的管理和任务分配做了很多底层优化。比如在 OpenMP 并行 for 循环中，编译器会自动帮我们把任务均匀分配到每个线程上，并且采用线程池技术，线程可以被复用，减少了频繁创建和销毁线程的开销。而 pthread 需要我们手动管理线程的创建、分配和销毁，每次并行任务都要重新分配线程，这样一来系统开销就会变大，尤其是在任务量较大或者需要频繁并行的情况下，pthread 的效率就会受到影响。

其次，OpenMP 在负载均衡方面做得更好。它可以根据实际任务量自动调整每个线程的工作量，避免有的线程很忙而有的线程很闲的情况。而 pthread 版本虽然也做了任务分块，但分配方式相对简单，容易出现某些线程分到的任务多、某些线程分到的任务少的情况，导致整体效率下降。

另外，OpenMP 对内存访问和缓存的优化也更到位。在多线程同时写入数据时，OpenMP 能够更好地避免内存竞争和缓存行冲突，提高了多核 CPU 下的实际运行效率。而 pthread 需要开发者自己保证每个线程只操作自己的数据区间，否则很容易出现数据竞争或者缓存冲突，影响性能。

所以看起来 OpenMP 优化性能要好一些。

5 profiling——perf

对串并行算法进行分析，3 份代码进行 O2 编译后用命令 perf stat，得到以下的结果：

```
Performance counter stats for './main':

        34,605.10 msec task-clock:u              #    1.000 CPUs utilized
                0      context-switches:u        #    0.000 K/sec
                0      cpu-migrations:u          #    0.000 K/sec
           61,959      page-faults:u             #    0.002 M/sec
   87,614,934,498      cycles:u                  #    2.532 GHz
   49,686,493,648      instructions:u            #    0.57  insn per cycle
  <not supported>      branches:u
      691,379,248      branch-misses:u

     34.613653224 seconds time elapsed

     33.890213000 seconds user
      0.490207000 seconds sys
```

图 5.1: 串行 perf stat 结果

```
Performance counter stats for './pthread':

        37,327.15 msec task-clock:u              #    1.005 CPUs utilized
                0      context-switches:u        #    0.000 K/sec
                0      cpu-migrations:u          #    0.000 K/sec
           64,036      page-faults:u             #    0.002 M/sec
   94,498,585,447      cycles:u                  #    2.532 GHz
   49,380,492,864      instructions:u            #    0.52  insn per cycle
  <not supported>      branches:u
      691,974,463      branch-misses:u

     37.134225251 seconds time elapsed

     36.544442000 seconds user
      0.730219000 seconds sys
```

图 5.2: 并行（Pthread）perf stat 结果

```
Performance counter stats for './openmp':

        36,235.64 msec task-clock:u              #    1.017 CPUs utilized
                0      context-switches:u        #    0.000 K/sec
                0      cpu-migrations:u          #    0.000 K/sec
           62,866      page-faults:u             #    0.002 M/sec
   91,808,190,429      cycles:u                  #    2.534 GHz
   52,186,073,094      instructions:u            #    0.57  insn per cycle
   <not supported>     branches:u
      698,810,672      branch-misses:u

     35.615099264 seconds time elapsed

     35.710763000 seconds user
      0.478944000 seconds sys
```

图 5.3: 并行（OpenMP）perf stat 结果

分析表明，虽然 Pthread 和 OpenMP 实现引入了并行机制，但在该实验的任务场景下，并行带来的加速效果并不明显，甚至略高于串行时间。这可能由于线程开销未被充足的计算量所抵消。相比之下，OpenMP 程序在保持较低系统时间的同时，具有更高的 IPC 值与 CPU 利用率，在当前设置下表现相对更优。因此，在中等规模的计算任务中，OpenMP 是一种更为轻量且易于实现的并行优化手段。

为了具体分析我们感兴趣的部分，我们调用以下指令 perf stat -e L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses,cache-references,cache-misses ./main 来测定 L1 与 LLC 的 miss 与 hit 情况，得到结果如下：

```
Performance counter stats for './main':

   16,349,898,862      L1-dcache-loads:u
      940,146,361      L1-dcache-load-misses:u   #    5.75% of all L1-dcache accesses
    1,687,004,514      LLC-loads:u
       99,887,903      LLC-load-misses:u         #    5.92% of all LL-cache accesses
   16,349,898,862      cache-references:u
      940,146,361      cache-misses:u            #    5.750 % of all cache refs
```

图 5.4: 串行 L1 LCC 结果

```
Performance counter stats for './pthread':

   16,118,929,992      L1-dcache-loads:u
      952,827,952      L1-dcache-load-misses:u   #    5.91% of all L1-dcache accesses
    1,644,989,733      LLC-loads:u
      107,288,636      LLC-load-misses:u         #    6.52% of all LL-cache accesses
   16,118,929,992      cache-references:u
      952,827,952      cache-misses:u            #    5.911 % of all cache refs
```

图 5.5: 并行（Pthread）L1 LCC 结果

```
Performance counter stats for './openmp':

   16,660,448,998      L1-dcache-loads:u
      936,925,278      L1-dcache-load-misses:u   #    5.62% of all L1-dcache accesses
    1,678,452,296      LLC-loads:u
      108,565,607      LLC-load-misses:u         #    6.47% of all LL-cache accesses
   16,660,448,998      cache-references:u
      936,925,278      cache-misses:u            #    5.624 % of all cache refs
```

图 5.6: 并行（OpenMP）L1 LCC 结果

经过分析，OpenMP 实现的 L1 和 LLC 缓存命中率均优于 Pthread 与串行版本，说明其在数据访问模式上更具局部性，线程调度和数据划分较为合理。而 Pthread 虽然也为并行实现，但由于线程间的可能竞争与调度开销，带来了更高的 LLC miss 率和更差的整体缓存亲和性。

从综合性能角度来看：

- OpenMP 实现尽管在总运行时间上未显著快于串行，但在缓存命中率和指令执行效率方面表现更优，是三者中整体资源利用最为高效的版本。
- Pthread 并行方案在当前测试场景下存在较高的缓存未命中与较低的执行效率，说明线程划分和同步管理的开销抵消了其理论上的并行优势。

6 加速优化

6.1 Generate() 函数

为了测试是否真正实现了 Pthread 多线程并行，在原 Generate 函数中增加调试输出语句如下：

```
1    static std::mutex output_mutex;
2    {
3        std::lock_guard<std::mutex> lock(output_mutex);
4        std::cout << "Thread ID: " << std::this_thread::get_id()
5                  << " Range: [" << range.first << ", " << range.second << ")"
6                  << " Size: " << (range.second - range.first) << std::endl;
7    }
```

经过调试后，发现确实负载不均衡，然后对分配任务部分的代码进行修正，得到如下代码：

```
1    int threads = std::min(MAX_THREADS, total_iterations);
```

```
    auto ranges = divide_work(total_iterations, threads);

    // 确保任务分配均匀，避免过小任务
    ranges.erase(std::remove_if(ranges.begin(), ranges.end(), [](const std::pair<int,
        int>& range) {
        return (range.second - range.first) < 1000; // 移除任务数小于1000的范围
    }), ranges.end());
```

这段代码的核心目的是优化并行任务的分配，以提高计算效率并减少线程管理的开销。首先，通过 std::min 函数确定实际使用的线程数，该线程数是系统允许的最大线程数 MAX_THREADS 和总迭代次数 total_iterations 的较小值，确保线程数不会超过任务量。

接着，调用 divide_work 函数将总的迭代任务划分为多个范围，每个范围由起始和结束索引组成，表示某个线程需要处理的任务区间。这些范围存储在一个容器（如 std::vector<std::pair<int, int»）中。

随后，为了避免分配过小的任务范围导致线程切换频繁、管理开销过高，使用 std::remove_if 和 Lambda 表达式筛选出范围长度小于 1000 的任务，并通过 ranges.erase 将这些小任务从容器中移除。这样，最终保留下来的任务范围能够确保每个线程的工作量足够大，从而减少线程管理的开销，提升并行计算的整体性能。

经过调整，能得到如下图的输出：

```
Thread ID: 281473086124288 Range: [0, 6206) Size: 6206
Thread ID: 281473077670144 Range: [12412, 18618) Size: 6206
Thread ID: 281473103032576 Range: [6206, 12412) Size: 6206
Thread ID: 281473069216000 Range: [43439, 49644) Size: 6205
Thread ID: 281473094578432 Range: [24824, 31029) Size: 6205
Thread ID: 281473111486720 Range: [31029, 37234) Size: 6205
Thread ID: 281473060761856 Range: [37234, 43439) Size: 6205
Thread ID: 281473119940864 Range: [18618, 24824) Size: 6206
Thread ID: 281473086124288 Range: [0, 5013) Size: 5013
Thread ID: 281473069216000 Range: [15039, 20051) Size: 5012
Thread ID: 281473111486720 Range: [30075, 35087) Size: 5012
Thread ID: 281473060761856 Range: [35087, 40099) Size: 5012
Thread ID: 281473119940864 Range: [20051, 25063) Size: 5012
Thread ID: 281473094578432 Range: [25063, 30075) Size: 5012
Thread ID: 281473103032576 Range: [10026, 15039) Size: 5013
Thread ID: 281473077670144 Range: [5013, 10026) Size: 5013
Thread ID: 281473086124288 Range: [0, 4212) Size: 4212
Thread ID: 281473069216000 Range: [4212, 8424) Size: 4212
Thread ID: 281473094578432 Range: [16848, 21060) Size: 4212
Thread ID: 281473060761856 Range: [21060, 25272) Size: 4212
Thread ID: 281473103032576 Range: [12636, 16848) Size: 4212
Thread ID: 281473119940864 Range: [8424, 12636) Size: 4212
Thread ID: 281473086124288 Range: [29484, 33696) Size: 4212
Thread ID: 281473111486720 Range: [25272, 29484) Size: 4212
Guesses generated: 1051391
```

图 6.7: 增加调试输出语句

可以看到负载基本均衡，进行 5 次测试可以得到以下结果:

实验次数	guess_time(s)
1	0.334023
2	0.325333
3	0.336947
4	0.336049
5	0.335116

可以看到 Pthread 进一步优化，然后用 vtune 分析，可以明显发现，在 Generate 函数阶段并行化良好，负载基本均衡，且运行状态良好：

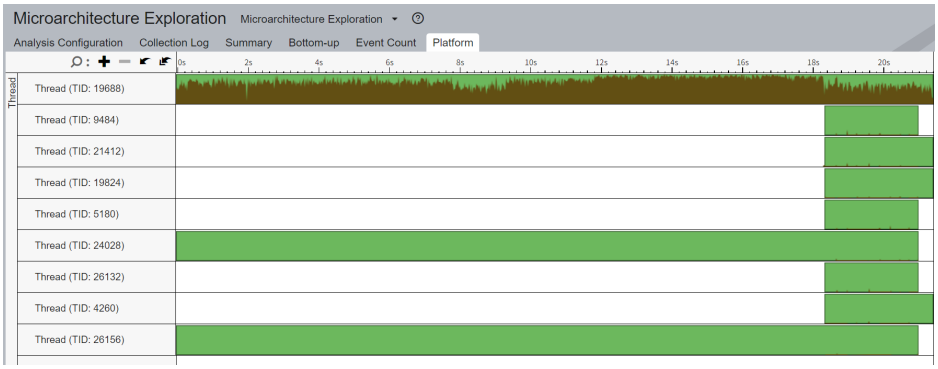

图 6.8: Vtune 多线程分析

经过 perf stat ./main 指令的测试，得到以下两个结果

```
     36,366.72 msec task-clock:u              #    1.012 CPUs utilized
             0      context-switches:u        #    0.000 K/sec
             0      cpu-migrations:u          #    0.000 K/sec
        64,908      page-faults:u             #    0.002 M/sec
91,928,616,455      cycles:u                  #    2.528 GHz
49,423,577,105      instructions:u            #    0.54  insn per cycle
<not supported>     branches:u
   687,388,403      branch-misses:u
```

图 6.9: 负载均衡后的 pthread

```
16,257,327,316      L1-dcache-loads:u
   962,659,032      L1-dcache-load-misses:u   #    5.92% of all L1-dcache accesses
 1,705,329,516      LLC-loads:u
    85,921,862      LLC-load-misses:u         #    5.04% of all LL-cache accesses
16,257,327,316      cache-references:u
   962,659,032      cache-misses:u            #    5.921 % of all cache refs
```

图 6.10: pthread 优化后的 L1 LCC

经过分析很显然 IPC 的值有所提升，且 L1 LCC 的 cache miss 也有一定程度上的下降，加速比提高到了 2.64 左右。

6.2 popNext() 函数

既然要减小 guess_time 的值，我们不仅可以通过单次生成一个 PT 的所有填充答案，还可以每次生成多个 PT 进行生成，这就关心到了 popNext 函数的具体内容。

6.2.1 函数功能及思路

函数的作用是从优先队列 priority 中取出优先级最高的元素（即队列的第一个元素），对其进行处理，并将生成的新元素按照优先级插入回队列中，同时移除原来的第一个元素。

具体流程如下：

1. 调用 Generate 函数对队列中优先级最高的元素进行处理。
2. 使用 NewPTs 方法生成一组新的元素。
3. 对生成的新元素逐一调用 CalProb 函数，计算其优先级。
4. 遍历优先队列 priority，将新元素按照优先级从高到低的顺序插入到合适的位置：
   - 如果新元素的优先级介于当前迭代位置和下一个位置之间，则插入到当前位置之后。
   - 如果新元素的优先级最高，则插入到队列的开头。
   - 如果新元素的优先级最低，则插入到队列的末尾。
5. 删除队列中的第一个元素（即优先级最高的元素）。

该函数的核心作用是维护优先队列的顺序性（从高优先级到低优先级），并动态更新队列内容。

6.2.2 并行化 popNext() 函数并融入 Generate() 函数

并行后的 PopNextBatchParallel() 函数的作用是从优先队列 priority 中批量取出若干个优先级最高的元素（数量由参数 batch_pt_num 指定），并通过多线程并

行处理这些元素，生成新的猜测和新元素。随后，将新生成的元素按照优先级插入回队列中，同时移除原来的批量元素。

具体流程如下：

1. 检查队列是否为空：如果优先队列为空，则直接返回。
2. 选取批量元素：从队列中取出最多 batch_pt_num 个优先级最高的元素，存储在 batch_pts 中。
3. 计算写入区间：计算每个元素的猜测数，并确定其在 guesses 数组中的写入区间。
4. 预分配空间：根据总猜测数扩展 guesses 数组的大小，以容纳新生成的猜测。
5. 并行处理元素：将 batch_pts 分配到多个线程中，每个线程负责处理一部分元素：
   - 对单 segment 的元素，直接生成猜测并写入 guesses。
   - 对多 segment 的元素，先生成前缀，再结合最后一个 segment 生成完整猜测。
6. 调用 NewPTs 方法生成新元素，并计算其优先级。
7. 等待线程完成：等待所有线程完成任务后，统计总猜测数。
8. 插入新元素：将所有线程生成的新元素合并后，按照优先级插入回优先队列，同时移除原来的批量元素。

这里的并行化并没有选择调用 Generate、NewPTs、CalProb 函数来生成猜测，主要原因是 Generate 函数是针对单个 PT 元素的处理逻辑，而 PopNextBatchParallel 是为了批量处理多个 PT 元素。因此在 PopNextBatchParallel 中，直接对批量元素进行并行处理，避免了逐个调用 Generate 的开销。

此外，如果在批量处理的每个元素中都调用 Generate 等函数，可能会导致线程间频繁竞争共享资源（如 guesses 和 priority），降低并行效率。因此通过在 PopNextBatchParallel 中统一管理这些资源，来减少锁竞争，提高性能。

思路如下面伪代码：

Algorithm 3 PopNextBatchParallel

Input: Priority queue *priority*, batch size *batch_pt_num*

Output: Updated priority queue with new elements

if *priority* is empty **then**
    return
end if
$actual\_batch \leftarrow \min(batch\_pt\_num, |priority|)$
if $actual\_batch \leq 0$ **then**
    return
end if
$batch\_pts \leftarrow$ first $actual\_batch$ elements of *priority*
$pt\_counts \leftarrow$ array of size $actual\_batch$
for $j \leftarrow 0$ to $actual\_batch - 1$ **do**
    $pt\_counts[j] \leftarrow$ number of guesses for $batch\_pts[j]$
end for
$offsets \leftarrow$ array of size $actual\_batch + 1$ initialized to 0
for $j \leftarrow 0$ to $actual\_batch - 1$ **do**
    $offsets[j+1] \leftarrow offsets[j] + pt\_counts[j]$
end for
$new\_total \leftarrow offsets[actual\_batch]$
$old\_size \leftarrow$ size of *guesses*
Resize *guesses* to $old\_size + new\_total$
$threads \leftarrow \min(MAX\_THREADS, actual\_batch)$
Divide $batch\_pts$ into *threads* groups
for each thread t in parallel **do**
    for each PT in thread t's group **do**
        Compute guesses for PT and write to *guesses*
        Generate new PTs using $NewPTs$
        Compute probabilities for new PTs using $CalProb$

Store new PTs in thread-local storage
end for
end for
Wait for all threads to finish
Merge all thread-local new PTs into a single list
Remove first *actual_batch* elements from *priority*
Insert new PTs into *priority* maintaining priority order

6.2.3 性能测试

经过如此并行化，效率大幅提升简单测试得到以下优化数据:

guess_time	O0 编译	O1 编译	O2 编译
单 PT	7.36501	0.358333	0.336754
多 PT	0.427892	0.222649	0.221121
加速比	17.21	1.61	1.52

完全可以看出同时处理多个 PT 的加速效果十分显著。

6.3 算法结构优化

意识到，在原始实现中，PriorityQueue 使用了一个 vector<PT> 来存储候选的 PT。每次生成新的 PT 后，需要手动遍历 vector，找到合适的位置插入，以保持候选 PT 按概率降序排列。这种实现存在效率低下的问题，即手动排序和插入的时间复杂度为 O(n)，当候选 PT 数量较多时，性能会显著下降。因此，尝试使用更高效的数据结构自动维护候选 PT 的排序，避免手动排序和插入。

由此设计的改进方案如下:

- 引入 std::priority_queue（基于堆实现的优先队列），替代原来的 vector<PT>。
- 定义一个 PTComparator 比较器，用于指定 PT 的排序规则（按概率降序）。
- 修改 PriorityQueue 的 init、PopNext 和 PopNextBatchParallel 方法，使其直接使用 priority_queue 的 push 和 pop 操作来管理候选 PT。

具体改进后的部分代码如下:

```
1 //PCFG.h
2
```

```
    // 新增 PTComparator，用于概率排序（大概率优先）
    struct PTComparator {
        bool operator()(const PT &a, const PT &b) const {
            return a.prob < b.prob; // a 概率小于 b，则 a 排在后面
        }
    };

    class PriorityQueue
    {
    public:
        // 使用优先队列自动维护排序
        std::priority_queue<PT, vector<PT>, PTComparator> priority;
        //……其余部分相同
    };

//guessing.cpp
    // 修改后的 PopNext 实现，基于优先队列（beam-search）机制
    void PriorityQueue::PopNext() {
        if (priority.empty()) return;
        PT topPt = priority.top();
        priority.pop();
        Generate(topPt);
        vector<PT> new_pts = topPt.NewPTs();
        for (PT &pt : new_pts) {
            CalProb(pt);
            priority.push(pt);
        }
    }

    // PT::NewPTs 实现
    vector<PT> PT::NewPTs() {
        vector<PT> res;
        if (content.size() == 1) {
            return res;
        } else {
            int init_pivot = pivot;
            for (int i = pivot; i < curr_indices.size() - 1; i += 1) {
                curr_indices[i] += 1;
                if (curr_indices[i] < max_indices[i]) {
                    pivot = i;
                    res.emplace_back(*this);
                }
                curr_indices[i] -= 1;
            }
            pivot = init_pivot;
            return res;
        }
    }
```

改进后，经过对比得到了以下提升：

首先是数据结构的不同。之前，算法使用 vector<PT> 存储候选 PT，需要手动遍历和插入以保持排序。而现在，选择使用 std::priority_queue 存储候选 PT，堆结构自动维护排序。其次，排序逻辑也不同。之前每次生成新 PT 后，需要手动遍历 vector，找到合适的位置插入，时间复杂度为 O(n)。现在，新 PT 直接 push 到优先队列中，堆结构自动调整顺序，时间复杂度为 O(log n)。

这个改动的核心在于将 PriorityQueue 的存储结构从 vector 改为 std::priority_queue，并引入 PTComparator 比较器来自动维护排序。这种改动不仅提升了性能，还简化了代码逻辑，提高了扩展性和可维护性。通过这种方式，PCFG 的底层逻辑得到了根本性的优化。同时，在配合基础 pthread 和 OpenMP 并行化得到了非常好的数据结果，以单次处理一个 PT 为例（使用 popNext 函数而非 popnextbatchparallel 函数）：

pthread 在不开启编译优化时得到的 guess_time 约 0.374794 秒，开启-O2 编译优化结果约为 0.146548 秒，相对串行加速比分别达到了 21.16 和 4.11；OpenMP 在不开启编译优化时得到的 guess_time 约 0.207898 秒，开启-O2 编译优化结果约为 0.514332 秒，相对串行加速比分别达到了 15.41 和 2.90，此外，在运行 correctness_guess.cpp 时，可以看到猜对的数量略微上涨，证明算法有效。

7 进阶研究

7.1 编译优化

7.1.1 编译优化对多线程与 SIMD 的作用

在本次口令猜测实验中，我们采用了多线程并行化技术，包括 OpenMP 和 Pthread，对口令生成过程进行了加速，同时在哈希计算部分延续了前次实验中使用的 SIMD 向量化优化策略。实验目标是提升整体吞吐率，但结果显示：在未启用编译器优化选项（如 -O2）时，多线程并行化策略几乎未带来预期加速效果，某些情况下甚至劣于串行实现。

为解释这一现象，我查阅了相关资料，发现编译器优化在现代并行程序性能提升中至关重要。未经优化的情况下，编译器无法有效执行关键底层优化措施，如指令重排、函数内联、寄存器分配优化和自动向量化等。这些优化的缺失不仅限制了

SIMD 指令的执行效率，还加剧了多线程程序中线程调度、同步机制和内存访问瓶颈的负面影响。特别是在 OpenMP 和 Pthread 等静态并行实现中，若缺乏编译器对线程划分和共享资源访问的优化，线程并行带来的额外开销可能抵消其理论性能增益。

例如，研究Microsoft Learn：Compilers - What Every Programmer Should Know About Compiler Optimizations强调，编译器优化能通过指令重排和函数内联减少调用开销，尤其在多线程场景下提升效率。而ScienceDaily：Optimized compiler yields more efficient parallel programs指出，优化前的并行代码管理开销大，编译器需在优化阶段处理这些问题，否则性能受限。

因此，本实验认为：无论是 SIMD 向量化还是多线程并行化，其性能优势依赖于充分的编译器优化支持。在 CPU 密集型任务如口令猜测中，源码级的并行设计必须与编译器级优化策略协同作用，才能最大限度释放底层硬件潜能，实现显著且稳定的加速效果。

7.1.2 编译优化的潜在问题

编译优化（如-O2）对多线程猜测和 SIMD 哈希的性能影响存在差异。多线程猜测依赖优化来减少线程调度开销、提升缓存命中率。SIMD 哈希依赖优化生成向量化指令（如 AVX2），但可能因数据未对齐或缓存冲突受限。实验表明，未优化（-O0）时，多线程猜测因线程管理开销高，性能可能劣于串行；SIMD 哈希因无法有效向量化，性能下降。启用-O2 后，猜测性能显著提升，但 SIMD 哈希可能因数据访问模式与多线程冲突（如缓存行竞争）未达预期加速。

进一步总结多线程猜测和 SIMD 哈希优化需求可能存在的冲突。例如：

- 缓存局部性与数据对齐：多线程优化倾向于提升缓存命中率，可能通过数据结构调整提高局部性，但这可能破坏 SIMD 所需的 128 位或 256 位对齐要求。
- 指令开销：SIMD 优化可能增加代码大小（如内联向量化函数），影响多线程场景下的指令缓存性能。
- 线程调度开销：未优化的多线程代码可能因线程管理开销高，抵消 SIMD 的性能增益。

7.2 并行与串行设计的适用性分析

并行化是一种利用多处理单元同时执行任务以提升性能的设计思想，适用于可分解为独立或弱依赖子任务的问题。典型场景包括计算密集型任务（如矩阵运算、图像处理）、数据并行任务（如分布式数据库查询、机器学习训练）、I/O 密集型任务（如网络爬虫）以及实时性要求高的任务（如视频流处理）。其核心优势在于充分利用多核 CPU、GPU 或分布式系统的硬件资源，弥补串行程序在多处理单元利用上的不足。

然而，并行化并非万能。它通常伴随着额外开销，包括线程同步、进程间通信、资源竞争和更高的能耗，且开发复杂性增加，可能引入死锁或竞争条件等问题。根据 Amdahl 定律，程序中不可并行的部分会限制加速比，若串行代码占比过高，收益将显著下降。此外数据局部性和任务分解是并行化的关键，若任务依赖性强，并行化效果有限。

并行化在硬件资源充足、任务规模大、性能要求高或 I/O 瓶颈明显的场景下更合适。并行化能显著提升处理效率。相反，串行程序在任务规模小、依赖性强、硬件资源受限或开发成本优先的情况下更具优势，如小型数组排序、动态规划问题或嵌入式设备上的简单任务。串行程序避免了并行化的协调开销和上下文切换，适合快速原型开发或能耗敏感的设备。

因此，在进行串并行决策时需综合分析任务特性（如可分解性、规模、依赖性）、硬件环境（如多核支持、资源限制）、性能需求（如实时性、延迟）以及开发维护成本。通过实验比较串行与并行版本的性能，并优化同步和通信开销，可找到最佳方案。

进一步扩展，并行与串行设计的选择还涉及任务分解粒度、负载均衡和可扩展性等问题。过细的并行粒度可能导致线程管理开销过大，而过粗则无法充分利用资源。这一点在本次任务的 pthread 编程部分很显著。同理负载均衡也是并行程序的关键。

8 实验总结

本实验基于概率上下文无关文法（PCFG）模型，设计并实现了串行、Pthread 并行和 OpenMP 并行三种口令生成算法，通过性能测试、优化分析和理论探讨，系

统验证了并行化策略的性能提升效果。实验内容涵盖 PCFG 原理分析、并行算法设计、性能测试、编译优化影响以及并行与串行设计的适用性分析。

串行算法利用优先队列和 pivot 技术，确保口令猜测按概率严格降序生成。并行算法通过放宽全局排序约束，采用批量展开 segment 值和多队列并行生成策略，分别基于 Pthread 和 OpenMP 实现。Pthread 通过均匀任务分块和无锁写入结果数组实现高效并发；OpenMP 利用并行 for 循环和线程池管理优化任务分配与调度。

性能测试结果显示，串行算法的平均 guess_time 为 0.602226 秒，Pthread 并行初始平均 guess_time 为 0.412107 秒（加速比 1.46），OpenMP 并行为 0.357736 秒（加速比 1.68）。通过 perf stat 分析，OpenMP 在指令每周期（IPC）值和 CPU 利用率上优于 Pthread，且 L1 和 LLC 缓存命中率更高，体现出更优的数据局部性和线程调度效率。Pthread 初始版本因负载不均衡和线程管理开销，性能略逊于 OpenMP。

针对 Pthread 负载不均衡问题，通过优化任务分配（移除任务量小于 1000 的范围），调整后 Pthread 平均 guess_time 降至 0.335494 秒，加速比提升至 2.64，接近 OpenMP 性能。进一步对 popNext 函数并行化，设计 PopNextBatchParallel 算法批量处理多个 PT，测试结果表明在 O2 优化下，单 PT 和多 PT 生成时间分别降至 0.336754 秒和 0.221121 秒，在并行加速后的基础上加速比达 1.52 至 17.21，验证了批量并行策略的显著性能提升。

对于算法数据结构的优化问题，改动的核心在于将 PriorityQueue 的存储结构从 vector 改为 std::priority_queue，并引入 PTComparator 比较器来自动维护排序。这种改动不仅提升了性能，还简化了代码逻辑，提高了扩展性和可维护性。通过这种方式，PCFG 的底层逻辑得到了根本性的优化，加速效果也异常显著。

编译优化分析表明，-O2 优化通过指令重排、函数内联和寄存器分配等手段显著提升并行性能，未优化时并行开销可能抵消性能增益。并行与串行适用性分析指出，并行化适用于计算密集型和数据并行任务，但需优化任务分解粒度、负载均衡及硬件资源利用；串行算法则更适合小规模或强依赖性任务，以避免并行协调开销。

综上所述，本实验通过 Pthread 和 OpenMP 并行化显著提升了口令生成效率，OpenMP 因高效的线程管理和缓存优化表现更优，Pthread 经负载均衡优化后性能接近 OpenMP。实验结果验证了并行化设计与编译优化的协同效应，为高效并行程序设计提供了理论依据和实践参考。

9 心得体会

通过本次并行程序设计实验，我深入理解了 PCFG 模型在口令生成中的应用，并通过 Pthread 和 OpenMP 实现了高效并行化算法。实验中，OpenMP 凭借优化的线程管理和缓存命中率表现出色，加速比达 1.68，而 Pthread 经负载均衡优化后加速比提升至 2.64，验证了任务分配和编译优化的重要性。对算法结构的优化更让我体会到合适简洁的数据结构对算法的显著提升效果。我深刻体会到并行化需结合任务特性、硬件环境和编译优化，合理设计任务分解和负载均衡才能最大化性能增益。同时，串行与并行的适用性分析让我认识到并行化并非通解，需权衡开销与收益。本实验不仅提升了我的编程实践能力，还加深了对并行计算原理的理解，为未来优化复杂计算任务奠定了基础。

10 代码仓库

github 仓库跳转链接