南開大學
Nankai University

计算机学院

并行程序设计实验报告

PCFG 口令猜测并行化选题:

pthread 多线程实验

OpenMP 多线程实验

姓名：陈语童

学号：2311887

专业：计算机科学与技术

2025 年 5 月 29 日

目录

1 实验环境

SSH 远程：OpenEuler ARM 服务器

课程提供的 OpenEuler 服务器集群，以物理机 + 虚拟机的形式搭建的，目前提供给并行课程使用的共有 1 个主节点 (master_ubss1) 和 17 个计算节点 (master_ubss1 9, node1_ubss1 8)，每个计算节点 8 核。

OpenEuler 是开源、免费的 Linux 发行版操作系统，使用 ARM 架构指令集。

通过 vscode （或 cmd/powershell）进行相关 SSH 配置并进行连接，登入服务器中为学生个人分配的节点，进行 ARM 架构上的并行代码测试。

2 问题描述

2.1 选题：PCFG 口令猜测

2.1.1 口令猜测

口令，就是俗称的“密码”。对于一定长度范围的口令，通过穷举进行暴力破解的效率是极低的。而本选题要做的，就是通过用户口令的语义和偏好规律，制定一定的策略来破解口令，并用**并行化**的方法对整个过程的各个部分进行优化与加速。

本选题中，不考虑个人信息、口令重用这些额外的信息，只考虑非定向的口令猜测。对于一个用户的口令，对其进行猜测的基本策略是：生成一个按照概率降序排列的口令猜测词典，这个词典包括一系列用户可能选择的口令。那么问题就变成了：

- 如何有效生成用户可能选择的口令；
- 如何将生成的口令按照降序进行排列。

在本选题中，选择使用最经典的 **PCFG**（Probabilistic Context-Free Grammar，概率上下文无关文法）模型来进行口令的生成，并且尝试将其并行化，以提升猜测的时间效率。

2.1.2 PCFG 串行训练

PCFG 串行训练流程关键步骤如下：

1. **内容分类**。将口令内容分为三类不同字段（segments）——Letters（字母字段）、Digits（数字字段）、Symbols（特殊字符）
2. **解构口令**。然后将实际每条口令内容用这三种字段类型解构其组成，形成一系列的口令 preterminal（类似于 pattern）。
3. **统计频率**。对不同的 preterminal，以及每个 preterminal 中的每个 segment 的不同值均需统计出现频次。

经过上面三步，PCFG 模型生成。

2.1.3 PCFG 串行猜测生成

猜测生成的本质要求，是构建出一个出现概率降序排列的口令**优先队列**，对其动态更新和从中取猜测值。具体生成策略较为繁琐，此处只做概括：

1. **初始化**。所有 preterminal 中出现概率最高的口令组合先入队列，preterminal 本身的概率越高越优先入队列。同时，赋给一个初始 pivot 值为 0.
2. **出列**。只要队列有口令，就从先入队列的开始取，取到的即作为当前口令猜测值。
3. **变异**。取出的口令不立即放回，而是根据 pivot 值，依次单项更新大于 pivot 值的 segment 值，依然是概率降序入队列，并且新入列的要设置 pivot 值到发生变异的 segment.
4. **遍历**。接着向后遍历和动态更新即可。

通过这种策略生成的优先队列可尽可能保证口令是不重复地以概率降序排列的，从而提高遍历查找效率和命中率。

2.1.4 PCFG 猜测生成并行化 *

这是本选题的重点和最终目标。

PCFG 猜测生成过程似乎比较复杂，难以进行并行化。并行化的最大阻碍，是按照概率降序生成口令这一过程。但 PCFG 往往用于没有猜测次数限制的场景中（例如哈希破解），实际应用中并不需要严格按照降序生成口令。

由此产生的并行化思路很显然：一次取出多个 preterminal，或是一次为某一 segment 分配多个 value 等。

选题代码框架采用的并行化方案如图示：

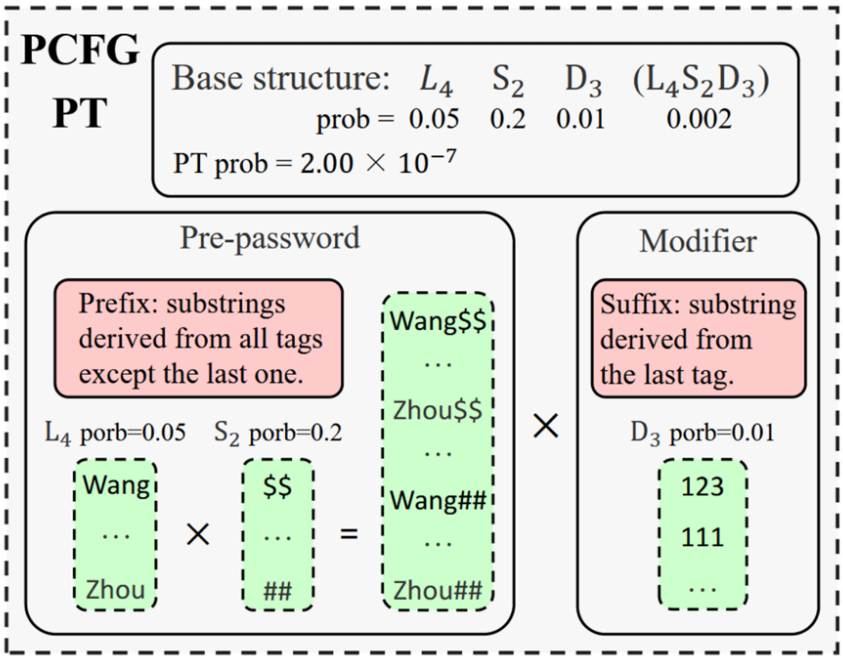

图 2.1: 本代码框架采用的并行化方案

这种方案和原始 PCFG 的区别在于，其对 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. 不难看出，这个过程是可以并行进行的。

本选题的代码框架，已经将这个并行算法用串行的方式进行了实现，实际实验只需将其用并行编程的方式，变成一个并行程序，即可完成 PCFG 的并行化。

2.2 本次实验：pthread 和 OpenMP 多线程实验

本次实验分别用 pthread 和 OpenMP 这两种多线程并发的编程 API，对于 PCFG 的猜测生成过程进行并行化。

2.2.1 pthread

全称： POSIX Threads

平台： Unix/Linux

语言： C/C++

特性：
- 是**底层线程 API**，提供对线程的细粒度控制；
- 包含线程的创建、同步（互斥锁、条件变量等）、终止、连接等操作；
- 使用方式相对**复杂**，需要显式管理线程和同步。

基础 API：

```c
int pthread_create(pthread_t *, const pthread_attr_t *,void * (*)(void *),void *);
//用于创建线程
//第一个参数为线程 id 或句柄（用于停止线程等）
//第二个参数代表各种属性，一般为空（代表标准默认属性）
//第三个参数为创建出的线程执行工作所要调用的函数
//第四个参数为第三个参数调用函数所需传递的参数
int pthread_join(pthread_t *, void **value_ptr);
//除非目标线程已经结束，否则挂起调用线程直至目标线程结束
//第一个参数为目标线程的线程 id 或句柄
//第二个参数允许目标线程退出时返回信息给调用线程，一般为空
void pthread_exit(void *value_ptr);
//通过 value_ptr 返回结果给调用者
int pthread_cancel(pthread_t thread);
//取消线程 thread 执行
```

简单示例：

```c
#include <pthread.h>
#include <stdio.h>

void* worker(void* arg) {
    printf("Hello from thread!\n");
    return NULL;
}

int main() {
```

```
    pthread_t thread;
    pthread_create(&thread, NULL, worker, NULL);
    pthread_join(thread, NULL);
    return 0;
}
```

2.2.2 OpenMP

全称： Open Multi-Processing

平台： 跨平台（主要在 Unix/Linux 和 Windows 上）

语言： C/C++ 和 Fortran

特性：
- 是一种**高层次并行编程**模型，通过**编译器指令（pragma）**实现并行化；
- 适合用于多核 CPU 上对循环、任务等结构化代码的自动并行化；
- 编码**简洁**，不需要显式管理线程或同步。

基础 API：

```
#pragma omp <directive> [clause[[,] clause] ...]
// 基础语法语句如上，将此格式的预编译指令用在一个语句块之前
// directive 可以是以下内容：
parallel;    // 创建线程，接下来的代码块是一个并行区域
single;      // 之后的代码块只会由一个线程（未必是主线程）执行
master;      // 之后的代码块只会由主线程执行
for;         // 之后的 for 循环将被并行化由多个线程划分执行，循环变量必须是整型
barrier;     // 所有线程在此同步
critical;    // 之后的代码块一次只有一个线程可以执行（临界区）
// clause 可以是以下内容：
if(scalar-expression);  // 当跟在 parallel 指令之后使用时表示是否并行化
private(list);          // 指定并行区域中的变量为每个线程独立的存储
shared(list);           // 指定并行区域中的变量为各个线程共享
default(shared|none);   // 未指定变量缺省的存储方式，shared 表示共享，none 表示必须为所有变量指定存
↪ 储方式
nowait;                 // 在离开下方的语句块时不进行线程同步
schedule(static|dynamic|guided[, chunk_size]);  // 用在 for 指令之后，指定循环任务的划分方法
// OpenMP 提供的一些其他库函数：
void omp_set_num_threads(int _Num_threads);
//设置之后一个并行区域的线程数，只能在串行代码区域使用
int omp_get_num_threads(void);
//获得当前线程数目
int omp_get_thread_num(void);
//返回当前线程 id.id 从 1 开始顺序编号，主线程 id 是 0
double omp_get_wtime(void);
// 获得当前墙上时钟时间，每个线程都有自己的时间
```

简单示例：

```
#include <stdio.h>
#include <omp.h>

```

```
int main() {
    #pragma omp parallel
    {
        printf("Hello from thread %d\n", omp_get_thread_num());
    }
    return 0;
}
```

2.2.3 PCFG 猜测循环的并行化

PCFG 算法中生成猜测部分的串行算法通过循环来实现：

```
// ... ... ... ...
// Multi-thread TODO:
// 这个 for 循环就是你需要进行并行化的主要部分了，特别是在多线程 &GPU 编程任务中
// 可以看到，这个循环本质上就是把模型中一个 segment 的所有 value，赋值到 PT 中，形成一系列新的猜测
// 这个过程是可以高度并行化的
for (int i = 0; i < pt.max_indices[0]; i += 1)
{
    string guess = a->ordered_values[i];
    guesses.emplace_back(guess);
    total_guesses += 1;
}
// ... ... ... ...
// Multi-thread TODO:
// 这个 for 循环就是你需要进行并行化的主要部分了，特别是在多线程 &GPU 编程任务中
// 可以看到，这个循环本质上就是把模型中一个 segment 的所有 value，赋值到 PT 中，形成一系列新的猜测
// 这个过程是可以高度并行化的
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;
}
// ... ... ... ...
```

而本此实验的主要目标是，通过 pthread/OpenMP 将以上两个循环（实际上原理是相同的）进行并行化，同时改变头文件、main 函数等文件，使不同线程的返回结果能够存起来，并且按照 main 函数里面的方式进行哈希和清理。

3 基础实验

基础要求： 将猜测生成过程用 pthread 和 OpenMP 进行多线程并行化。具体要求如下：

- 保证并行化后的猜测过程“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”，但总体上口令的生成结果基本不会有差别。检查正确性的方法是：在主函数中新增给定代码检查口令破解数，需要保证并行算法破解数量在数量级上和串行算法保持一致。
- 不要求真正实现相对串行算法的加速，但是仍需要给出实验数据，并且分析为什么没有能够实现加速。

3.1 并行度选择

服务器提供的最大线程数是 8，默认从最大线程数 $PARA\_NUM = 8$ 为实验起点。

3.2 pthread 的口令猜测并行化实现

3.2.1 并行化代码实现（无线程池）

综合考虑 pthread 的各种实现风格，发现使用“静态线程分配 + 线程互斥锁/信号量同步”的模式是相对优良的选择。一方面，静态线程不像动态线程每次执行任务都需要重新创建线程，减省巨大量的额外时间消耗；另一方面，线程互斥锁或信号量能够实现有一定条件的阻塞，保证程序同步。然而，本无线程池实现的静态线程分配方法并不是真正完全的静态线程分配，后续会进行分析。

接下来逐步实现无线程池的 pthread 并行化方法。

首先，在 [PCFG.h] 中定义线程结构体 `threadParam_t`，此结构体承载单个线程的所有任务信息，在创建线程和等待线程完成时都会被调用：

```
// 线程数据结构
typedef struct {
    int t_id;                   // 线程 id
    segment* a;                 // segment 指针，取表中值生成猜测需调用
    int max_index;              // 该 segment 的取值总数，即表值最大索引
    vector<string>* guesses;    // 指向所有生成的猜测
    int* total_guesses;         // 指向生成猜测总量
    string init_guess;          // 猜测前缀，多 segment 的猜测需调用
} threadParam_t;
```

而后，实现线程函数。将不同的数据或任务分配给不同线程进行完全相同或相似的处理，线程函数的作用就在于定义这一能够统一化的处理过程。在 [guessing.cpp] 中实现线程函数 `threadFunc` 的伪代码如下：

Algorithm threadFunc

```
function THREADFUNC(param)
    /* 从线程结构体中获取参数 */
    p ← CAST(param as threadParam_t*)
    t_id ← p.t_id
    a ← p.a
    max_index ← p.max_index
    guess ← p.init_guess

    /* 执行任务前准备 */
    my_guesses ← empty string vector                ▷ 线程局部猜测结果列表
    my_count ← 0                                    ▷ 线程局部猜测生成数目计数器
    chunk_size ← CEIL(max_index / NUM_THREADS)      ▷ 将任务平均划分为线程数目份
    if chunk_size = 0 then
        chunk_size ← 1
    end if
    start ← t_id × chunk_size                       ▷ 此线程生成猜测的起点
```

```
14:   end ← MIN(max_index, start + chunk_size)                ▷ 此线程生成猜测的终点
      /* 执行分配的任务 */
15:   for my_index = start to end - 1 do
16:     temp ← guess || a.ordered_values[my_index]            ▷ 拼接生成新口令
17:     APPEND(my_guesses, temp)                              ▷ 生成的猜测结果添加到局部猜测结果列表
18:     my_count ← my_count + 1
19:   end for
      /* 临界区：将每个线程的局部猜测写入共享猜测列表 */
20:   LOCK(mutex_guess)                                       ▷ 上锁保证线程安全
21:   INSERTRANGE(p.guesses, my_guesses)                      ▷ 合并线程的结果
22:   *(p.total_guesses) += my_count
23:   UNLOCK(mutex_guess)
24:   PTHREAD_EXIT(NULL)
25: end function
```

对于以上代码逻辑，作几点说明：

- **threadParam_t.guesses 和 threadParam_t.total_guesses：**线程结构体中的 `guesses` 和 `total_guesses` 分别是指向整个队列类中的共享猜测生产结果向量的指针，和指向共享猜测结果数目的指针。它们本身并不是指针，但**必须以指针形式传入**结构体，只有这样才能保证所有线程都能够修改**同一份共享的结果**，且访问操作相互不依赖，从而实现真正的多线程并行。
- **my_guesses 和 my_count：**现在线程内局部保存结果和相关参数，最后在临界区线程安全地更新到共享结果中，这么做是为了**减少锁竞争**提高效率，因为如果不设置局部变量而是每次生成后立即更新到共享结果中，虽然能够减少一些空间开销，但同时会产生过量锁竞争，这对效率的负面影响是远大于分配局部变量的影响的。
- **chunk_size：**这一参量代表单个线程生成猜测的数目，实际赋值式是 `chunk_size = (max_index + NUM_THREADS - 1) / NUM_THREADS`，其中 `(+ NUM_THREADS - 1)` 是为了达到**向上取整**的目的，因而在后面设置猜测生成终点时，需要取总猜测数和起点加上分配猜测数的结果这两者中的**较小值**，否则执行任务时可能会超出该 segment 可取值的索引范围。在后续有线程池的 pthread 实现和 OpenMP 的实现中同样有这两点设计。

完成线程结构体和配套的线程功能函数后，在原来串行算法的循环部分创建线程并调用线程等，用 pthread 进行并行化替换。在 [guessing.cpp] 中实现无线程池的 pthread 多线程并行版猜测生成函数 `PriorityQueue::PthreadGenerate` **修改部分**伪代码如下所示：

Algorithm PriorityQueue::PthreadGenerate

```
1: function PTHREADGENERATE(PT)
2:   if PT.content.size() == 1 then        ▷ 只包含一个 segment，可直接并行生成所有猜测值
       /* 此前代码不变 */
3:     Initialize thread array handles[NUM_THREADS] and params[NUM_THREADS]
   ▷ 初始化句柄和参数容器
```

```
    Initialize mutex mutex_guess                                   ▷ 初始化互斥锁
  /* 循环地创建线程 */
    for t_id = 0 to NUM_THREADS - 1 do
      Initialize params[t_id] with:                                ▷ 传入线程参数
        t_id, a, PT.max_indices[0], reference to guesses, total_guesses, and empty init_guess
      Create thread with threadFunc and params[t_id]               ▷ 创建线程
    end for
  /* 等待所有线程完成任务 */
    for t_id = 0 to NUM_THREADS - 1 do
      Join thread handles[t_id]
    end for
    Destroy mutex mutex_guess
  else                          ▷ 多 segment 情况，仅最后一个 segment 并行生成，前缀提前拼接
  /* 此间代码不变 */
    Initialize thread array handles[NUM_THREADS] and params[NUM_THREADS]   ▷ 初始化句柄和参数容器
    Initialize mutex mutex_guess                                   ▷ 初始化互斥锁
  /* 循环地创建线程 */
    for t_id = 0 to NUM_THREADS - 1 do
      Initialize params[t_id] with:                                ▷ 传入线程参数
        t_id, a, PT.max_indices[0], reference to guesses, total_guesses, and precomputed guess
                                                                   ▷ 多 segment 的唯一区别在于有 guess 前缀
      Create thread with threadFunc and params[t_id]               ▷ 创建线程
    end for
  /* 等待所有线程完成任务 */
    for t_id = 0 to NUM_THREADS - 1 do
      Join thread handles[t_id]
    end for
    Destroy mutex mutex_guess
  end if
end function
```

无线程池的 pthread 方法实现完整代码见 GitHub 仓库：[guessing.cpp]

3.2.2 并行化代码实现（有线程池）

无线程池的 pthread 方法编写完成后在服务器上进行测试，发现运行速度很慢（一般 > 0.7s，而串行一般仅需 < 0.6s，数据佐证见 3.5 节），并未实现加速。经过分析后，发现主要瓶颈在于，静态线程的设计理念并没有贯彻完全——无线程池的设计下，对于每个 PT，都要重新创建线程来进行猜测并

行任务，而整个猜测过程可能有成千上万个 PT 需要进行处理，每次都要重新创建线程，这带来不小的额外时间开销。

对此，考虑使用**线程池**来完全实现线程的静态化。线程池的大致设计思路是：

- 在整个程序开始时就创建线程池，并创建一定数量的线程供使用；
- 当有需要多线程处理的任务出现时，将任务划分为若干份，构成一个任务队列，线程池中的线程从中依次拾取，只要当前任务完成就从队列中再取一个，线程间互不影响；
- 当前总任务完成而程序未结束时，线程池中的线程切换到挂起状态，等待下一次的任务分配，而不是直接销毁再后续创建；
- 循环往复；
- 整个程序结束时销毁线程池，包括其中的所有线程。

从线程池的设计逻辑就可以看出，整个程序运行期间所有线程**只需创建一次**，完全地贯彻了静态线程理念。

有线程池的 pthread 方法实现，还需要且必须用到**条件变量**来实现阻塞、控制同步。具体设计了两个关键的条件变量，一个控制无任务和有任务状态之间（或线程挂起和激活状态之间）的转换，一个标识当前所有分配的任务均已完成（任务队列空）的状态。在实际方法函数中，调用这两种条件变量实现唤醒线程、等待全部任务完成等一系列操作。

接下来逐步实现有线程池的 pthread 并行化方法。

首先，在 [PCFG.h] 中定义线程任务结构体 `threadTask_t`，总任务被划分成若干分任务分配给线程，此结构体就承载了一个分任务的所有信息。在划分任务时和静态线程从任务队列中获取任务的过程中都会使用到：

```
// 线程任务结构
typedef struct {
    int t_start, t_end;          // 该任务生成猜测的起点和终点
    string t_preguess;           // 猜测前缀
    vector<string>* shared_values;  // 指向 segment 的所有取值的向量组
    string* shared_guesses;      // 指向总任务的猜测结果，所有线程共享一个字符串组
    bool t_active;               // 标识此任务是否是活跃状态（处于任务队列/正在被线程处理）
} threadTask_t;
```

而后，定义线程池类 `ThreadPool`。线程池要同时管理任务、线程、条件变量、线程互斥锁。首先在 [PCFG.h] 中声明类：

```
// 线程池类
class ThreadPool
{
    private:
    // 线程列表
    vector<pthread_t> threads;
    // 任务队列
    queue<threadTask_t> tasks;
    // 活跃任务数（包括已分配给线程正在运行的，和已入列但未分配的）
    int active_tasks;
```

```

    // 条件变量：有新任务或需要开始其他线程操作
    pthread_cond_t active_cond;
    // 条件变量：所有任务完成
    pthread_cond_t alldone_cond;

    // 队列互斥锁
    pthread_mutex_t task_mutex;
    // 线程安全的停止符号
    std::atomic<bool> terminate;

    public:
    // 构造函数
    ThreadPool();
    // 析构函数
    ~ThreadPool();

    // 添加任务
    void taskAppend(threadTask_t task);
    // 等待完成
    void waitAll();

private:
    // 线程任务函数
    static void* threadFunction(void* pool);
};
```

接下来在 [guessing.cpp] 中依次实现 ThreadPool 的成员函数。

实现 ThreadPool 构造函数，初始化线程列表、任务队列、条件变量和锁，然后创建静态线程：

```
ThreadPool::ThreadPool() : threads(), tasks(), active_tasks(0), terminate(false) {
    // 初始化锁和条件变量
    pthread_cond_init(&active_cond, NULL);
    pthread_cond_init(&alldone_cond, NULL);
    pthread_mutex_init(&task_mutex, NULL);
    threads.resize(NUM_THREADS);
    // 创建静态线程
    for (int t_id = 0; t_id < NUM_THREADS; t_id++) {
        pthread_create(&threads[t_id], NULL, &ThreadPool::threadFunction, this);
    }
}
```

实现 ~ThreadPool 析构函数，让所有线程处理完剩余任务，然后销毁锁和条件变量：

```
ThreadPool::~ThreadPool() {
    // 设置终止标志
    terminate = true;
    // 唤醒所有等待线程
    pthread_cond_broadcast(&active_cond);
```

```
    // 等待所有线程结束
    for (auto& thread : threads) {
        pthread_join(thread, NULL);
    }
    // 销毁锁和条件变量
    pthread_mutex_destroy(&task_mutex);
    pthread_cond_destroy(&active_cond);
    pthread_cond_destroy(&alldone_cond);
}
```

实现 taskAppend 任务入列函数，注意该过程需要上锁，因为处理的是所有线程共享的资源：

```
void ThreadPool::taskAppend(threadTask_t task) {
    pthread_mutex_lock(&task_mutex);
    tasks.push(task);
    active_tasks++;
    pthread_mutex_unlock(&task_mutex);
    pthread_cond_signal(&active_cond);
}
```

实现 waitAll 等待任务完成函数，同样需要上锁：

```
void ThreadPool::waitAll() {
    pthread_mutex_lock(&task_mutex);
    // 只要还有活跃任务，等待其全部完成
    while (active_tasks > 0 || !tasks.empty()) {
        pthread_cond_wait(&alldone_cond, &task_mutex);
    }
    pthread_mutex_unlock(&task_mutex);
}
```

实现 threadFunction 线程功能函数，这是**核心步骤**，对应无线程池实现中的 threadFunc 函数，对分任务的所有处理逻辑都包含在其中。伪代码如下：

Algorithm ThreadPool::threadFunction

```
function THREADFUNCTION(pool)
    t_pool ← static_cast<ThreadPool*>(pool)
    while true do                                   ▷ 循环地获取和处理分任务
        Initialize t_task with t_active = false     ▷ 初始没有获取任务，设置不活跃状态

    /* 尝试获取任务，进入临界区 */
        Lock t_pool.task_mutex
        while t_pool.tasks is empty and !t_pool.terminate do
            Wait on t_pool.active_cond with t_pool.task_mutex
        end while                                   ▷ 线程挂起状态：线程池仍存在，等待下发新任务
        if t_pool.tasks is empty and t_pool.terminate then
            Unlock t_pool.task_mutex
            return NULL
```

```
12:       end if                                          ▷ 退出：线程池关闭且无任务
13:       if !t_pool.tasks.empty() then
14:           t_task ← t_pool.tasks.front()
15:           t_pool.tasks.pop()
16:       end if                                  ▷ 活跃状态：从非空的任务队列取出首个任务
17:       Unlock t_pool.task_mutex

      /* 若成功获取任务，处理该任务 */
18:       if t_task.t_active then
19:           for i = t_task.t_start to t_task.t_end − 1 do
20:               guess ← t_task.t_preguess + (*t_task.shared_values)[i]
21:               t_task.shared_guesses[i] ← guess
22:           end for

      /* 任务完成，更新相关状态 */
23:           Lock t_pool.task_mutex
24:           t_pool.active_tasks−                                  ▷ 任务数减量
25:           if t_pool.active_tasks == 0 and t_pool.tasks.empty() then
26:               Signal t_pool.alldone_cond   ▷ 若全部分任务均已完成，通知等待函数已经完成
27:           end if
28:           Unlock t_pool.task_mutex
29:       end if
30:   end while
31: end function
```

`ThreadPool` 的所有成员函数全部实现之后，就可以在 [guessing.cpp] 中实现有线程池的 pthread 多线程并行版猜测生成函数 `PriorityQueue::PthreadPoolGenerate` 并调用 `ThreadPool` 中的相关方法。其伪代码如下所示：

Algorithm PriorityQueue::PthreadPoolGenerate

```
1: function PthreadPoolGenerate(pt)
2:     if pt.content.size() = 1 then
       /* 此前代码不变 */
3:         batch_size ← pt.max_indices[0]                  ▷ 此任务需要生成的总猜测数

       /* 如果总数小于一定值，直接选用串行方法处理 */
4:         if batch_size < 100000 then
5:             for i = 0 to batch_size-1 do
6:                 guesses.push_back(a.ordered_values[i])
7:             end for
8:             total_guesses += batch_size
9:             return
10:        end if

11:        chunk_size ← POOL_CHUNK_SIZE                    ▷ 获取分任务生成猜测数
```

```
        chunk_num ← ⌈ batch_size / chunk_size ⌉          ▷ 计算需要的分任务数，上取整
        Reserve space in guesses for batch_size
        shared_guesses ← new string array of size batch_size   ▷ 预留共享猜测结果向量空间

      /* 依次创建任务并递交至任务队列 */
        for id = 0 to chunk_num - 1 do
            start ← id× chunk_size
            end ← min(batch_size, start+ chunk_size)
            Construct threadTask_t task with range [start, end) and pass a.ordered_values
    and shared_guesses                                          ▷ 创建任务
            thread_pool.taskAppend(task)                        ▷ 递交任务
        end for

        thread_pool.waitAll()                                   ▷ 等待所有任务完成

        for i = 0 to batch_size-1 do
            guesses.push_back(shared_guesses[i])                ▷ 局部结果合并到共享结果中
        end for
        total_guesses += batch_size                             ▷ 更新生成猜测的总数
        delete[] shared_guesses
    else
      /* 此间代码不变 */
        batch_size ← pt.max_indices[last]                       ▷ 此任务需要生成的总猜测数

      /* 如果总数小于一定值，直接选用串行方法处理 */
        if batch_size < 100000 then
            for i = 0 to batch_size-1 do
                guesses.push_back(prefix + a.ordered_values[i])  ▷ 与单 segment 不同，有前缀
            end for
            total_guesses += batch_size
            return
        end if

        chunk_size ← POOL_CHUNK_SIZE                            ▷ 获取分任务生成猜测数
        chunk_num ← ⌈ batch_size / chunk_size ⌉          ▷ 计算需要的分任务数，上取整
        Reserve space in guesses for batch_size
        shared_guesses ← new string array of size batch_size   ▷ 预留共享猜测结果向量空间

      /* 依次创建任务并递交至任务队列 */
        for id = 0 to chunk_num - 1 do
            start ← id× chunk_size
            end ← min(batch_size, start+ chunk_size)
            Construct threadTask_t task with guess, [start, end), a.ordered_values, shared_guesses
    ▷ 创建任务
```

```
44:            thread_pool.taskAppend(task)                    ▷ 递交任务
45:        end for
46:        thread_pool.waitAll()                               ▷ 等待所有任务完成
47:        for i = 0 to batch_size-1 do
48:            guesses.push_back(shared_guesses[i])            ▷ 局部结果合并到共享结果中
49:        end for
50:        total_guesses += batch_size                         ▷ 更新生成猜测的总数
51:        delete[] shared_guesses
52:    end if
53: end function
```

注意到，除了线程池的部署，相对于无线程池的 pthread 实现，有线程池的实现还有几处不同的设计，对此作相应解释：

- **选用并行方法的阈值：**在调用 pthread 方法的时候，增加了一个对于总生成数的阈值要求，当 < 100000 的时候直接选用串行方法。这么做的原因是为了避免任务量过小时每个任务**进行并行化本身所需的开销**（如创建任务、线程取任务、临界区更新变量等）占据主要消耗，如果这样并行化提高效率的目的就不能够达到了。
- **预留空间给结果向量：** `PthreadPoolGenerate` 中有为共享结果向量预留空间的操作，在实际代码中写作 `guesses.reserve(guesses.size() + batch_size)`，即提前为其分配此任务**要求生成的猜测数**的新空间，达到预留扩容效果。这么做的目的是防止在处理过程中对共享结果向量进行频繁扩容而产生额外开销。
- **&a->ordered_values：**之前设计的 pthread 方法中直接给线程结构体传入 `a`，也就是指向最后一个 segment 的指针，而新设计的 pthread 方法中传给线程任务的是 `a->ordered_values`，即 segment 的取值向量组。这样当线程访问取值时，不用再经过 segment 指针，也能一定程度上减省额外开销。

以上就完成了有线程池的 pthread 方法的核心代码部分。此外，还需要两个简单的辅助函数接口，使得主函数中能够进行线程池的创建和删除：

```
1  void initThreadPool() {
2      if (!thread_pool) {
3          thread_pool = new ThreadPool();
4      }
5  }
6
7  void deleteThreadPool() {
8      if (thread_pool) {
9          delete thread_pool;
10         thread_pool = NULL;
11     }
12 }
```

有线程池的 pthread 方法实现完整代码见 GitHub 仓库：[guessing.cpp]

3.3 OpenMP 的口令猜测并行化实现

OpenMp 相对于 pthread 来说拥有更加高层的 API，编写起来十分简单，只要按照原逻辑，对相应部分进行并行化同时保证必要处的线程安全即可。伪代码：

Algorithm PriorityQueue::OpenMPGenerate

```
1: function OpenMPGenerate(PT pt)
2:     if pt.content.size() = 1 then
       /* 此前代码不变 */
3:         #pragma omp parallel
4:             Declare thread_guesses as local list of strings ▷ 给每个线程分配局部向量存储其生成的猜测
5:             #pragma omp for nowait
6:         for i = 0 to pt.max_indices[0] - 1 do
7:                 guess ← a.ordered_values[i] ▷ 生成猜测
8:                 thread_guesses.push_back(guess) ▷ 存储到局部向量中
9:         end for
10:            #pragma omp critical
11:                guesses.insert(guesses.end(), thread_guesses.begin(), thread_guesses.end()) ▷ 局部结果插入到共享/全局结果中
12:                total_guesses += thread_guesses.size()
13:    else
       /* 此间代码不变 */
14:        #pragma omp parallel
15:            Declare thread_guesses as local list of strings ▷ 给每个线程分配局部向量存储其生成的猜测
16:            #pragma omp for nowait
17:        for i = 0 to pt.max_indices[pt.content.size() - 1] - 1 do
18:                temp ← guess + a.ordered_values[i] ▷ 生成猜测
19:                thread_guesses.push_back(temp) ▷ 存储到局部向量中
20:        end for
21:            #pragma omp critical
22:                guesses.insert(guesses.end(), thread_guesses.begin(), thread_guesses.end()) ▷ 局部结果插入到共享/全局结果中
23:                total_guesses += thread_guesses.size()
24:    end if
25: end function
```

其中 `pragma omp parallel` 表示下面的部分进行 OpenMP 并行化；`pragma omp for nowait` 表示下面语句块各自执行完，无需线程同步；`pragma omp critical` 表示下面语句块是临界区，一次只有一个线程执行。给每个线程分配了局部变量来减少线程竞争。

OpenMP 方法实现完整代码见 GitHub 仓库：[guessing.cpp]

3.4 正确性检验

根据实验新提供的检测代码，在 [main.cpp] 中添加相应代码段进行**口令破解数**的监测：

```
// ... ... ... ...
//加载一些测试数据
unordered_set<std::string> test_set;
ifstream test_data("/guessdata/Rockyou-singleLined-full.txt");
int test_count=0;
string pw;
while(test_data>>pw)
{
    test_count+=1;
    test_set.insert(pw);
    if (test_count>=1000000)
    {
        break;
    }
}
int cracked=0;
// ... ... ... ...
        cout<<"Cracked:"<< cracked<<endl;   // 输出破解数
// ... ... ... ...
// 实验默认使用 4 路并行的 SIMD 进行 MD5
// 则在此方法中添加破解计数的逻辑
    // <===== SIMD *4 =====>
        bit32 state[4 * 4];
        for (int i = 0; i < q.total_guesses; i+=4) {
            string pw[4] = {"", "", "", ""};
            for (int i1 = 0; i1 < 4 && (i + i1) < q.total_guesses; ++i1) {
                pw[i1] = q.guesses[i + i1];
                if (test_set.find(pw[i1]) != test_set.end()) {      // 新增
                    cracked+=1;                                     // 新增
                }                                                   // 新增
            }
            SIMDMD5Hash_4(pw, state);
        }
// ... ... ... ...
```

对比各个算法间“Cracked”参数的大小，就可以检验实现的并行方法是否保证了结果的**相对正确**。

多次提交测试，结果稳定依次如下所示：

```
Guess time:0.590564seconds
Hash time:5.74715seconds
Train time:27.6621seconds
Cracked:358217
```

图 3.2: 串行算法破解结果

串行算法：Cracked = 358217.

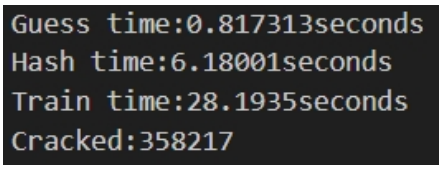

图 3.3: 无线程池 pthread 算法破解结果

无线程池 pthread 算法：Cracked = 358217.

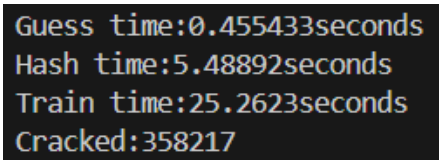

图 3.4: 有线程池 pthread 算法破解结果

有线程池 pthread 算法：Cracked = 358217.

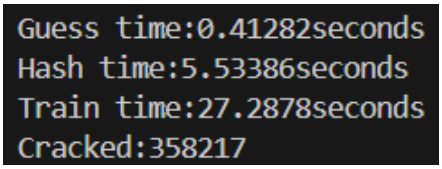

图 3.5: OpenMP 算法破解结果

OpenMP 算法：Cracked = 358217.

注：由于服务器不稳定等因素，猜测时间波动误差较大，理论上只要有较明显的加速比就足以证明是相应的并行算法的测试结果。

可以观察到，有/无线程池 pthread 算法和 OpenMP 算法的口令破解数目都和串行算法**完全一致**，证明实现的所有并行算法都是**正确**的。

3.5 加速效果测试

经过调试，发现 $POOL\_CHUNK\_SIZE$ 设置在 1000 和 10000 之间加速效果较好（比较过程略）。从而，在 $POOL\_CHUNK\_SIZE = 10000$，$NUM\_THREADS = 8$ 的条件下进行多次实验，分别测试有/无线程池 pthread 算法和 OpenMP 算法的加速效果。口令猜测生成用时结果如下表所示（多组实验，剔除最高和最低）：

no.	1	2	3	4	5	**avg.**
serial	0.582727	0.578463	0.569371	0.581616	0.581333	**0.57870**
pthread (w/o t-pool)	0.715748	0.724429	0.740107	0.711382	0.724787	**0.72329**
pthread (w/ t-pool)	0.453436	0.449738	0.4287	0.435065	0.45188	**0.44376**
OpenMP	0.434318	0.423028	0.422429	0.440298	0.4533	**0.43467**

表 1: 串行和各并行算法下的猜测生成用时（单位：秒）

根据以上数据依次求得各并行算法在猜测生成上的**加速比**：

$$\text{Speedup(pthread(-))} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{pthread (w/o t-pool)})} = \frac{0.57870}{0.72329} \approx 0.80 < 1$$

$$\text{Speedup(pthread(+))} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{pthread (w/ t-pool)})} = \frac{0.57870}{0.44376} \approx 1.30 > 1$$

$$\text{Speedup(OpenMP)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{SIMD*4})} = \frac{0.57870}{0.43467} \approx 1.33 > 1$$

三种并行算法中，无线程池的 pthread 算法不但没有实现加速，反而还有**负加速**效果；而其余两种，分别是有线程池的 pthread 算法和 OpenMP 算法，均有较为明显的加速效果，这两种算法的实现是**成功**的。

无线程池的 pthread 算法并不能实现加速的原因，在前文已有所提及。其根本问题在于，每次处理一个用于生成猜测的 PT 原型时，都需要**重新创建线程**，而生成规定数目的猜测所需要的 PT 可能是成百上千的，这将产生巨大的额外开销仅仅用于线程本身的各种操作而非并行地执行实际任务，从而抵消了其多线程并行的加速效果。甚至，在此例中，负面影响超过了并行加速的正面影响，显现为**负优化**。相反地，有线程池的 pthread 算法在整个程序运行周期内只创建一次线程池、创建一次线程，一个任务完成后无需销毁线程并到接收下一个任务时再重新创建，而是将线程挂起，接收到新任务时再唤醒。无线程池的 pthread 算法是**伪静态线程**设计，有线程池的 pthread 算法才是**真静态线程**设计。

数据真实性证明见 GitHub 仓库：[speedup min data.md]

4 进阶实验

4.1 实现加速并对比加速效果

pthread 和 OpenMP 的加速在基础实验中均已实现，见 3.5 节.

对比 pthread 和 OpenMP 的加速效果：

$$\text{Speedup(pthread(+))} = \frac{0.57870}{0.44376} \approx 1.30$$

$$\text{Speedup(OpenMP)} = \frac{0.57870}{0.43467} \approx 1.33$$

OpenMP 的加速效果略优于 pthread。对于这一现象，分析可能原因有以下几点：

1. pthread 手动显式管理锁，而 OpenMP 无需，其自动避免频繁加锁、**减少锁竞争**，可以优化性能；
2. pthread 中各个线程需要**竞争地访问**一些变量（如活跃任务总数），而 OpenMP 是自动分配任务的，无需手动管理这些变量，可能实现了更优于 pthread 下手动实现的管理方式；
3. pthread 线程池由手动方式提交任务，而 OpenMP 由编译器控制，相比于 pthread 能更好地按线程核数划分任务、减少线程切换，在任务数量较多且执行时间短的情况下，OpenMP 的**线程调度效率**更胜一筹；
4. pthread 线程池仍要维护线程等待、唤醒、任务队列，而 OpenMP 算法下编译器始终视线程组为静态，这在任务粒度细小时（实际上对于本实验这个简单的循环来说是较为符合的）OpenMP 的**启动/销毁线程**操作更为轻量，成本接近零。

4.2 探究编译选项对于加速比的影响

前述实验都是以**-O2 最高优化**下编译运行的，接下来探究不同优化选项下分别对于 pthread 和 OpenMP 并行算法的加速效果有什么影响。

增加两大组实验，分别选用无优化和-O1 优化选项，仍是在 $POOL\_CHUNK\_SIZE = 10000$，$NUM\_THREADS = 8$ 的条件下，对 pthread 算法（只测试有线程池能加速的版本）、OpenMP 算法和串行算法分别进行多组实验并去除最低最高值，得到的结果如下表所示：

*type**no.*		1	2	3	4	5	**avg.**
$-O0$	serial	8.06626	7.97587	7.94925	8.00274	8.00876	**8.00057**
	pthread	7.83287	7.88268	7.90866	7.79561	7.86378	**7.85672**
	OpenMP	7.75266	7.80385	7.81049	7.80779	7.78919	**7.79280**
$-O1$	serial	0.583904	0.588692	0.615786	0.600929	0.590185	**0.59590**
	pthread	0.509655	0.521472	0.529954	0.528007	0.510589	**0.51994**
	OpenMP	0.470365	0.503564	0.499296	0.490452	0.48622	**0.48998**
$-O2$	serial	0.582727	0.578463	0.569371	0.581616	0.581333	**0.57870**
	pthread	0.453436	0.449738	0.4287	0.435065	0.45188	**0.44376**
	OpenMP	0.434318	0.423028	0.422429	0.440298	0.4533	**0.43467**

表 2: 不同优化选项下串行和并行猜测生成用时（单位：秒）

根据均值分别计算 pthread 和 OpenMP 算法下的加速比：

$$[-O0]: \quad \text{Speedup(pthread)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{pthread})} = \frac{8.00057}{7.85672} \approx 1.02$$

$$[-O0]: \quad \text{Speedup(OpenMP)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{OpenMP})} = \frac{8.00057}{7.79280} \approx 1.03$$

$$[-O1]: \quad \text{Speedup(pthread)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{pthread})} = \frac{0.59590}{0.51994} \approx 1.15$$

$$[-O1]: \quad \text{Speedup(OpenMP)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{OpenMP})} = \frac{0.59590}{0.48998} \approx 1.22$$

$$[-O2]: \quad \text{Speedup(pthread)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{pthread})} = \frac{0.57870}{0.44376} \approx 1.30$$

$$[-O2]: \quad \text{Speedup(OpenMP)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{OpenMP})} = \frac{0.57870}{0.43467} \approx 1.33$$

很显然，对于 pthread 和 OpenMP 算法，加速效果都有：$[-O2] > [-O1] > [-O0]$. ($-O0$ 就是无优化选项) 产生这一趋势的原理在 上次实验（Lab2-SIMD）报告 中已经全面分析。

在 pthread 和 OpenMP 两者之间，编译优化的影响还略有区别。按点进行陈述与分析：

- OpenMp 的优势始终保持，在任意编译选项下加速比都大于 pthread，或至少不小于，说明其由编译器控制的优化无论编译优化如何都能够发挥作用；

- OpenMP 在-O1 编译选项下相对于 pthread 的优势是明显大于在-O2 编译选项下的，根据 -O1 与 -O2 的优化差异（详细分析见 上次实验（Lab2-SIMD）报告）来分析，猜测是 OpenMP 算法本身需要更少的循环，或是编译器对于 OpenMP 算法恒定地进行循环展开、循环分块、指令调度、跨函数等方面的优化，这些优化对于 pthread 算法来说是只有使用 -O2 选项才能够实现的。

测试过程中观察数据发现 MD5 的优化并不受影响，加速比和上次单独进行实验测试的加速比是基本一致的，分析其原因是：

- pthread 或 OpenMP 算法使用的资源与 SIMD 使用的资源并不重合，SIMD 实现的加速主要来源于数据向量化后的并行处理，直接依赖编译器对于指令等的优化，而 pthread 或 OpenMP 实现的加速实际上来源于多核同时执行任务，并非编译器级指令重排或向量优化；
- pthread 算法中可选择的 SIMD 优化在此实现中并未使用到，未产生依赖。

要对比编译选项对于两次实验算法加速比影响的差异，直接对比上次实验获得的结果即可。

截取上次实验中**四路 SIMD** 并行化 MD5 在不同编译优化选项下的加速比测试结果：

$$[-O0]: \quad \text{Speedup(SIMD*4)} \approx 0.64$$

$$[-O1]: \quad \text{Speedup(SIMD*4)} \approx 1.67$$

$$[-O2]: \quad \text{Speedup(SIMD*4)} \approx 1.77$$

对比发现，SIMD 并行化的 MD5 在 -O1 和 -O2 编译选项下能够实现明显的加速，但在无优化（-O0）的选项下实现负加速；而 pthread 和 OpenMP 算法并行化的猜测生成过程在所有编译选项下都能实现加速（考虑到数据误差，在 -O0 选项下可能没有加速效果，但绝不会实现负加速）。这再次说明，pthread 和 OpenMP 的加速来源于多核硬件同时执行任务，而 SIMD 的加速来源于编译时的向量化等而非硬件，所以才会在不选择编译优化时失去加速效果。

数据真实性证明见 GitHub 仓库：[opt min data.md]

4.3 实现猜测生成和 MD5 哈希的同时加速

上一小节已经分析到，在此实现中 pthread/OpenMP 对于猜测生成的加速和 SIMD 对于 MD5 哈希的加速并不会相互影响，所以原本就已经实现了两者的同时加速。为了验证，采用四路 SIMD 优化 MD5 分别进行 pthread（有线程池）和 OpenMP 测试，在猜测生成过程确实能够加速的前提下，记录 MD5 过程的时间，与上次实验得到的数据进行对比，说明加速效果正常：

no.	1	2	3	4	5	**avg.**
pthread	1.67345	1.68174	1.68263	1.68068	1.67655	**1.6790**
OpenMP	1.67797	1.68026	1.68119	1.6786	1.67834	**1.6793**

表 3: pthread 和 OpenMP 算法下的 MD5 用时（单位：秒）

从 上次实验（Lab2-SIMD）报告 中获取 SIMD 四路并行 MD5 的平均用时是 1.6726s，计算得 pthread 和 OpenMP 下 MD5 的平均用时与其之差未超过 ±0.01s，参照串行算法的平均用时 2.9658s，差值范围是可以接受的，仍能够说明 pthread 和 OpenMP 在加速猜测生成过程的同时，SIMD 能正常加速 MD5 过程。

数据真实性证明见 GitHub 仓库：[MD5 min data.md]

5 总结思考

相对于上次 MD5 实验，此次实验的工程难度提高了。从原理上来看，本次实验需要实现并行的逻辑本身很简单明了，使用的工具也提供了很多现成的 API，但是实际操作时，会出现各种各样额外开销导致并行加速效果被掩盖的情况，需要按步寻找瓶颈、设计解决方案、编写代码实现。而每一个优化方案提高的效率可能只是细微的，所以以上步骤要重复进行很多次，才能得到真正能够加速循环过程的并行算法。对于 pthread 算法的设计和实现，这种体会尤为深刻，而相比之下 OpenMP 的实现就简单了好几倍，因为其提供的 API 接口是高层的，无需实现各种底层的线程操作，很轻松就能够实现并行加速。

在实验过程中也引出一个问题：

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充分利用的设备资源。并行化适合处理什么样的问题？什么时候并行化更合适，什么时候串行更合适？

通过自己的实验体会，加以一定的资料查询，对此问题作以下几点回答：

1. 并行算法确实需要占用相对于串行算法更多的软硬件资源。同时，管理这些资源必要的一些工作（线程启动、切换等）带来的额外开销也是非常大的。所以，如果想要并行化提高效率，就必须让其加速**计算的任务体量足够大**，来放大加速效果，从而填补**资源消耗和管理开销大**的缺陷。

2. 并行化适合处理的问题大致要满足两个要求：(1) **存在可并行计算的部分**，这是前提，如果任务有很强的前后依赖关系等而无法进行并行计算（伪并行，实现后其实仍是串行），必然无法使用并行化的算法处理；(2) **任务计算量足够大**，只有这样，其加速效果得以足够放大化，才能填补资源消耗和管理开销大的缺陷。

3. 并行化处理问题一般分为两类：**数据并行问题**和**任务并行问题**。数据并行问题（*e.g.* 矩阵运算、视频处理、密码哈希）是将**数据划分**为独立块，不同线程对不同数据作相同或相似处理；任务并行问题（*e.g.* Web 服务器处理多个客户端请求、图像渲染中多个光线的独立追踪）是将整个**任务划分**为独立快，不同线程对相同数据作不同任务处理。其中任务并行问题必须满足各个划分后的独立任务之间不存在或几乎**不存在依赖**，否则无法实现并行或并行效果极差。

4. 综上所述，并行算法适合用于**任务/数据量较大**，且分任务之间**无或几乎无依赖**的任务；反而言之，串行算法就适合使用于**任务/数据量较小**，且分任务之间有**强依赖性**（递归、链式计算等）的任务。当然，在实际工程中，还需考虑资源问题，代码工程难度问题，以及整体性能要求。资源富集、代码工程难度较低、性能要求较高的条件下倾向考虑并行，反之倾向考虑串行。

5. 具体在本次实验的算法设计和代码实现中，服务器提供的并行资源是足够的（多核），那么就既要尽可能多地使用并行算法优化，也要同时注意对于**粒度小**（分任务工作量小）的任务，是否需要回归到串行算法。实验要求进行并行化的是固定的**循环体**，一定是可以并行化的，不用过多考虑任务的依赖性。然而任务粒度小是会实际发生的。具体来说，以一个传入的 PT 为基础生成猜测时，可能当前赋值的 segment 可选的值很少，那就不应当进行并行化，否则开启、调度线程产生的额外开销将远大于小型任务并行化带来的速度提升，导致负优化。所以在实际编程中，对于**猜测数小**于一定数目的情况，直接采用**串行**算法处理，这才达到了并行与串行的平衡，综合提高了猜测生成的效率。

本次实验所有代码均已上传至 GitHub：

{OpenEuler 服务器中的 NEON 实现}