南開大學
Nankai University

计算机学院

并行程序设计实验报告

期末报告

姓名：林盛森

学号：2312631

专业：计算机科学与技术

2025 年 7 月 6 日

目录

1 摘要

我们本学期针对 PCFG 口令猜测算法进行了全面的并行化优化，通过 SIMD、多线程、MPI 和 GPU 四种并行技术实现了算法不同环节的加速，此次报告基于以前的实验内容进行总和，并尝试创新不同的算法设计。在正文部分，我将先对之前的 SIMD、多线程、MPI 和 GPU 四次实验进行总和，然后在此基础上提出创新工作，并进行实验验证与性能分析。

2 引言

PCFG（Probabilistic Context-Free Grammar，概率上下文无关文法）口令猜测是一种基于语言建模原理的密码猜测攻击技术，它结合了统计学习和文法规则，用于高效生成可能的密码候选列表，优于传统暴力破解或字典攻击。其核心思想是：利用密码样本学习密码结构和模式的概率分布，构建一个概率文法模型，再据此生成猜测结果，按概率从高到低进行尝试。

在理解了实验指导书上的内容之后，我们可以归纳出其基本需要实现以下几个过程：

1. **分析训练集**：首先会对训练集进行分析，提取一些密码的结构并计算概率，比如 L8D3S3 这种，代表这个密码由 8 个字母 +3 个数字 +3 个符号组成，并且会计算出其在训练集中出现的概率。

2. **建立概率模型**：为每种模式生成一个文法规则，并为每种模式及其组成部分（字母、数字、符号）计算出现概率。也就是计算某个密码出现的概率，分为上一过程提到的模式概率和组件概率（也就是向其中填入真正的字符信息）。比如说 L8D1S1 的概率是 0.2，在 L8 中，12345678 出现的概率为 0.5，在 D1 中 a 出现的概率为 0.1，在 S1 中！出现的概率为 0.1，那么密码 12345678a！的总概率就为 0.5*0.2*0.1*0.1（条件概率）。

3. **生成口令候选**：根据概率文法生成新的口令，优先生成高概率组合。

4. **猜测过程**：生成的口令候选从高概率到低概率依次尝试，大幅减少猜测空间，提高成功率。

另外，PT 指的是一种拆分结构（如 L6S1），其中的每个部分叫 Segment。

3 统一问题建模

3.1 PCFG 模型核心流程

PCFG 口令猜测算法包含三个关键阶段：

- **训练阶段**：分析密码数据集构建概率模型
- **生成阶段**：基于概率模型产生候选口令
- **验证阶段**：计算口令哈希值进行匹配验证

```
init() → 构建初始结构 → 插入优先队列（q）
    ↓
while q not empty:
    PopNext()
        ↓
    Generate(pt)        → 输出猜测字符串
        ↓
    NewPTs()            → 延展结构
        ↓
```

```
10    CalProb() for each → 插入优先队列
```

3.2 串行算法性能瓶颈分析

3.2.1 计算瓶颈

MD5 哈希计算: 该串行算法的核心瓶颈首先体现在 MD5 哈希计算的串行处理上，原始代码采用三重循环结构逐字节处理口令字符串，其中非线性逻辑函数（FF/GG/HH/II）的连续调用形成了严格的数据依赖链，这就导致 CPU 流水线效率非常低下。特别是在处理长密码时，位操作无法充分利用现代处理器的指令级并行能力进行优化，尤其是在 ARM 架构下，由于其设计理念是精简指令集，很多位运算操作需要通过多条指令组合来得到，比如说在逻辑计算

$$\neg X \wedge Z$$

这种时，SSE(X86) 有专门的指令 _mm_andnot_si128，但是在 arm 中，需要使用两个指令，vmvnq_u32 按位取反，vandq_u32 按位取与，NEON(arm) 需通过组合指令实现相同功能，增加了指令流水线的压力，这就使得在 x86 平台上基于 SSE 的操作指令数更少，运行效率更高。

概率计算：与此同时，概率计算模块存在的浮点运算密集型特征也构成显著的瓶颈，在 CalProb() 函数中会进行频繁的浮点除法和连乘操作而缺乏向量化支持，每个概率模板 (PT) 需要完成与 segment 数量成正比的标量运算，性能十分低下。

3.2.2 内存瓶颈

数据局部性：在 Generate 函数中，算法需要频繁访问模型数据结构（如 m.letters 、m.digits 、m.symbols ），这些数据分散在内存的不同位置。每次处理新的 PT 时，都要重新查找对应的 segment 数据，导致 CPU 缓存频繁失效。特别是在字符串拼接过程中，需要反复访问 ordered_values 数组中的不连续元素，这种跳跃式的内存访问模式严重影响了缓存效率，使得大量时间浪费在等待内存数据加载上。

内存分配:在生成密码猜测的核心循环中,每次都要进行字符串拼接操作(如 guess + a->ordered_values[i])，这会触发新的内存分配和数据复制。同时，guesses 向量的动态扩容也是一个重要瓶颈，当向量容量不足时会触发整个数组的重新分配和数据迁移，这不仅消耗大量内存带宽，还会产生内存碎片。虽然代码中尝试使用 reserve() 来预分配空间，但在处理大规模数据时，频繁的字符串创建和销毁仍然会给内存管理系统带来巨大压力。

3.2.3 控制流瓶颈

优先级队列维护：在 PopNext 函数中，每次处理完一个 PT 后都需要将新生成的 PT 按概率顺序插入到优先级队列中。这个插入过程采用了线性搜索的方式，需要遍历整个队列来找到合适的插入位置。随着队列长度的增长，这种 O(n) 的插入操作会变得越来越慢，特别是当队列中有成千上万个 PT 时，每次插入都要进行大量的概率比较和迭代器操作。更糟糕的是，vector 的中间插入操作还会触发大量元素的移动，这进一步加剧了性能问题。整个算法的执行流程被这种串行的队列维护操作严重拖累。

分支预测：算法中存在大量难以预测的条件分支，这些分支的执行模式很难被 CPU 的分支预测器准确捕获。在 Generate 函数中，需要根据 segment 的类型（字母、数字、符号）进行不同的处理，

这些类型判断的结果往往是随机的，无法形成稳定的模式。同样，在优先级队列的插入过程中，概率比较的结果也是高度不可预测的，因为不同 PT 的概率分布可能差异很大。当 CPU 的分支预测器频繁预测错误时，就会导致指令流水线的清空和重新加载，造成大量的时钟周期浪费。这种分支预测失效在处理大规模数据时会显著降低 CPU 的执行效率。

3.3 并行化可行性论证

3.3.1 数据并行性

PCFG 算法的核心操作——基于概率模型的候选口令生成具有天然的**数据独立性**。在串行实现中，每个候选口令的生成过程仅依赖于概率模型的状态转移矩阵，与其他口令的生成过程完全独立，这种独立性为数据划分并行提供了基础条件，使得算法可以按照概率分布将整个密码空间划分为若干独立子集（若干子块），为后续 SIMD 向量化处理提供理论基础。特别值得注意的是，MD5 哈希计算作为验证阶段的核心操作，其固定长度的计算特征（128 位分组）也呈现出规整的数据并行模式。

3.3.2 任务并行性

算法中的概率语法树具有显著的**分支并行特性**。通过分析 PCFG 模型的结构特征发现：不同语法成分（字母段 L、数字段 D、特殊字符段 S）构成的子树在扩展时相互独立，并且同一层级内不同概率值的节点扩展任务也不存在执行顺序约束。这种树形结构的可分解性为多线程任务分配和 MPI 的分布式计算提供了潜在可能，可以将不同分支或层级的计算任务分配给多个计算单元同时执行。。

3.3.3 流水线并行

从算法流程的**阶段化特征**来看，PCFG 口令猜测可分解为三个主要阶段：训练阶段、生成阶段、验证阶段。这三个阶段呈现出明确的计算边界。训练阶段完成模型构建后，生成阶段可以立即开始工作；同样，生成阶段产生部分候选口令后，验证阶段即可并行执行。各阶段的计算负载特征不同，训练阶段侧重统计分析，生成阶段侧重概率采样，验证阶段侧重哈希计算。这种阶段化特征为构建多级并行流水线创造了条件，通过合理的任务调度并建立合理的缓冲区机制可以实现计算资源的充分利用，但需要注意阶段间的负载均衡问题。

4 多架构并行设计

4.1 SIMD 向量化优化

由于这部分处理的内容是与其他三个部分独立的，所以这里格外说明一下 MD5hash 的流程，主要包括以下 5 个部分：

1. 消息填充（调用 StringProcess 函数）
2. 分块处理
3. 初始化寄存器
4. 依次处理每个块
5. 合并结果

我们的并行算法思路就是：在原来一次处理一个字符串的基础上，我们将其改为一次性处理 4 个字符串的逻辑，具体实现是基于 **NEON128 位指令集**，其每条指令处理一个 128 位向量，也就是 4 个 32 位数据，从而解决 MD5hash 计算的瓶颈。在这一部分，我们主要需要并行化的就是 9 个逻辑函数，这里只粘贴出三个：

```
#define F_NEON(x , y , z ) \
    vorrq_u32 ( \
        vandq_u32 (( x) , (y) ) , \
        vandq_u32(vmvnq_u32(x) , ( z ) ) \
)
#define ROTATELEFT_NEON( vec , n) \
vorrq_u32 ( vshlq_n_u32 (( vec ) , (n) ) , vshrq_n_u32 (( vec ) , 32 - (n) ) )
#define FF_NEON(a , b , c , d , x , s , ac ) do { \
    uint32x4_t _f_term = F_NEON(( b) , ( c ) , (d) ) ; \
    uint32x4_t _f_add = vaddq_u32(_f_term , (x) ) ; \
    _f_add = vaddq_u32(_f_add , vdupq_n_u32( ac ) ) ; \
    ( a ) = vaddq_u32 (( a ) , _f_add) ; \
    ( a ) = ROTATELEFT_NEON(( a ) , ( s ) ) ; \
    ( a ) = vaddq_u32 (( a ) , (b) ) ; \
} while (0)
```

F_NEON 函数的原型是计算

$$(X \wedge Y) \vee (\neg X \wedge Z)$$

，并行版本即每次处理一个 128 位向量，对应的运算操作也应该适配于 128 位向量，vorrq_u32 对应的是或运算，vandq_u32 对应的是与运算；ROTATELEFT_NEON 的目的是实现 128 位向量的循环左移操作，即 4 个 32 位数的循环左移操作，只需将每个数据左移 n 位，右移 32-n 位的结果合并起来 (取或) 即可。vshlq_n_u32 是左移操作，第一个参数为操作数，第二个参数为移动位数，vshrq_n_u32 对应了右移操作；FF_NEON 是求取 hash 的一个步骤，用于对每个块进行操作，其参数 a、b、c、d 代表的是四个需要迭代的变量，每次函数处理只对第一个参数进行修改，x 代表的是传入四个字符串某个块的某 4 字节的 4 维向量，s，ac 都是一些常量，用于 hash 求取的过程，不再赘述。

然后比较重要的修改就是，我们需要修改 MD5hash 函数的 NEON 版本，给出代码如下：在 md5.cpp 中添加了 MD5Hash_NEON 函数另外还需要在 main.cpp 中修改调用时的逻辑，比较简单，这里不再赘述。

```
void MD5Hash_NEON( s t r i n g inputs [ 4 ] , bit32 ** state )
{
    Byte *paddedMessages [ 4 ] ;
    int *messageLengths = new int [ 4 ] ;
    for ( int i = 0; i < 4; i += 1) //############一 次 处 理 四 个 输 入
    {
    paddedMessages [ i ] = StringProcess ( inputs [ i ] , &messageLengths [ i ] ) ;
    assert(messageLengths[i] == messageLengths[0]) ;
    }
    int n_blocks = messageLengths [ 0 ] / 64;
    uint32x4_t state_temp [ 4 ] ;
    // bit32 * state= new bit32 [ 4 ] ;
    state_temp [0]=vdupq_n_u32(0 x67452301 ) ;
```

```
    state_temp [1]=vdupq_n_u32(0 xefcdab89 ) ;
    state_temp [2]=vdupq_n_u32(0 x98badcfe ) ;
    state_temp [3]=vdupq_n_u32(0 x10325476 ) ;
```

以上操作是一些预处理的步骤，传入参数为一个包含 4 个字符串的数组和一个二维数组 state 用于存储 hash 结果。我们需要一次性对这 4 个字符串进行操作。通过调用 StringProcess 函数，进行预处理，填充消息使其长度符合 MD5 算法要求（512 位的倍数）。assert 用于保证处理后的消息长度相同，另外我们定义了 n_blocks 这个变量用于记录一个字符串被分为了多少个块 (由于输入的字符串长度都差不多，一个块即可搞定，所以可以并行化，每个块 64 字节)。然后定义了 4 个 128 位的向量寄存器 state_temp，利用 vdupq_n_u32 指令，将初始常量值复制到每个向量的所有维度上，即我们在循环中需要使用的 a、b、c、d。

```
// 逐 block 地 更 新 state
for ( int i = 0; i <n_blocks ; i += 1) //每 个 块64字 节
{
    uint32x4_t x[16] ;
    for ( int i1 = 0; i1 < 16; i1+=4) //将1个 块 分 成16部 分 每 部 分4字 节
    {
        uint32_t data1 [4] ;
        uint32_t data2 [4] ;
        uint32_t data3 [4] ;
        uint32_t data4 [4] ;
        data1 [ 0 ] = ( paddedMessages [ 0 ] [ 4 * i1 + i * 64]) | //第0字 节
        (paddedMessages[0] [4 * i1 + 1 + i * 64] << 8) | //第1字 节
        (paddedMessages[0] [4 * i1 + 2 + i * 64] << 16) | //第2字 节
        (paddedMessages[0] [4 * i1 + 3 + i * 64] << 24) ; //第3字 节
        data1 [ 1 ] =paddedMessages [ 1 ] . . .
        data1 [ 2 ] =paddedMessages [ 2 ] . . .
        data1 [ 3 ] =paddedMessages [ 3 ] . . .
        x [ i1 ] = vld1q_u32 ( data1 ) ;
对data2、data3、data4数组作相同处理，并传入x[i1+1]、x[i1+2]、x[i1+3]之中，此处省略
    }
```

外层循环遍历每个块，内层循环将每个块分成 16 个部分，每个部分 4 字节。在循环内部，当 i1=0 时，data1 取出的就是四个块的第一部分，即前 4 个字节，并存储到 x[i1] 中，之后遍历依次取 4 个块的某一部分，即对应的 4 个字节。值得一提的是，我在这里使用了循环展开，使得一次循环可以处理每个块的前 16 个字节，这样提高了效率。另外，我需要对这里的逻辑说明一下，这里的操作是将 4 个连续的字节组合成一个 32 位整数，并且通过移位操作取出字节，或操作拼接字节，按照小端序组合字节，vld1q_u32 指令可以把数组元素存储在向量寄存器中。

```
uint32x4_t a=state_temp[0],b=state_temp[1],c=state_temp[2],d=state_temp[3];
auto start = system_clock::now();
/* Round 1 */
FF_NEON(a , b , c , d , x [ 0 ] , s11 , 0xd76aa478 ) ;
/* Round 2 */
/* Round 3 */
/* Round 4 */
. . . . . .
```

```
state_temp [ 0 ] = vaddq_u32(a,state_temp [ 0 ]) ;
state_temp [ 1 ] = vaddq_u32(b,state_temp [ 1 ]) ;
state_temp [ 2 ] = vaddq_u32(c,state_temp [ 2 ]) ;
state_temp [ 3 ] = vaddq_u32(d,state_temp [ 3 ]) ;
}
```

这里是算法的核心部分，首先将当前哈希状态复制到临时变量 a、b、c、d 中，然后去执行 MD5 的四轮计算，依次去调用我们实现的 FF_NEON, GG_NEON, HH_NEON, II_NEON 函数，每轮进行 16 次操作，另外需要将计算后的临时状态 a、b、c、d 与原始状态相加，更新 hash 状态，这里相加使用了 vaddq_u32 指令。这些操作通过向量寄存器同时对 4 个输入进行并行处理，大大提高了计算效率。

```
for (int k = 0; k < 4; ++k) {
    state [0][k] = vgetq_lane_u32(state_temp[k],0);//第一个输入的a/b/c/d
    state [1][k] = vgetq_lane_u32(state_temp[k],1);//第二个输入的a/b/c/d
    state [2][k] = vgetq_lane_u32(state_temp[k],2);//第三个输入的a/b/c/d
    state [3][k] = vgetq_lane_u32(state_temp[k],3);//第四个输入的a/b/c/d
}
for ( int i = 0; i < 4; i++){
    uint32_t value = state[i][0];
    state [i][0] = ((value & 0xff) << 24) |//将最低字节移到最高位
        ((value & 0xff00) << 8) | //将次低字节左移
        ((value & 0xff0000) >> 8) | //将次高字节右移
        ((value & 0xff000000) >> 24) ;//将最高字节移到最低位
    value = state[i][1];state[i][1] = .........
    value = state[i][2];state[i][2] = .........
    value = state[i][3];state[i][3] = .........
}
```

在这部分，首先从 4 个向量寄存器中取出结果，存储在二维数组 state 中，使用了 vgetq_lane_u32 指令。然后对得到的每个 hash 值进行字节序变换，从小端序转变为大端序。拿一个来解释逻辑，(value& 0xff) 将某四个字节与 0xff 相与，即保留最低位，再将其左移 24 位，即保留到了最高位上，另外三个逻辑也类似。

```
    for ( int i = 0; i < 4; i++) {
        delete [ ] paddedMessages [ i ] ;}
    delete [ ] messageLengths ;
}
```

最后记得释放之前动态分配的内存。

在学习完了本学期的并行算法之后，由于这一部分的主要瓶颈就在于计算，所以我觉得可以用 gpu 来实现 md5hash 并行加速，后续在报告中对这一点进行了实现。

4.2 多线程优化

我们多线程并行化的思路，就是分配多个线程，把这个任务量比较大的任务细化，分配给多个线程，每个线程处理一部分数据，并在最后总和各个线程完成的任务。在这次实验中我们做了 pthread 动态编程、pthread 静态线程池编程、openmp 编程等，而且还做了批量处理 pt 的逻辑，由于篇幅限制，在这里只汇报后两者。

4.2.1 pthread 静态线程池

首先这里定义了线程池任务结构体。每个任务会将 prefix + value[i] 拼接形成完整口令，这个结构体会被在线程池中被使用。

```
struct Task {// 线程池任务结构
    int start, end;
    string prefix;
    vector<string>* values;
    string* guesses_out;        // 可以为空，表示不使用共享结果数组
    int* guesses_count;  //结果数组
    int thread_id;              // 线程ID，用于标识写入哪个线程本地结果数组
    vector<string>* local_results; // 线程本地结果数组
};
```

在这里我们定义了一个线程池类，有一些变量结构，具体代表什么已在以下注释中给出。

```
class ThreadPool {// 线程池类
private:
    vector<pthread_t> workers;  //定义了线程容器
    queue<Task> tasks; //任务列表
    pthread_mutex_t queue_mutex;//互斥锁，用于保护 tasks 队列。
    pthread_cond_t condition;//任务可用的条件变量
    pthread_cond_t completion_cond;  // 任务完成条件变量
    int active_tasks;                // 表示当前正在执行的任务数
        配合stop和任务完成变量使用
    bool stop;//同于判断线程池是否处于终止状态。
```

定义了线程池的初始化，通过 pthread_create 创建工作线程，并将 this 指针传给静态线程函数 WorkerThread，供其访问成员变量。

定义了析构函数，stop = true 的标志会通知所有线程退出，在等待所有线程完成操作之后，会销毁锁等。

这里定义了一个函数 Enqueue，可以将任务压入队列，并唤醒一个等待中的线程处理任务。

```
    void Enqueue(Task task) {     // 添加任务到线程池
        pthread_mutex_lock(&queue_mutex);
        tasks.push(task);
        active_tasks++;  // 增加活跃任务计数
        pthread_mutex_unlock(&queue_mutex);
        pthread_cond_signal(&condition);}
```

这里的 WaitAll 函数会等待所有任务全部完成，所有任务完成后才返回。

```
    void WaitAll() {     // 等待所有任务完成
        pthread_mutex_lock(&queue_mutex);
        while (active_tasks > 0 || !tasks.empty()) {
            pthread_cond_wait(&completion_cond, &queue_mutex);
        }
        pthread_mutex_unlock(&queue_mutex);}
```

这里是进行口令猜测的核心部分，逻辑和之前相同，每个线程将其负责的 value 拼接上 prefix，写入结果数组 guesses_out 中，会在初始化时被调用于 pthread_create 中，也就会在 Generate_pthread_pool 函数中被调用，当接收到信号 stop=true 时，while 循环会停止，结束线程。

```
private:    // 工作线程函数
    static void* WorkerThread(void* arg) {
        ThreadPool* pool = static_cast<ThreadPool*>(arg);
        while (true) {
            Task task;bool got_task = false;
                pthread_mutex_lock(&pool->queue_mutex);
                if (pool->stop && pool->tasks.empty()) {
                    pthread_mutex_unlock(&pool->queue_mutex);
                    return NULL;}
                if (!pool->tasks.empty()) {
                    task = pool->tasks.front();
                    pool->tasks.pop();
                    got_task = true;
                } else if (!pool->stop) {
                    pthread_cond_wait(&pool->condition, &pool->queue_mutex);
                    pthread_mutex_unlock(&pool->queue_mutex);
                    continue;
                } else {
                    pthread_mutex_unlock(&pool->queue_mutex);
                    continue;}
                pthread_mutex_unlock(&pool->queue_mutex);
            if (got_task) { // 执行任务
                int local_count = 0;
                if (task.local_results != nullptr) {
                    for (int i = task.start; i < task.end; ++i) {
                        string guess = task.prefix + (*(task.values))[i];
                        task.local_results->push_back(std::move(guess));
                        local_count++;}
                } else if (task.guesses_out != nullptr) {
                    for (int i = task.start; i < task.end; ++i) {
                        string guess = task.prefix + (*(task.values))[i];
                        task.guesses_out[i] = std::move(guess);
                        local_count++;}}
                pthread_mutex_lock(&pool->queue_mutex);
                *(task.guesses_count) = local_count;
                pool->active_tasks--;
                if (pool->active_tasks == 0 && pool->tasks.empty()) {
                    pthread_cond_signal(&pool->completion_cond);}
                pthread_mutex_unlock(&pool->queue_mutex);}}
        return NULL;}        };
```

这个函数用于初始化全局线程池，会在 main.cpp 中被调用，以创建静态线程。

```
// 全局线程池，在程序启动时创建
ThreadPool* global_thread_pool = nullptr;
```

```
// 初始化全局线程池
void InitThreadPool(int num_threads) {
    if (global_thread_pool == nullptr) {
        global_thread_pool = new ThreadPool(num_threads);}}
```

同样，在主程序要结束时，会调用这个函数销毁线程。

```
// 清理全局线程池
void CleanupThreadPool() {
    if (global_thread_pool != nullptr) {
        delete global_thread_pool;
        global_thread_pool = nullptr;}}
```

Generate_pthread_pool 函数比较简单，我们添加了一个动态分配的逻辑，就是当任务量比较小的时候直接用串行算法执行，否则再用并行，if (total < 100000)，这个数值设置的越小越倾向于做并行操作。通过设置相关变量，传入 pthread_create 相关的参数、函数、线程数组等。在等待所以线程结束后，要把 result 中保存的结果返回给 guesses 中。

4.2.2 openmp

实现逻辑比较简单，我们的确实现了优化，体现了 openmp 的方便与强大。

```
        #pragma omp parallel{
            std::vector<std::string> local_guesses;
            #pragma omp for nowait
            for (int i = 0; i < pt.max_indices[0]; i++){
                string guess = a->ordered_values[i];
                local_guesses.emplace_back(guess);}
            #pragma omp critical{// 合并局部结果
                guesses.insert(guesses.end(), local_guesses.begin(),
                    local_guesses.end());
                total_guesses += local_guesses.size();
            }}
```

可以看到代码量很少，复杂性很低。我们来分析一下这些语句：

#pragma omp parallel 用于启动一个并行区域，openmp 将创建多个线程，所有线程执行这个大括号内的代码块。另外值得一提的是，这里让每个线程都创建一个自己的 local_guesses 向量，避免多个线程直接访问共享资源导致的冲突，这个向量将用于保存该线程生成的 guess 字符串列表。

#pragma omp for nowait 这个语句会告诉编译器，将接下来的 for 循环中的迭代任务分配给多个线程并行执行，而 nowait 关键字表示：执行完该循环的线程无需等待其他线程完成，可以去执行别的任务，减少线程同步的等待时间，适合循环没有依赖关系的代码。

#pragma omp critical 定义了一个临界区，在这个代码块里，同一时刻只允许一个线程执行该代码段，以防止对共享资源的同时访问造成数据错误，用于将每个线程的结果合并到最终结果里，其实也就是在合并数据时进行了加锁解锁，保证了线程安全。

4.3 MPI 多进程优化

单线程处理时，每次扩展和计算概率的复杂度较高，且需要维护和排序大量节点。将扩展任务分配到多个进程中，可以极大缩短整体搜索时间。

由于每个节点的扩展与概率计算是独立操作，且扩展后生成的新节点只需要归并回主队列即可，适合用 MPI 进行分布式并行计算。

我们并行化的思路，就是分配多个进程，把这个任务量比较大的任务细化，分配给多个进程，每个进程处理一部分数据，并在最后总和各个进程完成的任务。

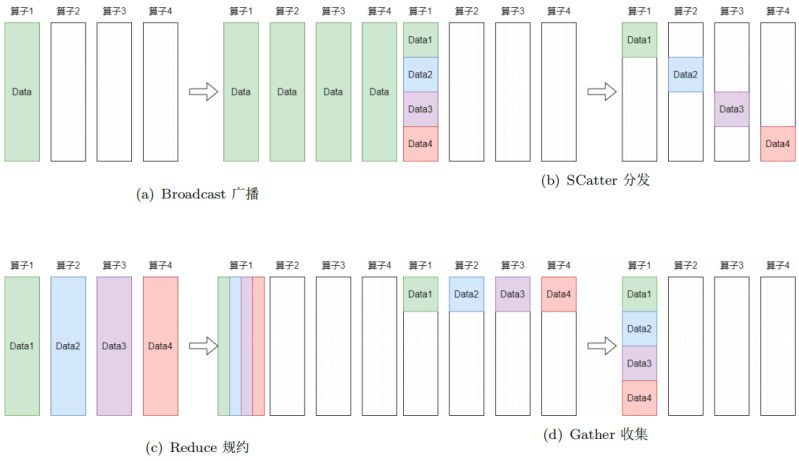


图 4.1: MPI 分布式计算流程

1. 初始化 MPI 环境，获取当前进程编号（rank）和总进程数。
2. 主进程（rank 0）维护全局任务队列，存储待扩展的密码节点。
3. 主进程将任务队列按批次拆分成多个子任务，通过 MPI 发送给各个工作进程。
4. 工作进程接收任务后，独立进行节点扩展和概率计算，并通过 MPI 将结果发送回主进程。
5. 主进程接收各工作进程返回的结果，合并回全局队列，更新搜索状态。
6. 重复分发任务与结果合并的过程，直至满足终止条件。
7. 所有进程完成后，释放 MPI 资源，结束计算。

Generate_mpi 这个函数是整个 MPI 并行计算的核心。它的任务就是根据当前的 PT 节点，生成所有对应的猜测组合，并且把任务合理分配给各个 MPI 进程来并行计算。

```
void PriorityQueue::Generate_mpi(PT pt)
{
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    CalProb(pt);
```

首先，函数会通过 MPI 的接口拿到当前进程的编号 rank 和总进程数 size，这是分配任务的基础。然后，它调用 CalProb（在上次实验已经详细分析过，这里不再赘述）来算出当前 PT 的概率，这个概率会用来计算最终每个猜测组合的概率。

```
    ..............................
        int base_chunk = total / size;
        int remainder = total % size;
        int start_idx, end_idx;
        if (rank < remainder) {// 前remainder个进程多分配一个任务
            start_idx = rank * (base_chunk + 1);
            end_idx = start_idx + base_chunk + 1;
        } else {// 后面的进程分配base_chunk个任务
            start_idx = remainder * (base_chunk + 1) + (rank - remainder) *
                base_chunk;
            end_idx = start_idx + base_chunk;}
```

接着，函数判断这个 PT 的内容段数。如果只有一段内容，也就是只需要猜测一个单独元素，情况比较简单。它先算出这段内容的所有可能取值的总数，比如说这个位置能出现多少不同的字母或数字。然后，它把这个总数平均分成 size 份，分配给每个进程一个区间，让每个进程只负责计算自己区间内的所有猜测。这样不同的进程就不会重复计算，从而就保证了负载均衡。

```
        for (int i = start_idx; i < end_idx; i++) {// 生成密码
            guesses.push_back(a->ordered_values[i]);}
        int local_count = (end_idx - start_idx);// 本地计数
        // MPI合并：收集所有进程的计数
        int global_total = 0;
        MPI_Allreduce(&local_count, &global_total, 1, MPI_INT, MPI_SUM,
            MPI_COMM_WORLD);
        // 更新为全局总计数（类似原始Generate函数的total_guesses += total）
        total_guesses = global_total;}
```

然后，每个进程会遍历自己负责的索引区间，把每个索引转换成实际猜测的内容，计算概率，并把结果保存起来。这部分工作各进程间完全独立，所以不需要通信。在所有进程完成各自计算后，我们可以使用 MPI 的 Allreduce 操作把所有进程产生的猜测数量汇总起来，保证每个进程都知道整个系统中猜测的总数。这一步保证了进程间同步，也方便后续步骤管理猜测等。

```
    else {string prefix;int seg_idx = 0;
        for (int idx : pt.curr_indices) {
            if (pt.content[seg_idx].type == 1) {
                prefix +=
                    m.letters[m.FindLetter(pt.content[seg_idx])].ordered_values[idx];}
            ................
            seg_idx++;
            if (seg_idx == pt.content.size() - 1) {
                break;}}
        segment *a;
        int last_seg_idx = pt.content.size() - 1;
        if (pt.content[last_seg_idx].type == 1) {
            a = &m.letters[m.FindLetter(pt.content[last_seg_idx])];}
```

```
13          ..................}
```

如果 PT 有多段内容的话，情况就相对稍微复杂一点。函数先把除了最后一段之外的内容段做一个前缀组合，也就是说先去确定前面所有 segment 的猜测值，然后只对最后一个 segment 的内容做任务划分。这样就可以把多段组合的任务拆成“前缀 + 后缀”的结构，每个进程只负责不同后缀的猜测部分。

```
1           int total = pt.max_indices[pt.content.size() - 1];
2           int base_chunk = total / size;
3           int remainder = total % size;
4           int start_idx, end_idx;
5           if (rank < remainder) {// 前remainder个进程多分配一个任务
6               start_idx = rank * (base_chunk + 1);
7               end_idx = start_idx + base_chunk + 1;
8           } else {// 后面的进程分配base_chunk个任务
9               start_idx = remainder * (base_chunk + 1) + (rank - remainder) *
                    base_chunk;
10              end_idx = start_idx + base_chunk;}
11          for (int i = start_idx; i < end_idx; i++) {// 生成密码
12              guesses.push_back(prefix + a->ordered_values[i]);}
```

后续同样把最后一段的猜测空间均分，分配给不同进程，让每个进程在固定的前缀基础上遍历自己负责的后缀范围，从而生成完整猜测，计算概率。

```
1           int local_count = (end_idx - start_idx);// 本地计数
2           // MPI合并：收集所有进程的计数
3           int global_total = 0;
4           MPI_Allreduce(&local_count, &global_total, 1, MPI_INT, MPI_SUM,
                MPI_COMM_WORLD);
5           //更新为全局总计数（类似原始Generate函数的total_guesses += total）
6           total_guesses = global_total;}}
```

在所有进程完成各自计算后，再次调用 MPI 的 Allreduce 操作把所有进程产生的猜测数量汇总起来，保证每个进程都知道整个系统中猜测的总数。这一步保证了进程间同步，也方便后续步骤管理猜测。

总结来说，我们的 Generate_mpi 函数通过**合理划分计算任务，把复杂的猜测组合生成工作均匀分配给所有 MPI 进程，既避免重复计算，又能充分利用集群资源**。同时，它利用 MPI 通信来同步和汇总计算结果，确保并行计算的正确和高效。

对于 main 函数，我们给 main 函数设置了两个参数，使得我们可以通过命令行向程序中传入 mpi 初始化的参数，以帮助 mpi 解析命令行参数。这部分主要负责程序的初始化、模型训练、密码生成与哈希处理，并通过 MPI 实现并行计算。程序首先通过 MPI_Init 初始化 MPI 环境，确定当前进程编号（rank）和总进程数（size）。接下来，rank 为 0 的主进程会启动模型训练，并记录训练所耗时间。

训练完成后，所有进程进入密码生成阶段。程序会不断从优先队列中弹出新的猜测，并定期进行全局同步，统计所有进程生成的猜测总数。当生成数量达到一定的阈值时，会触发 MD5 哈希操作，对当前猜测集合执行 MD5 哈希。哈希过程也采用批处理方式，提高效率并避免内存溢出（我们第一个 simd 实验）。

程序设定了猜测上限（如三千万条），一旦达到上限，主进程 rank0 将会输出各阶段的耗时信息，包括训练时间、猜测时间、哈希时间和总运行时间。最后，还会收集所有进程的总耗时，统计最大、最

小和平均执行时间，并计算负载均衡效率，从而评估并行效果和资源利用情况。

并且我们按照了报告里的要求，计时测试工具使用 MPI 进行计时（MPI_Wtime()），以获得更精确的时间测量。

由于 mpi 中的每个进程内仍使用的是 CPU 串行逻辑处理，所以我们在后续添加了基于 GPU 的并行计算逻辑，实现 MPI 多进程并行 +GPU 在每个进程内并行加速。

4.4 GPU 加速方案

图 4.2: CPU 与 GPU 架构对比

1. 初始化 CUDA 环境，分配 GPU 内存资源
2. 主线程维护优先级队列，存储待处理的口令节点
3. 主线程根据阈值判断，将任务按批次分配给 GPU 或 CPU
4. GPU 工作线程接收任务后，并行执行字符串拼接和概率计算
5. GPU 计算完成后，异步传输结果回主线程，更新全局队列
6. 重复任务分发与结果合并过程，直至满足终止条件
7. 所有计算完成后，释放 GPU 资源，清理 CUDA 环境

通过以上步骤，我们通过有效利用 CUDA 计算资源，通过阈值优化、异步处理和内存池管理，从而实现了口令猜测的 GPU 加速。

```
#define GPU_ACCELERATION_THRESHOLD 1000
#define CHECK_CUDA_ERROR(operation) do // CUDA运行时错误检测宏定义
```

首先我们进行了两个宏定义。GPU_ACCELERATION_THRESHOLD 定义了 GPU 加速处理的数据量阈值，当待处理数据量达到 1000 时启用 GPU 并行计算，否则就使用 CPU 串行处理。通过这个，当数据量比较小的时候，我们就会采取 CPU 计算，避免了 GPU 的大幅度开销。第二个宏定义用于监测程序是否出错。

```
__global__ void string_concatenation_kernel // CUDA核心函数：字符串拼接处理
```

在这里我们定义了一个函数，GPU 并行字符串拼接核函数。该函数的实现主要是，每个 CUDA 线程负责处理一个字符串的拼接操作，首先会获取线程索引，确定要处理的数据项，然后将前缀字符串与

输入字符串合并生成完整的口令候选，并且要注意在最后添加字符串结束符。通过并行执行实现高效的批量字符串处理。

然后我们定义了一个 GPU 资源管理类，用于对 CUDA 进行内存分配、数据传输和核函数执行的统一管理。其中构造函数会通过类中变量分配 GPU 内存空间，输入空间为 60MB，输出空间为 120MB，通过 cudaStreamCreate 创建 CUDA 流用于异步操作管理，还会设置一些数组、变量等。析构函数会释放所有 GPU 内存资源并销毁 CUDA 流。

其中的 process_string_batch 函数是我们实现的核心。伪代码如下：

```
function process_string_batch(input_data, input_count, output_buffer):
    // 1. 输入验证
    if input_count > MAXIMUM_ITEM_COUNT:
        return error
    // 2. 计算内存需求
    total_input_size = 0
    for each item in input_data:
        total_input_size += item.length
    if total_input_size > MAXIMUM_INPUT_CAPACITY:
        return error
    // 3. 分配主机内存
    allocate host_input_buffer[total_input_size]
    allocate host_offset_array[input_count]
    allocate host_length_array[input_count]
    allocate host_output_buffer[MAXIMUM_OUTPUT_CAPACITY]
    // 4. 数据组织
    current_offset = 0
    for i = 0 to input_count-1:
        copy input_data[i] to host_input_buffer[current_offset]
        host_offset_array[i] = current_offset
        host_length_array[i] = input_data[i].length
        current_offset += input_data[i].length
    // 5. 异步数据传输到GPU
    async_copy host_input_buffer to device_input_buffer
    async_copy host_offset_array to device_offset_array
    async_copy host_length_array to device_length_array
    // 6. 启动GPU核函数
    launch string_concatenation_kernel
    // 7. 异步传输结果回主机
    async_copy device_output_buffer to host_output_buffer
    // 8. 同步等待完成
    wait_for_gpu_completion()
    // 9. 处理结果
    copy host_output_buffer to output_buffer
    // 10. 清理资源
    free.......
    return success
end function
```

该函数主要负责实现字符串批量拼接的完整 GPU 加速流程。该方法首先进行严格的**输入验证**，检查

输入字符串数量是否在预设的 MAXIMUM_ITEM_COUNT 范围内，并计算所有字符串的总字节数是否超过输入容量的限制，确保不会超出 GPU 内存容量。接着，该函数会在主机内存中准备三个很重要的**数据结构**：host_input_data 连续存储所有字符串的原始内容，并通过**扁平化处理**消除字符串间的间隙；host_offset_data 记录每个字符串在连续内存中的起始位置；host_length_data 存储各字符串的长度信息，这三个数组共同构成了 GPU 处理所需的完整输入描述。随后我们根据字符串数量和预设的最大字符串长度来**计算输出缓冲区大小**，并检查是否超出最大输出容量，以确保输出空间充足。

在这个函数的核心部分，我们采取了**异步流水线设计策略**：首先通过 cudaMemcpyAsync 将输入数据、偏移数组和长度数组并行拷贝到设备内存，同时处理前缀字符串的拷贝；然后配置 CUDA 内核的执行参数，并启动我们之前实现的 string_concatenation_kernel 核函数进行并行拼接；最后异步将结果拷贝回主机。

在整个过程中，所有操作都绑定到同一个 CUDA 流中从而实现操作序列化，并通过 cudaStreamSynchronize 确保所有异步操作完成。在最后的结果处理阶段，我们将设备返回的扁平化结果数据重新组织为 vector<string> 的格式，将每个字符串从结果缓冲区的固定偏移位置提取，并在最后清理临时内存，然后返回成功状态。

通过这个函数，我们不仅从内存角度最大化了内存访问效率，而且通过异步操作的流水线设计降低了等待延迟，从而加速了计算能力。另外特别值得注意的是，我们通过固定长度输出缓冲区的设计从而避免了动态内存分配，以实现更好的性能。

然后我们定义了几个接口函数，initialize_cuda_environment 是 CUDA 环境初始化函数，用于创建我们上面定义的全局 GPU 资源管理器实例，确保 GPU 资源的正确初始化。cleanup_gpu_resources 是资源清理函数，在 main 函数中被调用，用于释放 CudaComputeManager 实例，回收所有 GPU 相关资源。processWithGPU 是对外的 GPU 处理接口函数，封装 GPU 批量处理逻辑，提供统一的 GPU 加速服务。确保 GPU 环境已初始化后再调用批量处理函数。

```
void PriorityQueue::Generate(PT pt)
═══════════════════ for循环 ═══════════════════
        int num_values = pt.max_indices[0];
        if (num_values >= GPU_ACCELERATION_THRESHOLD) {
            vector<string> temp_results;
            if (processWithGPU(a->ordered_values, "", temp_results)) {
                guesses.insert(guesses.end(), temp_results.begin(),
                    temp_results.end());
                total_guesses += temp_results.size();
                return;}}
        // GPU处理失败或数据量不足，使用CPU处理
        for (int i = 0; i < num_values; i++){
            guesses.push_back(a->ordered_values[i]);
            total_guesses++;}
```

以上是我们 PCFG 算法的实现区域，我们在这里主要说明 Generate 函数中的修改，主要修改也就是在**检查阈值**后，首先会去尝试进行 GPU 处理，当数据量很小的时候，便会回退到 CPU 处理，从而减少了 GPU 分配内存的开销，提高了性能。

在这次实验中，当我实现了以上的并行算法之后，我发现并没有得到很好的效果，guess time 仍和串行差不多甚至更慢一些，为此我尝试了进阶选题，在尝试批量处理 pt 时，错写了一版代码，但没想到收获到了很好的效果。修改如下：

```
void PriorityQueue::PopNext()
{
    const int BATCH_SIZE = 64;// 批处理大小
    vector<PT> batch_pts;
    for (int i = 0; i < BATCH_SIZE && !priority.empty(); i++) {
        batch_pts.push_back(priority.front());
        priority.erase(priority.begin());
    }
    if (!batch_pts.empty()) {
        BatchGenerate(batch_pts);
    }
```

可以看到我们对 PopNext 函数进行了修改，每次批量处理 64 个 pt，调用 BatchGenerate 函数。

```
void PriorityQueue::BatchGenerate(const vector<PT>& batch_pts) {
    for (const PT& pt : batch_pts) {
        Generate(const_cast<PT&>(pt));
    }
}
```

这里对 BatchGenerate 函数进行了实现，其会调用 Generate 函数进行处理。但是，很明显可以发现，我们通过这两个函数，其实并没有实现批处理，本质上还是对于每一个 pt 通过 for 循环调用 generate 函数每次处理一个 pt，这是一个伪批处理的串行处理过程，但是在我进行测试后，在正确性得到保证的前提下，其性能明显提高，优于串行算法。

为此，我进行了**一些思考：为什么同样是串行处理，后者却比前者效果好得多？**

首先，可以看到，这样处理之后，我们调用 PopNext 的次数要减少很多，那么就有可能是我们在 PopNext 函数中花费了很多时间。原来的代码中每处理一个 PT 就执行一次 priority.erase(priority.begin())，这是 O(n) 复杂度的操作。而 guessing_gpu.cu 批量删除 64 个元素，虽然单次成本更高，但**总体删除次数减少 64 倍，显著降低了队列维护开销**。

其次，也有可能是因为**批量数据的缓存友好性**，batch_pts 向量虽然逻辑上仍是串行处理，但 64 个 PT 对象在内存中连续存储，提高了 CPU 缓存的命中率。相比 example 文件每次只访问 priority.front()，批量访问能更好地利用缓存行，减少内存访问延迟。

另外，通过我们在计算机组成原理课上学到的**分支预测**也可以解释说明这一点，在修改后的代码中，统一的循环结构减少了复杂的条件分支，使 CPU 的分支预测器能更准确预测执行路径，减少流水线停顿。**从指令的角度**来看，BatchGenerate 函数中的简单 for 循环使得 Generate 函数的指令序列被重复执行 64 次，这些指令能长时间驻留在 CPU 指令缓存中，提高指令缓存命中率。**从调度的连续性角度**来看，批量处理的连续模式减少了操作系统上下文切换，从而收到了较好的效果。

从内存管理的角度来看，64 个 PT 在短时间内连续调用 processWithGPU，使 CUDA 运行时能更有效管理 GPU 内存分配释放，减少内存碎片化。并且**连续的 GPU 调用**能保持 GPU 活跃状态，避免频繁的空闲激活状态间的切换，从而提高整体吞吐量，减少了内存碎片和分配的开销。

这并非巧合，而是一种可以考虑并且可行的并行算法，也给了我在以后的并行算法设计时提供了一个很好的方向，看似无用的设计也能有可能带来很好的效果。

4.5 四种并行算法的设计对比

在本学期的实验中，我们针对 PCFG 口令猜测算法实现了四种不同的并行化方案，每种方案都展现出了独特的设计特点和适用场景。从最底层的 SIMD 指令级并行到最高层的 MPI 分布式计算，这些方案构成了一个完整的并行计算技术集合，为算法优化提供了多维度的解决方案，也为我以后在算法优化领域提供了很好的经验和思路。

我们的 **SIMD 向量化方案（基于 ARM 架构的 NEON 以及基于 X86 架构的 SSE）充分利用了现代处理器的单指令多数据能力**，特别适合处理算法中大量重复的 MD5 哈希计算。通过将 128 位的哈希计算分解为 4 个 32 位的并行处理单元，我们实现了计算资源的细粒度利用。这种方案的优势在于其极高的能效比，我们不需要额外的线程或进程开销，仅通过硬件层面的并行指令即可以获得加速。然而，它的局限性也很明显，只能应用于高度规则化的数据并行任务，比如我们的数据已经限制了其长度所以能够这样进行设计，对算法中不规则的计算部分可能就难以发挥作用。

多线程方案则着眼于任务级并行，通过创建线程池来并发处理不同的口令生成任务。这种方案充分利用了现代多核处理器的计算能力，特别适合处理 pt 概率语法树中可独立扩展的分支节点。与 SIMD 相比，多线程并行的优势在于其灵活性，可以处理更加复杂、不规则的计算任务。但随之而来的是复杂的线程管理和巨大的同步的开销，特别是在任务分配不均时容易出现负载失衡的问题，易造成瓶颈。也就是说，对于多线程编程，并不一定是线程数越多越好，当问题规模较小时明显没必要，就算问题规模比较大时，可能会由于频繁地创建和销毁线程导致时间效益明显下降，正如我们动态编程，所以我设置的线程数为 4 而不是 8。但对于基于 pthread 的静态线程池编程，我们只在主程序的开始创建一次线程，在主程序的结束销毁一次线程，就不用考虑这个问题，可以把线程数设置为 8，使用多线程数带来的收益远远高于创建的损失。

MPI 分布式计算方案将并行规模扩展到了进程级别，通过多台机器的协作来应对超大规模的口令猜测任务。这种方案采用经典的主从架构，由主进程负责任务分配和结果收集，从进程执行实际计算。MPI 的优势在于其近乎线性的扩展能力，通过增加计算节点可以持续提升整体吞吐量。但分布式计算也带来了显著的数据通信开销，特别是在频繁交换任务状态时，网络延迟可能成为性能瓶颈。对于多进程编程，也是和我们上面的多线程编程一样，并不是进程数量越多越好，太多了会带来额外的开销，太少的话并行度不够，没有充分利用 cpu 资源。所以我们要在并行度和开销之间找到最佳平衡点。我们通过修改进程数进行了实验，数据显示，在 4 进程配置下我们获得最好的加速比，但进程数继续增加时收益开始递减。

GPU 加速方案则代表了对大规模数据并行计算的极致，通过 CUDA 架构的数千个线程并发执行口令生成和哈希计算（在后面具体实现）。这种方案特别适合处理算法中高度并行的批量操作，如同时生成大量候选口令并进行哈希验证，与 CPU 方案相比，GPU 的优势在于其惊人的吞吐量，但代价是较高的编程复杂度和设备间的数据传输开销，这便可能成为瓶颈。我们的实验结果表明，当批量处理 2000 个以上字符串时，GPU 方案开始展现出显著优势，但当处理较小规模的任务时，反而可能因为启动销毁的开销而导致性能下降。

综合分析以上这四种并行方案，我们可以发现一个很明显的规律：随着并行粒度的增大，从指令级到线程级再到进程级，方案的适用场景逐渐从细粒度规则计算转向更加粗粒度的复杂任务。SIMD 和 GPU 擅长处理高度并行的数据计算，而多线程和 MPI 更适合管理复杂的任务依赖关系。**在实际的应用中，一个最好的并行算法往往需要混合使用这些技术，根据算法不同阶段的不同特点选择最适合的并行方案**。例如，可以使用 MPI 在节点间分配任务，在每个节点内部使用多线程处理任务分支，在核心计算部分又采用 SIMD/GPU 优化，从而构建一个多层次的混合并行系统。

5 实验与分析（加速比对比）

在这一部分，我们对这几种不同的并行算法的对于我们的口令猜测 PCFG 算法加速效果进行对比分析。

表 1: 不开编译优化加速比对比

问题规模	**SIMDhash**	pthreadguess	openmpguess	MPIguess	GPUguess
500w	**0.6558**	1.064	1.338	0.994	1.712
1000w	**0.6351**	1.098	1.488	1.167	1.243
2000w	**0.6631**	1.087	1.523	1.101	1.347
3000w	**0.6406**	1.084	1.561	1.171	1.170

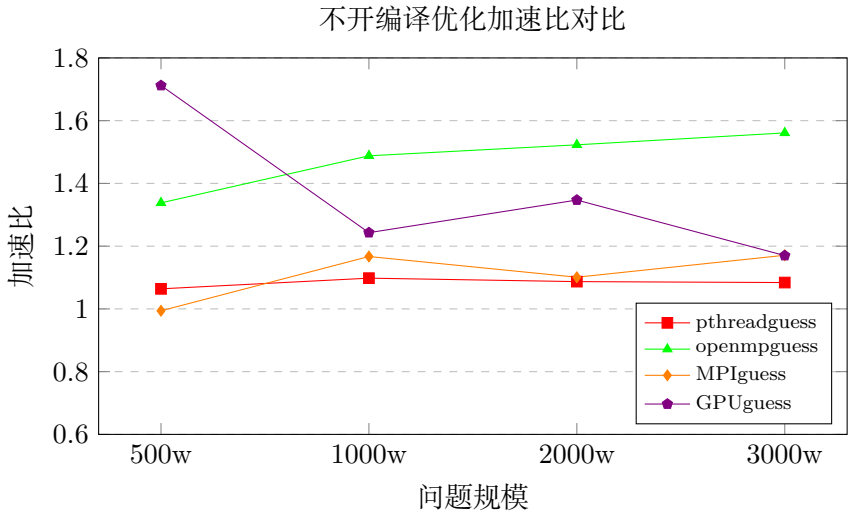
表 2: O1 编译优化加速比对比

问题规模	**SIMDhash**	pthreadguess	openmpguess	MPIguess	GPUguess
500w	**1.7035**	1.059	1.111	1.166	0.698
1000w	**1.4187**	1.057	1.203	1.251	0.665
2000w	**1.6696**	1.073	1.214	1.277	0.734
3000w	**1.6677**	1.104	1.316	1.268	0.715

表 3: O2 编译优化加速比对比

问题规模	**SIMDhash**	pthreadguess	openmpguess	MPIguess	GPUguess
500w	**1.5721**	1.114	1.143	1.07	0.698
1000w	**1.7147**	1.135	1.222	1.119	0.657
2000w	**1.4957**	1.153	1.260	1.316	0.748
3000w	**1.5867**	1.172	1.243	1.177	0.722

实验结果如上所示，在表格显示了对不同编译条件下、不同数据规模问题在不同并行算法下的加速比对比。为了进一步分析，我们绘制如下的折线图，其中由于 simd 算法是对 MD5hash 部分的并行加速，在接下来这一部分先不体现，在后续与 gpu 处理 md5hash 部分我们再进行分析。这个折线图

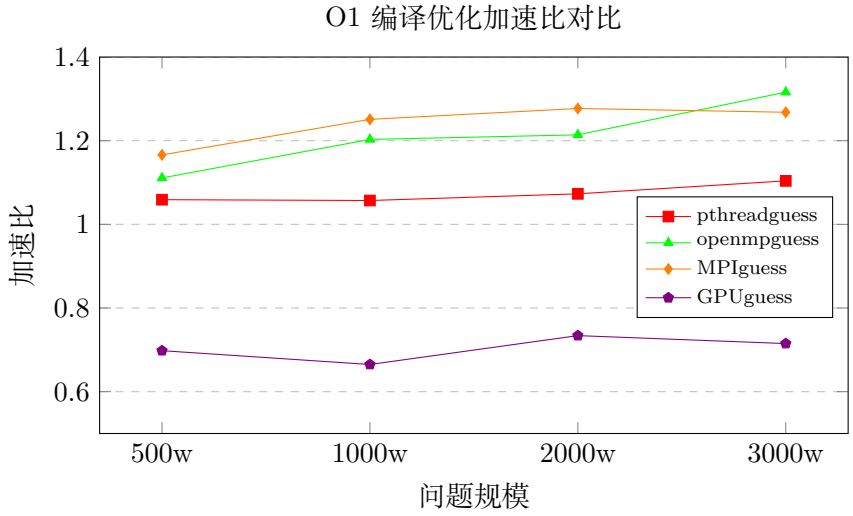


体现了四种算法在不开编译优化的情况下，在不同数据规模下的对于 guess 部分的加速比对比。从图

中可以看出，在不开启编译优化的条件下，**比较稳定且优化效果比较好的是 openmp 算法**，这也体现了 openmp 多线程编程的优越性。并且随着问题规模的增大，guess time 加速比也在逐渐增大，这与我们的预期相符，数据规模越大，多线程可以并行使用多个核，加速计算，多线程并行编程的效果理论上也会更好。

优化效果排名第二的应该是 **GPU，这说明我们的 CUDA 函数确实发挥了优秀的作用**。其中在数据规模为 500w 的情况下，加速比甚至达到了 1.712。但是比较不合乎我们认知的就是，随着数据规模的增大，gpu 的加速比并没有得到提升，反而下降了，我觉得出现这个情况有很多种原因，一个原因就是我们的程序中有很多**参数**，比如批处理的大小，分配的 GPU 内存大小等等，有可能是批处理太小，那么批次就会变多，会增加内存开销，频繁的内存分配和释放可能导致 GPU 内存碎片化，碎片化严重时，即使总内存足够，也可能无法分配连续的大块内存，GPU 内存太小，而导致在处理大数据时，原先分配的内存并不足够，可能会回退到 CPU 处理，所以优化效果并不如小规模数据。还有一个原因可能是数据规模增大时，**串行部分的比重增加**，从而导致并行加速比会下降。

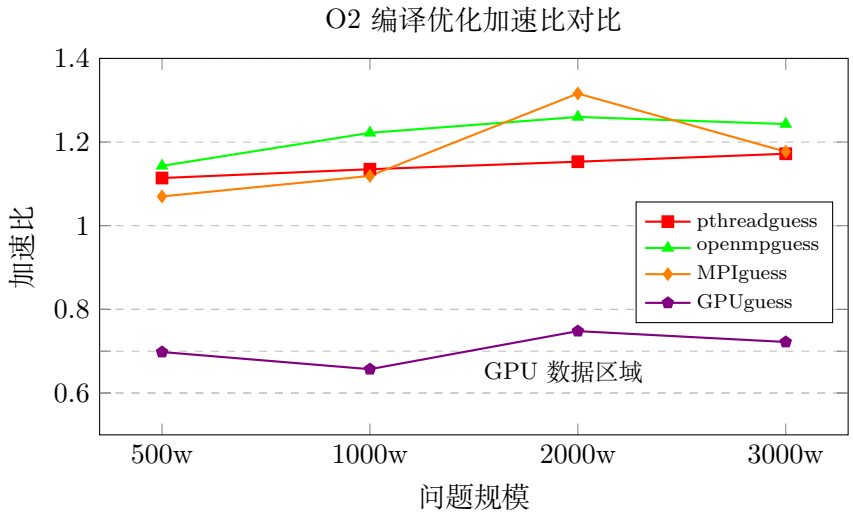
接着是 **pthread 和 mpi 算法，其虽然实现了加速，但是效果并不是很好**，这主要是由于 pthread 算法具有复杂的线程管理和巨大的同步的开销，mpi 算法也类似，具有显著的数据通信开销，导致优化效果并不理想，但是由折线图的走向可以看出，随着数据规模的增大，mpi 和 pthread 算法的加速效果应该越来越好，当处理任务规模更大的问题时，多个进程/线程可以更充分地利用多核资源，整体计算速度也就越快，这时采取 mpi 和 pthread 算法就会有很好的效果。



在 O1 编译优化条件下，整体上加速比下降了，这主要是由于在 O1 编译优化的情况下会进行循环展开，导致串行算法大幅优化，使得加速比得到了下降。但是我们可以看到这时候 mpi 算法得到了更好的效果，与 openmp 不相上下，这是由于在 mpi 算法中，在每个进程内部，仍然进行的是 CPU 串行处理，所以循环进一步展开会对其进行加速。

pthread 算法仍是实现了加速，但加速效果并不理想。对于 GPU 算法来说，在 O1 的情况下，可以看到加速比小于 1，为负优化，也就说明了，O1 编译条件对于串行算法的优化效果明显优于并行算法。我认为是有以下几个原因引起的：首先，我们都知道，CPU 之所以强大，有很大一部分归功于其所具有的**分支预测功能**，在没开编译优化的情况下，分支预测失效，缓存利用率差，而在 O1 编译优化的条件下，串行算法的分支预测进一步增强，减少分支预测失败的开销，从而使得其优化效果更好一些。其次，对于串行算法来说，其循环次数更多，那么通过**循环展开或向量化**等，就可以加速对字符串的处理，通过函数内联优化也可以对频繁的函数调用进行优化。并且 O1 优化还会进行**指令重排**，会提高指令级并行度和流水线效率，从而使得串行算法优化效果很好。

而对于 CUDA 而言，**CUDA 编译器 nvcc 本身已经高度优化**，O1 编译优化对其提升较小（相对于 g++ 来说），并且我们详细分析可以发现，O1 优化主要在指令级进行优化，即减少指令条数，而**GPU 的性能瓶颈主要在内存带宽和并行度，而非指令级优化**，另外 CPU 和 GPU 之间的数据传输开销也不会因编译优化而减少。



在 O2 编译优化的条件下，可以看到大体上与 O1 的情况差不多，我们不再赘述。

综合来看，**不同并行技术对编译优化的响应差异显著**。

6 结论

通过以上的部分，我们完成了对本学期学到的四种核心并行编程思路的分析，各个算法的并行角度不同、实现不同，给我们提供多种并行思路。基于本学期的学习，我们可以归纳出以下的对比表格：

表 4: 四种并行算法理论特性对比

特性	SIMD	多线程	多进程 (MPI)	GPU
并行粒度	指令级	线程级	进程级	线程级
硬件基础	向量寄存器	多核 CPU	多节点集群	流处理器
内存模型	共享内存	共享内存	分布式内存	分层内存
通信机制	寄存器传输	共享变量	消息传递	共享内存 + 全局内存
适用场景	规则数据并行	任务并行	分布式计算	大规模数据并行

接下来我们基于本学期的学习，尝试进行创新工作。

7 额外工作 (创新)

7.1 基于 GPU 的 MD5hash

7.1.1 算法设计

```
// MD5块处理
function MD5_PROCESS_BLOCK(state, block):
    a, b, c, d = state[0], state[1], state[2], state[3]
```

```
    for i = 0 to 15:
        FF(a,b,c,d, block[i], shift[i], CONSTANTS[i])
        循环变量 a,b,c,d
    for i = 0 to 15:
        GG(a,b,c,d, block[索引], shift[i], CONSTANTS[16+i])
        循环变量 a,b,c,d
    for i = 0 to 15:
        HH(a,b,c,d, block[索引], shift[i], CONSTANTS[32+i])
        循环变量 a,b,c,d
    for i = 0 to 15:
        II(a,b,c,d, block[索引], shift[i], CONSTANTS[48+i])
        循环变量 a,b,c,d
    state[0] += a state[1] += b state[2] += c state[3] += d
```

我们设计的这个 **MD5 块处理函数 cuda_md5_process_block 实现了 MD5 算法的核心计算逻辑**，负责处理每个 512 位的数据块。函数接收当前的哈希状态和一个 512 位的消息块作为输入，通过四轮变换来更新状态。我们首先将输入的四个 32 位状态值复制到局部变量 a、b、c、d 中，然后依次执行四轮变换操作。每轮变换包含 16 次操作，总共进行 64 次计算。第一轮使用 FF 函数按顺序处理消息块中的 16 个 32 位字，第二轮使用 GG 函数以特定的跳跃模式访问这些数据，第三轮和第四轮分别使用 HH 和 II 函数采用不同的访问顺序。每次操作都会结合消息数据、预定义的常量和不同的位移量来更新状态变量。通过这种设计，我们可以确保输入数据的每一位都能影响到最终的哈希结果，同时通过不同轮次的不同处理方式来增强算法的安全性。处理完成后，函数将变换后的结果累加到原始状态上，实现状态的更新。

```
function CUDA_MD5_KERNEL(inputs, lengths, results, num_inputs, max_length):
    thread_id = blockIdx.x * blockDim.x + threadIdx.x
    if thread_id >= num_inputs: return
    // 获取当前线程的输入
    current_input = inputs[thread_id * max_length]
    current_length = lengths[thread_id]
    // 初始化MD5状态
    state = [0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476]
    // 预处理字符串
    padded_blocks, num_blocks = STRING_PROCESS(current_input, current_length)
    // 处理每个512位块
    for i = 0 to num_blocks-1:
        MD5_PROCESS_BLOCK(state, padded_blocks[i*16])
    // 字节序转换并存储结果
    for i = 0 to 3:
        results[thread_id*4 + i] = BYTE_SWAP(state[i])
```

CUDA 核函数 cuda_md5_kernel 是在 GPU 上整个并行实现的核心，它在 GPU 上为每个输入密码分配一个线程进行独立处理。函数开始时通过线程索引计算从而确定当前线程要处理的具体口令，并进行边界检查以避免越界访问。接下来，函数初始化 MD5 算法的标准起始状态，这是一组固定的 32 位常量（我试了一下，放在数组里由于连续访问比不放在数组里的直接使用要快）。然后调用字符串预处理函数将输入密码转换为符合 MD5 标准的数据块格式，包括添加填充位和长度信息。对于生成的每个数据块，函数都会调用 cuda_md5_process_block 进行处理，逐步更新哈希状态。计算完成

后，函数执行字节序转换，将 GPU 内部使用的小端格式转换为标准的大端输出格式，并将最终的 128 位哈希值存储到全局内存中供主机读取。通过这种设计，我们实现了真正的并行计算，数千个线程可以同时处理不同的密码，大大提高了计算的效率。

```
function MD5_HASH_CUDA(inputs):
    num_inputs = inputs.size()
    max_length = 找到最大字符串长度
    // 分配主机内存
    h_inputs = 分配(num_inputs * max_length)
    h_lengths = 分配(num_inputs)
    h_results = 分配(num_inputs * 4)
    // 准备输入数据
    for i = 0 to num_inputs-1:
        复制inputs[i]到h_inputs[i*max_length]
        h_lengths[i] = inputs[i].length()
    // 分配GPU内存
    d_inputs = GPU分配(num_inputs * max_length)
    d_lengths = GPU分配(num_inputs)
    d_results = GPU分配(num_inputs * 4)
    // 数据传输到GPU
    复制h_inputs到d_inputs
    复制h_lengths到d_lengths
    // 启动CUDA核函数
    grid_size = (num_inputs + block_size - 1) / block_size
    CUDA_MD5_KERNEL<<<grid_size, block_size>>>(d_inputs, d_lengths, d_results,
        num_inputs, max_length)
    // 等待GPU完成
    同步GPU
    // 复制结果回主机
    复制d_results到h_results
    // 整理结果格式
    results = 转换h_results为二维数组格式
    // 释放内存
    释放所有分配的内存
    return results
```

MD5Hash_CUDA 函数是整个 CUDA MD5 实现的主控函数，主要负责协调 CPU 和 GPU 之间的数据传输和计算调度。我们首先分析输入的密码列表，确定最大字符串长度并进行内存对齐优化，以提高 GPU 内存访问效率。然后在 CPU 端分配临时的内存空间，将输入的字符串向量转换为 GPU 友好的连续内存布局，同时记录每个字符串的实际长度。接下来，函数在 GPU 端分配相应的内存空间，并将准备好的数据从 CPU 传输到 GPU。在执行配置阶段，我们根据输入规模动态计算网格大小和线程块大小，从而确保对 GPU 计算资源的充分利用。核函数启动后，主机端等待 GPU 完成所有计算任务。计算完成后，函数将结果从 GPU 内存传输回 CPU，并转换为标准的二维向量格式以便后续使用。最后，函数释放所有分配的内存资源，避免内存泄漏。

7.1.2 结果分析

实验结果如下：

表 5: 无编译优化条件下的性能对比

数据规模	GPU 计算时间 (s)	串行计算时间 (s)	加速比
500w	2.24467	2.77645	1.237
1000w	4.71646	6.23402	1.322
2000w	9.36105	12.6429	1.351
3000w	13.7941	18.9401	1.373

在无编译优化条件下的数据可以看出，我们的 GPU 计算相比串行计算展现出了稳定的性能优势。在 500w 数据规模下，GPU 计算时间为 2.24467 秒，而串行计算需要 2.77645 秒，加速比达到 1.237 倍。随着数据规模的增长，这种优势逐渐扩大：1000w 数据规模下加速比提升至 1.322 倍，2000w 规模达到 1.351 倍，3000w 规模进一步提升到 1.373 倍。这种趋势表明 GPU 并行计算的优势随着计算量的增加而更加明显，与我们的预期相符。

表 6: O1 编译优化条件下的性能对比

数据规模	GPU 计算时间 (s)	串行计算时间 (s)	加速比
500w	0.782126	0.99368	1.270
1000w	1.41771	2.22803	1.572
2000w	2.67239	4.52045	1.692
3000w	3.87519	6.75098	1.742

而在 O1 编译优化的条件下，GPU 算法也表现出了很好的对 hash time 的加速效果。但是可以看到，在 O1 编译下的加速比高于不开编译优化的加速比，这在一定程度上与我们之前的解释（O1 对 gpu 优化效果相对串行不高）不太符合。可能的原因是，在 MD5hash 算法中，在没有编译优化的情况下，GPU 代码可能存在一些低效的实现，比如未优化的内存访问模式、过多的寄存器溢出、或者分支分歧等问题，这些问题严重影响了 GPU 的并行效率，使得 GPU 性能甚至不如 CPU。而当启用 O1 优化后，CUDA 编译器能够识别并修复这些问题，包括改善内存合并访问、优化寄存器使用、减少线程的分支分歧等，从而提高了 GPU 的并行计算潜力。**这也让我意识到，对于不同问题来说，同一编译优化选项也可能造成不同的结果**。

表 7: O2 编译优化条件下的性能对比

数据规模	GPU 计算时间 (s)	串行计算时间 (s)	加速比
500w	0.807259	0.954543	1.182
1000w	1.41326	2.15268	1.523
2000w	2.67896	4.38973	1.639
3000w	3.85293	6.4932	1.685

O2 编译优化下表现与 O1 差不多，不再赘述。

7.1.3 与基于 simd 的 md5hash 对比

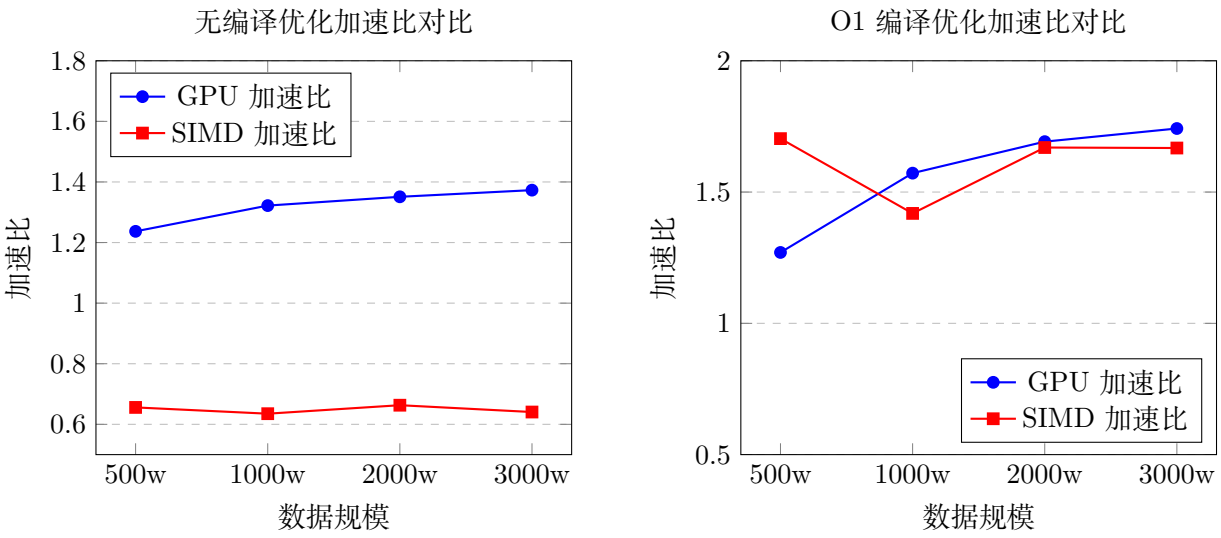


图 7.3: GPU 与 SIMD 加速比在不同编译优化条件下的对比

我们在这一部分进行该 GPU 算法与基于 simd 的 md5hash 的对比，折线图如上所示，描述了无编译优化和 O1 编译优化两种情况下的加速比对比。可以看到，在不开启编译优化的条件下，与 simd 相比，我们的基于 gpu 的 md5hash 算法加速效果更好，并且从折线走向来看，随着数据规模的增大，加速效果越来越好。

在 O1 编译优化的条件下，由于**循环展开**、**分支预测**、**函数内联展开**等，simd 算法进行了大幅度加速。起初由于数据规模太小，GPU 算法由于内存分配等开销而不如 simd 算法，但随着数据规模的增大，GPU 处理大规模数据的优异性开始体现，加速效果开始超过 simd 算法，并随着数据规模的增大加速效果越来越好。

7.2 基于 MPI+GPU 的 guessing

这部分主要是基于之前做的 MPI 和 GPU 实验进行一个合作实现，主要修改就是把之前的 mpi 部分中的 CPU 串行处理逻辑修改成 GPU 并行处理逻辑，**实现两层并行，在 MPI 进程间并行 + 每个进程内 GPU 并行加速**，并且我们还添加了智能处理逻辑，只有当批量大小达到阈值时才会使用 GPU，批量大小没有达到阈值或者 GPU 失败时都会自动回退到 CPU，这样就实现了动态负载均衡优化，提高代码的鲁棒性。

7.2.1 算法设计

我们设计的基于 MPI+GPU 的 guessing 核心部分的伪代码如下：

```
function Generate_mpi_cuda(pt):
    # 获取MPI进程信息
    rank, size = MPI_Comm_rank(), MPI_Comm_size()
    # 计算概率
    # 获取segment数据
    # MPI负载均衡分配（这部分省略，与mpi的相同）
    local_count = end_idx - start_idx
```

```
    # MPI+GPU协调处理（核心逻辑）
    if (local_count >= GPU_ACCELERATION_THRESHOLD and
        global_cuda_manager and global_cuda_manager.isInitialized()):
        # GPU处理MPI分片
        gpu_success = global_cuda_manager.process_string_batch_mpi(
            segment_data.ordered_values, prefix, temp_results, start_idx, end_idx
        )
        if gpu_success:
            guesses.insert(guesses.end(), temp_results.begin(), temp_results.end())
        else:
            # GPU失败回退CPU
            for i in range(start_idx, end_idx):
                guesses.push_back(prefix + segment_data.ordered_values[i])
    else:
        # CPU处理MPI分片
        for i in range(start_idx, end_idx):
            guesses.push_back(prefix + segment_data.ordered_values[i])
    # MPI全局聚合
    global_total = 0
    MPI_Allreduce(local_count, global_total, MPI_SUM, MPI_COMM_WORLD)
    total_guesses = global_total
```

Generate_mpi_cuda 这个函数是我们设计的 MPI 与 CUDA 混合并行计算的核心实现，通过精心构造多层次并行架构，我们实现了 MPI 分布式计算与 GPU 加速的有机结合。函数首先通过 MPI_Comm_rank 和 MPI_Comm_size 获取当前进程在 MPI 通信域中的标识信息，建立了 MPI 计算的基础框架。随后针对单段内容和多段内容分别实施相应的处理逻辑，采用分支处理策略。

在负载均衡机制方面，该函数采用**动态分配算法**确保各 MPI 进程间的工作负载均衡。通过计算基础块大小和余数分配，我们将总任务量按进程数进行科学分割，其中余数部分优先分配给编号较小的进程，从而实现了近似均匀的负载分布。这种设计有效避免了传统静态分配可能导致的负载不均的问题。

接下来这部分是我们设计的核心，在 **MPI 与 GPU 协调机制的实现**上，该函数建立了基于阈值判断的智能调度策略。我们首先评估当前进程分配的数据量是否达到设定的 GPU_ACCELERATION_THRESHOLD 阈值，同时验证全局 CUDA 管理器的初始化状态，只有在这两个前置条件都满足时才会启用 GPU 加速处理。一旦确定使用 GPU 处理，函数通过调用 **global_cuda_manager->process_string_batch_mpi 方法**，将 MPI 分配的数据片段连同相应的前缀字符串传递给 GPU 进行批量字符串拼接操作。通过这一调用，我们实现了在这个混合并行系统中 MPI 进程空间与 GPU 设备空间之间的关键数据流之间的有效转换。

另外我们还进行了**完备的容错与回退机制**的实现，当 GPU 处理失败或数据量不足以触发 GPU 加速时，算法就会自动回退至 CPU 处理逻辑，从而确保了系统的鲁棒性和可靠性。无论采用何种处理方式，各 MPI 进程均将处理结果存储在本地 guesses 向量中。最终，函数通过 MPI_Allreduce 集合通信操作实现所有进程本地计数的全局聚合，得到系统总的猜测数量统计。这样我们就实现了：**MPI 负责进程间粗粒度并行分工，GPU 负责进程内细粒度并行加速**，两者通过精心设计的数据分片机制和同步协调策略实现了高效的有机结合。

```
void PriorityQueue::BatchGenerate(const vector<PT>& batch_pts) {
    for (const PT& pt : batch_pts) {
```

```
        Generate_mpi_cuda(const_cast<PT&>(pt));
    }
}
```

另外，我们这里还是采用了之前表现较好的**伪多 pt 处理逻辑**，通过使用这个逻辑，我们可以进一步加速。

7.2.2 结果分析

以下我们对不同编译条件、不同数据规模下，我们的 mpi+gpu 混合策略的加速效果进行分析。

表 8: 无编译优化条件下的性能对比

数据规模	MPI+GPU 计算时间 (s)	串行计算时间 (s)	加速比
500w	2.97637	5.31092	1.784
1000w	5.81971	7.69532	1.322
2000w	10.7587	15.1684	1.410
3000w	16.9752	20.6399	1.216

可以看到，在不开启编译优化的条件下，我们确实实现了加速，其中在 500w 时，加速比甚至来到了 1.784。但是可以发现，随着数据规模的增大，加速效果反而下降，这主要是由于 gpu 的限制，主要原因就是我们上面分析到的那些，可能是复杂的内存开销导致的，也可能是数据规模增大时，串行部分的比重增加，从而导致并行加速比会下降。

表 9: O1 编译优化条件下的性能对比

数据规模	MPI+GPU 计算时间 (s)	串行计算时间 (s)	加速比
500w	0.262792	0.261951	0.997
1000w	0.45474	0.431638	0.949
2000w	0.840032	0.83621	0.995
3000w	1.43293	1.19678	0.835

在 O1 编译优化条件下，可以看到加速比小于 1，为负优化，这也主要是由于 GPU 的限制。一个是 CPU 在 O1 编译下其**分支预测功能**更加强大，减少分支预测失败的开销，从而使得其优化效果更好一些；另外，从指令的角度，O1 优化还会进行**指令重排**，会提高指令级并行度和流水线效率，从而使得串行算法优化效果很好；从**循环展开或向量化**的角度，对于串行算法来说，其循环次数更多，那么通过循环展开或向量化等，就可以加速对字符串的处理，通过函数内联优化也可以对频繁的函数调用进行优化。

而对于 CUDA 而言，CUDA 编译器 nvcc 本身已经高度优化，O1 编译优化对其提升较小（相对于 g++ 来说），并且我们详细分析可以发现，O1 优化主要在指令级进行优化，即减少指令条数，而**GPU 的性能瓶颈主要在内存带宽和并行度，而非指令级优化**，另外 CPU 和 GPU 之间的数据传输开销也不会因编译优化而减少。

表 10: O2 编译优化条件下的性能对比

数据规模	MPI+GPU 计算时间 (s)	串行计算时间 (s)	加速比
500w	0.260427	0.252256	0.967
1000w	0.435376	0.407982	0.937
2000w	0.817369	0.792387	0.969
3000w	1.30707	1.13171	0.866

O2 编译条件下与 O1 类似，我们不再赘述。

7.2.3 与单 mpi 及单 gpu 的对比

为了进一步分析，我们作出如下的折线图，来与单 mpi 及单 gpu 进行对比。从折线图中可以看出：

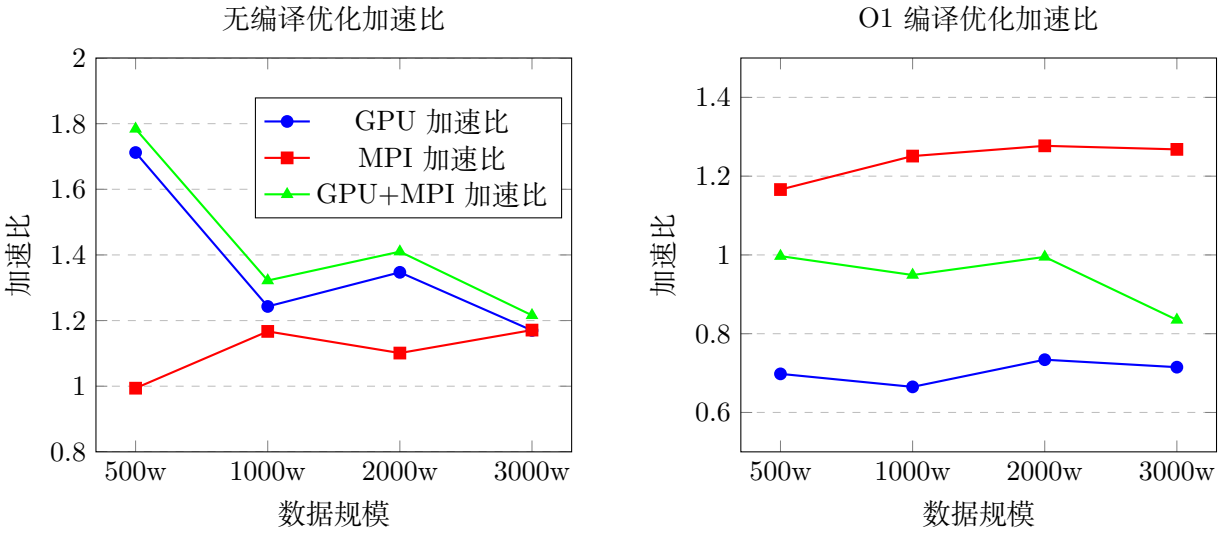


图 7.4: 不同编译优化条件下各并行方案的加速比对比

在无编译优化情况下，从加速比来看，GPU+MPI 混合策略的加速效果最好，其次是 GPU 算法，最后是 MPI 算法，这说明了我们的混合策略的强大，优于 GPU 算法，得益于 MPI 的进程间并行优化；优于 MPI 算法，又得益于 GPU 在进程内的并行计算能力。但是，从总体走向来看，混合策略优于 GPU 的限制，随着数据规模的增大，加速比减小，主要原因我们已经在 GPU 部分进行了分析，这里不再赘述。

在 O1 编译优化的条件下，从加速比来看，MPI 的加速效果最好，这时候我们的混合策略不如 MPI 算法主要还是由于 GPU 的限制，主要原因也已经分析过，不再赘述。而可以明显地看到，我们的混合策略优于单 GPU 的算法，这说明我们的 MPI 在进程间并行优化确实发挥了作用，在一定程度上弥补了 GPU 在开启编译优化条件下加速比反而下降的不足。另外，从总体走向来看，混合策略仍是随着数据规模增大，加速比会逐渐下降。

8 总结反思

8.1 关于串行算法和并行算法的思考

回顾以往的学习，我发现并行化适合处理计算密集型或数据密集型的问题，尤其当任务可以被拆分为多个相互独立的子任务时，其效果尤为显著。这些任务在串行执行时无法充分利用多核 CPU 的并发能力，而通过并行化可极大提高处理速度和系统的资源利用率，从而实现更好的效能。

但是，并行算法并不总是最优选择。当问题规模较小、逻辑依赖强或线程之间共享数据频繁时，并行带来的线程调度、同步开销反而可能超过其带来的性能收益，甚至有可能改变程序的正确性。这类情况下，串行执行具有更低的资源消耗和更高的执行效率，代码也更易维护和调试。

以我们做的多线程口令猜测实验为例，这是一个典型的数据并行问题。每个线程可以独立处理一段候选口令的生成与检验，很适合并行化。但在并行实现中也需注意优化策略，例如避免多个线程同时写入共享内存，而是使用线程本地缓存，并在最终阶段合并结果，或者用一个共享数组实现各个线程读写独立；对于线程计数变量应使用 reduction 避免锁竞争（这个 reduction 是 openmp 的一个指令）；任务调度应采用动态划分策略，防止工作不均衡导致的资源浪费。

总之，要想设计一个并行化算法，**应基于任务特性去权衡开销与收益。在任务独立性强、数据量大时并行化更合适；而在任务轻量或前后逻辑相互串联时，串行处理可能更高效**。

8.2 编译时的选项会对加速比产生怎样的影响

基于 g++ 编译器编译时的优化选项进行调研:

-O1 优化选项

-O1 是 GCC/G++ 的默认优化级别，它提供了一组合理的优化选项，旨在不显著增加编译时间的前提下提高程序的性能。-O1 包括的一些优化技术有：

函数内联：对一些小型函数进行内联展开，减少函数调用的开销。

常量传播：将代码中的常量表达式在编译时计算，减少运行时的计算量。

死代码消除：删除程序中永远不会执行的代码，减少程序的大小。

-O2 优化选项

-O2 优化级别在-O1 的基础上，进一步增加了更多的优化技术，以获得更好的性能提升。这些技术包括：

循环优化：包括循环展开、循环交换等，以减少循环的开销并提高缓存的利用率。

分支预测：优化分支指令，减少分支错误预测的可能性。

指令调度：重新排列指令的执行顺序，以提高流水线的利用率。

-O3 优化选项

-O3 优化级别包括-O2 的所有优化，并进一步应用了一些更激进的优化技术，这些技术可能会增加编译时间，但能获得更多的性能提升。例如：

进一步的指令调度和循环优化。

更激进的函数内联：对更大的函数进行内联展开。

浮点运算优化：对浮点运算进行特定的优化，以提高性能。

使用编译器标志

编译器标志是指导编译器应用特定优化技术的重要手段。通过合理使用这些标志，开发者可以显著提高程序的性能，减少资源消耗，并缩短开发周期。例如-flto（链接时优化）和-march=native（为目标 CPU 优化）。

性能关键路径：可以使用-funroll-loops（循环展开）和-ftree-vectorize（循环向量化）等标志进行针对性优化。

内存敏感应用：可以使用-fdata-sections 和-ffunction-sections 标志来启用数据和函数级别的链接优化，从而减少程序的总内存占用。

代码大小优化：可以使用-Os 标志来优化代码大小，同时尽量保持性能。

另外，使用-fopt-info-优化级别标志可以生成优化信息报告。

8.3 心得体会

至此，本学期的并行实验就结束了。我个人感觉这门课是我这学期为之花费心思最多的一门课，但是也确实是我收获最多的一门课，起初不会连接远程服务器，不知道如何使用脚本文件需要找助教帮助，到现在已经能够熟练掌握几种并行算法的原理以及设计与实现，我真的受益匪浅。在此特别感谢各位助教老师的耐心指导。通过这门课，我重塑了我的编程思维，从过去偏重功能实现转变为现在始终考虑数据局部性、负载均衡和能效比等性能指标，并且让我对之前学过的数据结构有了进一步的思考，对时间复杂度有了切实的具像化认知，这种思维的转变将会持续指导我未来的科研工作。

9 代码链接

点击访问 GitHub 主页