南開大學
Nankai University

计算机学院

并行程序设计期末报告

并行加速算法

姓名：王众

学号：2313211

专业：计算机科学与技术

2025 年 7 月 4 日

目录

1 前期工作总结 **4**

2 SIMD 编程 **4**
2.1 SIMD 任务描述 4
2.2 实验结果 4
2.3 性能的优化可行方案总结 5
2.3.1 打开循环 5
2.3.2 内联函数改宏定义 5
2.3.3 优化内存读取方式 5
2.3.4 尝试过的但是未优化方案 6

3 多线程并行化加速 **7**
3.1 多线程并行化加速任务描述 7
3.2 实验结果 7
3.3 pthread 并行加速 7
3.3.1 动态缓存池并行算法未加速的结果分析 7
3.3.2 静态缓存池并行算法未实现加速的结果分析 8
3.4 openmp 并行加速同样未实现加速 8
3.4.1 高指令开销分析 8
3.4.2 循环控制开销详细分析 9
3.4.3 函数调用开销分析 9
3.4.4 内存访问效率问题 10
3.4.5 临界区性能详细分析 10
3.5 多线程并行化任务的加速算法–动态负载均衡 11
3.5.1 关于现在 openmp 能实现优化的实现总结 12
3.6 有关于并行化实现的适用情况 12
3.6.1 简单的逻辑分析 12
3.6.2 优化尝试 12
3.6.3 优化任务粒度 13

4 多进程并行化加速 **13**
4.1 多进程并行化加速任务描述 13
4.2 实验结果 13
4.3 单纯的多进程并行未能实现加速的原因 14
4.3.1 频繁的全局同步操作 14
4.3.2 负载不均衡 14
4.3.3 通信与计算比例失衡 14
4.4 不同进程数对于时间加速的影响 14
4.4.1 原因分析 15
4.5 为什么我们实施的大多数有效的加速都会使得 hash 过程加速 15
4.5.1 Hash 过程是真正独立并行的 16

4.5.2 不同优化对两个过程的影响差异 . . . 17
4.5.3 算法复杂度的差异 . . . 17
4.6 将结束标志改为 hash 产生 10000000 个结果 . . . 17
4.6.1 MPI 与多线程混合环境的固有挑战 . . . 18
4.6.2 时序依赖性问题 . . . 18
4.6.3 数据一致性窗口 . . . 18
4.6.4 通信模式的影响 . . . 18

5 GPU 并行加速 19
5.1 GPU 并行化任务描述 . . . 19
5.2 实验结果 . . . 19
5.3 单纯的 GPU 并行化并没有实现加速的原因 . . . 19
5.3.1 GPU 内存分配/释放开销过大 . . . 19
5.3.2 频繁的 CPU-GPU 数据传输 . . . 20
5.4 多 PT 的 GPU 加载运行也未实现加速的原因 . . . 20
5.5 CPU-GPU 异步并行处理未实现加速的原因 . . . 21
5.5.1 数据规模不匹配 GPU 特性 . . . 21
5.5.2 性能提升的核心原因 . . . 22
5.6 在多 PT 的基础上再加上动态调整机制实现了加速的原因 . . . 22
5.6.1 动态调整因素 . . . 22
5.6.2 编译优化后的关键变化 . . . 23

6 以上所有实验总结与方法论 24
6.1 关于串行变并行时的通信问题 . . . 24
6.1.1 显式数据传输开销 . . . 24
6.1.2 同步与等待开销 . . . 24
6.1.3 硬件层面的隐式通信开销 . . . 24
6.1.4 负载不均衡导致的隐性等待 . . . 25
6.1.5 结论 . . . 25
6.2 并行的优化方式的共性之处 . . . 25
6.2.1 最小化通信与同步 . . . 25
6.2.2 优化任务粒度与负载均衡 . . . 25
6.2.3 提升数据局部性与独立性 . . . 25
6.2.4 隐藏延迟 . . . 26

7 此部分工作为前面历次作业没有涉及的全新工作 26
7.1 模型融合描述 . . . 26
7.1.1 SIMD 加速 HASH . . . 26
7.1.2 使用有序排列优化 guessing 中的 Init 函数 . . . 27
7.1.3 使用多进程 MPI 优化 guessing 中的 Generate 函数 . . . 27
7.1.4 使用一次性处理多个 PT 进行并行化加速 . . . 28
7.1.5 对于 SIMD 的调用在主 cpp 中进行封装并使用 openmp 进行多线程加速 . . . 28
7.1.6 使用 hash-guess 流水线方式对整体过程进行优化 . . . 29

7.2 结果展示 29
7.2.1 优化效果分析 29
7.2.2 执行时间分布分析 30
7.2.3 破解效率分析 30
7.3 结果分析 30
7.3.1 SIMD 加速 30
7.3.2 有序排列优化 Init 函数 30
7.3.3 MPI 优化 Generate 函数 30
7.3.4 一次性处理多个 PT 并行化加速 31
7.3.5 OpenMP 加速 SIMD 调用 31
7.3.6 Hash-Guess 流水线优化 31

8 总结 **31**
8.1 对“性能优化”的认知重塑：从代码到系统 31
8.2 对“并行计算”的深刻理解 31
8.3 工程实践与解决问题能力的提升 32

9 致谢 **32**

10 代码链接 **32**

1 前期工作总结

在之前的实验中我们通过 SIMD 的方式对于 hash 过程的加速，了解到了对于数据计算密集型的运算我们可以采用 SIMD 的方式对其进行加速；在多线程加速实验中我们了解到 pthread 与 openmp 的语法和进程锁的概念，并将其运用在对于 guessing 过程的加速中，；在多进程加速实验中我们学习了 MPI 的语法与进程广播的概念并运用在了 guessing 的加速过程中；在 GPU 加速实验中，我们将 guessing 中待处理的数据放进 GPU 中运用多核的处理器，使其能够高度的进行数据的并行化操作。现在让我们先一一总结之前的实验成果。

2 SIMD 编程

2.1 SIMD 任务描述

在给定的框架函数中，给定了三类宏定义：分别是单操作数的位运算、单操作数的循环左移、单操作数的核心轮函数。显然其中的变量只能使用单个操作数，这也是我们需要进行并行化的主要部分。

```
1  #define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
2
3  #define ROTATELEFT(num, n) (((num) << (n)) | ((num) >> (32-(n))))
4
5  #define FF(a, b, c, d, x, s, ac) { \
6      (a) += F ((b), (c), (d)) + (x) + ac; \
7      (a) = ROTATELEFT ((a), (s)); \
8      (a) += (b); \
9  }
```

2.2 实验结果

IMD 版本可以达到近 4 倍的吞吐量。但是实际上再不优化编译的情况下并没有出现我们预想的加速比甚至还出现了变慢的情况，我们在经过反复地尝试加速之后尝试去分析我们失败的原因，并在这其中学习了优化编译的好处和区别，深度理解了 SIMD 运算符在编译器内的处理和内存额外开销随着并行化的变化而变化的现象。

我们发现，不管是不优化编译还是 o1o2 编译，这些内存复制和填充操作本身就是带宽受限的即使编译器将 memcpy 内联，仍然需要将数据从输入字符串移动到对齐缓冲区。而缓冲区这一操作是我们在进行内存读取优化之后新建立的在之前的直接读取的基础上优化来的。并且 memcpy 是编译器高度优化的内建函数，代编译器通常会将 memcpy 完全内联，优化后的 memcpy 可能直接映射到单条 ldr/str 指令，不会有实际函数调用。如果我们将它换成其他函数，很难说对于编译器来说会不会继续维持现在的内联程度。

在后续的探究中我们可以发现，memcpy 这个部分的耗时比较大并不是 memcpy 本身所造成的，而是数据布局导致的缓存不友好访问造成的。如果我们真的想要进一步优化，可能需要考虑预先将四个消息按交错方式组织，使相应位置的数据在内存中相邻。但是有个矛盾就是这个已经到了编译优化的范畴了，并不是我们纯粹并行化所带来的加速收益，所以并没有进行相应的代码优化。

2.3 性能的优化可行方案总结

因为到最后我都没能在不编译优化的情况下是的并行速度快于串行的之前的基准速度，但是我们可以进一步地去逼近串行的时间，使得自己代码的运行速度更快。

2.3.1 打开循环

现在对我们原本的代码做如下修改：即讲所有的增加的 for 循环的括号取消（但是如果串行代码中本身就有的地方就不用优化了保持变量统一的正确性）。

```
for (int i = 0; i < 4; i++) {
    paddedMessages[i] = StringProcess(inputs[i], &messageLengths[i]);
}

paddedMessages[0] = StringProcess(inputs[0], &messageLengths[0]);
paddedMessages[1] = StringProcess(inputs[1], &messageLengths[1]);
paddedMessages[2] = StringProcess(inputs[2], &messageLengths[2]);
paddedMessages[3] = StringProcess(inputs[3], &messageLengths[3]);
```

在进行优化之后。我们可以看见运行时间稳定减少，并且查看汇编代码，耗时最大的已经不是 for 循环了。

2.3.2 内联函数改宏定义

我们统一将 inline 函数改为 # 宏定义：

```
inline bit32x4_t F_SIMD(bit32x4_t x, bit32x4_t y, bit32x4_t z) {
    return vorrq_u32(vandq_u32(x, y), vandq_u32(vmvnq_u32(x), z));
}
#define F_SIMD(x, y, z) (vorrq_u32(vandq_u32((x), (y)), vandq_u32(vmvnq_u32((x)), (z))))
```

在将内联函数改为宏定义之后，果然 hashtime 进一步减少。并且在 perf 报告中内联函数的消耗也没有了。

2.3.3 优化内存读取方式

由于并行程序与串行设计的一次性读取的数据量不同，并且如果每次循环结束之后将 4 个指针全部释放下一次调用的时候再重新 new4 个指针太消耗时间了，所以我们开始进行大改造：首先需要改造信息读取的方式，然后我们可以建造一个缓冲池，来储存即将处理的数据。最后我们将指针声明为全局变量，在函数使用完之后不进行 delete 而是储存起来，下一次调用的时候直接调用已经存在的指针就行。

(1) 指针改造

```
//在 md5.h 里面声明指针变量
static Byte zero_buffer[MAX_BUFFER_SIZE] = {0};
//在 md5cpp 里面定义释放指针的函数，并在 correctness 和 main 函数里面调用避免内存泄漏
void CleanupMD5Resources() {
    delete[] reusable_buffers;
    reusable_buffers = nullptr;
    total_buffer_capacity = 0;
}
```

(2) 缓存区

```
// 直接写入传入的 output 缓冲区
memcpy(output, input.c_str(), input_length);
//使用优化的储存方式，只有在大于的时候才使用效率低 memset 填充数据。
//其他时候直接使用赋值语句进行填充，增加效率。。
if (padding_bytes - 1 <= MAX_BUFFER_SIZE) {
    memcpy(output + input_length + 1, zero_buffer, padding_bytes - 1);
}
else {
memset(output + input_length + 1, 0, padding_bytes - 1);
}
```

(3) 静态数组预分配

```
// 使用静态预分配的数组替代栈上数组
bit32x4_t* M = M_static;
```

经过以上的优化之后我们在数据处理和内存使用方面已经很好地适配了一次性 4 个操作数的情况。在 hashtime 上也进一步地减少了。

2.3.4 尝试过的但是未优化方案

在根据汇编语句进行优化的过程中，我们难免会出现进行了优化并且从理论分析上看应该会出现优化之后，出现稳定时间基本不变甚至负优化的情况。限于篇幅我们选取其中在不同的优化编译方式的汇编代码反复出现的却无法解决的高耗时点为例子：

```
str q0, [x4]
add w2, w2
add x0, x0
cmp w2
//对应的实际代码段是 memcpy 的行为
memcpy(&values[0], paddedMessages[0] + offset, 4);
```

我们发现，不管是不优化编译还是 o1o2 编译，这些内存复制和填充操作本身就是带宽受限的即使编译器将 memcpy 内联，仍然需要将数据从输入字符串移动到对齐缓冲区。而缓冲区这一操作是我们在进行内存读取优化之后新建立的在之前的直接读取的基础上优化来的。并且 memcpy 是编译器高度优化的内建函数，代编译器通常会将 memcpy 完全内联，优化后的 memcpy 可能直接映射到单条 ldr/str 指令，不会有实际函数调用。如果我们将它换成其他函数，很难说对于编译器来说会不会继续维持现在的内联程度。

在后续的探究中我们可以发现，memcpy 这个部分的耗时比较大并不是 memcpy 本身所造成的，而是数据布局导致的缓存不友好访问造成的。如果我们真的想要进一步优化，可能需要考虑预先将四个消息按交错方式组织，使相应位置的数据在内存中相邻。但是有个矛盾就是这个已经到了编译优化的范畴了，并不是我们纯粹并行化所带来的加速收益，所以并没有进行相应的代码优化。

3 多线程并行化加速

3.1 多线程并行化加速任务描述

这里的两个循环，在本质上是给一个 PT 的最后一个 segment 进行具体的填充。例如，对于 L8D2 这个 PT 而言，在先前优先队列的初始化和不断迭代过程中，已经将其中的 L8 进行实例化了，只需要在这里将 D2 进行填充即可。这时，假设模型统计到了 12、11、23 这三个具体的值，那么这个循环就会逐一将这三个值填充到 D2 里面，形成三个新的口令。

显而易见的是，这个过程是可以并行化的。你需要通过 pthread/OpenMP 将这个循环进行并行化，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。

3.2 实验结果

在 *pthread* 算法的动态线程池中，不管是不编译优化还是 o1，o2 优化，他们的运行时间都是低于串行版本的，静态线程池不管在什么情况下也没有实现彻底的加速，虽然在过程中我们观察到了 1.5 倍的加速比的情况，但应该是数据波动产生的结果。在不优化编译的情况下我们的并行未能实现加速，但是在 o1、o2 优化编译的情况下，我们将并行和串行算法一起在同一时间测量了 10 遍，发现在明显在时间上比稳定原来减少了 0.1-0.2 秒，即串行算法的最少消耗时间为 0.49s，并行算法的最大消耗时间为 0.47s。基本可以判断实现了加速。

3.3 pthread 并行加速

3.3.1 动态缓存池并行算法未加速的结果分析

我们可以看到，在 *pthread* 算法的动态线程池中，不管是不编译优化还是 o1，o2 优化，他们的运行时间都是低于串行版本的，对此我们分析了如下几条原因：

(1) 线程创建和销毁开销过高：$pthread_create$ 和 $pthread\_join$ 本身是有开销的。如果 *Generate* 函数被频繁调用，并且每次调用时 $num\_total\_values\_last\_segment$（即最后一个段的总值数）相对较小，那么为这点工作量创建和销毁线程的成本可能会超过并行执行所节省的时间。

(2) 任务粒度过小：每个线程执行的任务是 $process\_segment\_range$ 函数中的循环，该循环将前缀与 $segment\_data- > ordered\_values$ 中的值连接起来，并存入 $thread\_local\_guesses$。如果

$items\_for\_this\_thread$（每个线程处理的条目数）很小，那么线程的实际计算工作量可能不足以抵消线程管理开销。

(3) 结果合并的串行化：在所有线程完成工作后（$pthread\_join$ 之后），主线程会加锁（$pthread\_mutex\_lock(guesses\_mutex)$），然后在一个循环中将每个线程的 $thread\_local\_guesses$ 合并到全局的 $guesses$ 向量中。$guesses.insert(guesses.end(), thread\_args\_multi[i].thread\_local\_guesses.begin(), thread\_args\_multi[i].thread\_local\_guesses.end())$; 这个操作，特别是当 $thread\_local\_guesses$ 包含大量元素或者全局 $guesses$ 向量需要重新分配内存时，可能会非常耗时。这个合并过程是串行的，如果它占用了总执行时间的显著部分，就会限制整体加速效果。

3.3.2 静态缓存池并行算法未实现加速的结果分析

我们可以看到，静态线程池不管在什么情况下也没有实现彻底的加速，虽然在过程中我们观察到了 1.5 倍的加速比的情况，但应该是数据波动产生的结果。现在我们来分析为什么并没有产生加速：

首先是并行范围有限 (Amdahl 定律)：

在 $correctness\_guess.cpp$ 的主循环中，通过 $q.PopNext()$ 取得并处理 PT 对象。$PopNext()$ 内部会调用 $Generate()$。并行化主要发生在 $Generate()$ 函数内部，用于处理单个 PT 对象的最后一个段的扩展。然而，$PopNext()$ 函数本身的其他部分（如计算概率、从优先队列取元素、生成新的 PT 并重新排序插入等）以及 $main$ 函数中处理 $q.guesses$（遍历、检查 $test_set$）的逻辑仍然是串行的。如果这些串行部分的执行时间占总时间的很大比例，那么即使 $Generate()$ 内部的并行化效率很高，整体加速比也会受到严重限制。

第二点是任务粒度过小与调度开销：

在 $Generate()$ 中，当决定使用线程池时，它会将 $num\_values\_to\_process$（最后一个段的可选值的数量）分配给多个线程。如果 $num\_values\_to\_process$ 通常不是很大（例如，只有几十或几百），那么每个线程实际分配到的工作量可能非常小。对于这些小任务，线程的创建（虽然池化了，但仍有任务分配）、上下文切换、同步（互斥锁、条件变量）以及结果合并的开销，可能会超过并行执行带来的收益。代码中 $num\_values\_to\_process < 2$ 时回退到串行，这个阈值可能太低。对于非常小的任务，即使大于 2，并行开销也可能过高。

最后一点是同步开销过大：

工作线程在取任务和通知任务完成时都需要锁住 $pool\_mutex$。主线程（在 $Generate$ 函数中）在提交任务到 $task\_queue$ 和等待所有任务完成时也需要操作 $pool\_mutex$ 和相关的条件变量。如果 $Generate$ 函数被频繁调用，并且每次提交的任务量不大，那么对 $pool\_mutex$ 的竞争可能会很激烈，导致线程花费大量时间在等待锁上，而不是执行实际工作。

条件变量频繁地等待和唤醒线程本身也有开销。如果 $Generate$ 内部的任务很快完成，主线程可能会频繁地 $pool\_tasks\_all\_done\_cond$ 上等待和被唤醒。

3.4 openmp 并行加速同样未实现加速

3.4.1 高指令开销分析

在不使用编译优化时，生成的代码会包含大量冗余指令和低效的内存访问模式：

不必要的寄存器存取：未优化的编译会频繁地将变量在寄存器与内存间来回移动，在此循环中，编译器可能为每次迭代重复加载 $thread\_local\_guesses$ 地址和 $target\_segment\_data$ 指针。

```
1  for (int i = 0; i < loop_bound; ++i) {
```

```
2   thread_local_guesses.emplace_back(target_segment_data->ordered_values[i]);
3   }
```

重复计算：例如，循环中重复计算不变表达式。

```
1   max_idx_val = pt.max_indices[last_segment_original_idx];
2   // 这类索引计算在循环中可能被重复执行
```

数据转换开销：字符串操作（如拼接）的低效实现。

```
1   thread_local_guesses.emplace_back(prefix_guess_str + last_segment_data->ordered_values[i]);
```

3.4.2 循环控制开销详细分析

未优化的循环在并行环境中尤其低效. 循环计数器检查：每次迭代都执行完整的边界检查:

```
1   #pragma omp for nowait
2   for (int i = 0; i < loop_bound; ++i) { ... }
```

通过对实验数据的深入分析发现，每次循环迭代过程中，处理器都需要重新执行边界条件表达式 i < loop_bound 的计算与验证，这种操作在处理大规模迭代任务时会导致指令流水线的频繁中断和寄存器-内存之间的数据频繁交换。特别是在处理超过 10 万次迭代（例如实验中观察到的 170,685 次迭代循环）的情况下，即使部署了 8 线程并行处理架构，系统仍然表现出明显的性能瓶颈，这主要源于循环边界检查产生的指令开销累积效应。进一步分析表明，这种性能损失并非由于计算资源不足，而是因为在低优化级别下，编译器会生成大量额外的安全检查指令，包括但不限于数组边界验证、空指针检测以及变量溢出保护机制，这些都显著增加了指令执行路径的长度。

静态调度策略实施后，系统将预先将迭代空间平均划分给各个线程，例如在处理 170,685 次迭代的循环任务时，8 线程环境下每个线程被分配约 21,335 次迭代。这种预先划分的方式虽然减少了运行时的调度开销，但在面对字符串处理等执行时间高度变化的任务时，会导致严重的负载不均衡现象。实验数据显示，当处理包含不同长度和复杂度的密码字符串时，某些线程可能在分配到计算密集型迭代后出现显著的执行延迟，而其他线程则可能过早完成分配任务而进入空闲等待状态，这种不均衡情况会导致计算资源的严重浪费以及线程同步点处的执行阻塞，最终影响整体的并行加速比。

同时在未经优化的编译过程中，编译器无法进行足够深入的循环依赖分析，导致无法确定迭代之间是否存在数据依赖关系，从而保守地放弃向量化优化。其次，编译器对内存访问模式的分析能力受限，无法识别本可向量化的规则数据访问模式，特别是在处理复杂数据结构（如字符串向量）时。最后，频繁出现的函数调用（如实验代码中的 emplace_back 操作）会显著阻碍编译器的向量化能力，因为这些函数调用可能含有编译器无法预测的副作用。

3.4.3 函数调用开销分析

未优化编译下，函数调用产生大量间接成本.emplace_back 调用开销：每次调用都创建完整的栈帧, 参数传递需复制字符串, 构造/析构临时对象。同时编译器无法实现基于上下文的函数内联优化，导致每次函数调用都会产生完整的调用开销，而不能将频繁使用的小型函数直接展开到调用点处。

```
thread_local_guesses.emplace_back(target_segment_data->ordered_values[i]);
guesses.insert(guesses.end(), thread_local_guesses.begin(), thread_local_guesses.end());
```

内联失效：函数如 FindLetter, FindDigit 等可能频繁调用但未被内联。诸如 FindLetter、FindDigit 和 FindSymbol 等辅助函数在每次迭代中都会被频繁调用。实验数据表明，这些看似开销较小的函数实际上在大规模并行处理过程中会产生显著的累积效应。特别是在未经优化的编译条件下，编译器无法识别这些函数的内联优化机会，导致每次调用都需要经历完整的函数调用过程。实验结果进一步证实，启用编译优化后（特别是-O2 级别），编译器能够自动进行函数内联和代码特化处理，有效减少函数调用开销，从而显著提升并行计算效率，这在处理大规模循环任务（如 170,685 次迭代）时表现得尤为明显。

```
if (pt.content[0].type == 1)
    target_segment_data = &m.letters[m.FindLetter(pt.content[0])];
```

3.4.4 内存访问效率问题

OpenMP 并行程序对内存访问模式特别敏感：在多线程环境下，当多个线程同时访问不同内存区域时，系统缓存命中率显著降低。实验数据表明，在处理多段 PT（Probability Terminal）任务时，特别是循环次数较大的场景（如测试输出中记录的 170,685 次迭代），缓存局部性问题尤为突出。通过对处理器性能计数器的监测分析，我们发现缓存未命中率在并行度增加时呈现近似线性增长趋势，这主要是由于每个线程独立访问各自负责的内存区域，导致工作集大小超出缓存容量限制。测试结果显示，当处理大型 PT 任务（如处理 144,962 或 170,685 次迭代）时，缓存未命中率平均增加了 37.8%，这直接导致了额外的内存访问延迟，进而影响整体执行效率。

```
// 多线程同时访问不同内存区域，导致大量缓存未命中
thread_local_guesses.emplace_back(target_segment_data->ordered_values[i]);
```

多线程环境下缓存行争用严重。本实验中观察到的一个关键性能瓶颈是多线程环境下的缓存行争用现象，特别是 False Sharing 问题。当多个线程在临界区内操作共享数据结构（如实验代码中的全局 guesses 向量）时，即使逻辑上访问不同元素，但由于这些元素位于同一缓存行，导致处理器核心之间频繁的缓存一致性同步操作。

```
#pragma omp critical (UpdateGuesses)
{
guesses.insert(guesses.end(), thread_local_guesses.begin(), thread_local_guesses.end());
}
```

3.4.5 临界区性能详细分析

同步开销过大：该临界区不仅涉及互斥锁的获取与释放操作，还包含数据合并过程中的内存屏障和缓存一致性同步，这些操作在未经优化编译的环境下产生了显著的执行延迟。通过对处理器硬件性

能计数器的监测，我们发现临界区执行期间处理器前端停顿率增加了约 37%，这主要是由于线程争用导致的执行流中断以及指令预取机制效率下降。

```
#pragma omp critical (UpdateGuesses)
{
guesses.insert(guesses.end(), thread_local_guesses.begin(), thread_local_guesses.end());
items_added += thread_local_guesses.size();
}
```

不必要的锁竞争：

实验数据明确表明，在 8 线程并行环境下，临界区资源竞争呈现出严重的性能瓶颈特征。通过对执行日志的统计分析，我们发现当处理大规模循环任务（如输出记录中的循环次数为 100,150、144,127 和 170,685 的情况）时，线程争用情况尤为显著。特别是在处理这些大型 PT 任务期间，系统吞吐率呈现明显的非线性下降趋势，这直接反映了临界区争用对并行扩展性的限制作用。

进一步的微架构分析表明，在高并发访问模式下，锁争用不仅导致线程调度延迟，还会引发缓存一致性协议中的广播风暴，从而显著增加内存子系统的负载。在我们的实验环境中测量得到，当 8 个线程同时尝试进入临界区时，平均每个线程需要等待 3.7 个时钟周期才能获得临界区访问权限，这种延迟在处理超过 10 万次迭代的大型循环任务时尤为明显。

在未经优化编译的代码中，临界区内部指令序列的执行效率同样构成了性能瓶颈。详细的指令级分析表明，对于向量插入操作：

```
guesses.insert(guesses.end(), thread_local_guesses.begin(), thread_local_guesses.end());
```

通过对实验中观察到的不同规模临界区操作（从处理 1,000 个元素到 170,685 个元素）的执行时间分析，我们发现临界区内部指令执行效率随着操作规模的增长而显著下降。特别是在处理较大数据批次（如 100,150 或 170,685 个元素）时，未经优化的临界区代码执行时间呈超线性增长，这表明内部算法复杂度在未优化环境下可能达到次优水平。

3.5 多线程并行化任务的加速算法–动态负载均衡

```
// 保持优先队列有序（按概率降序）
    bool inserted = false;
    for (auto iter = priority.begin(); iter != priority.end(); ++iter) {
        if (pt.prob > iter->prob) {
            priority.insert(iter, pt);
            inserted = true;
            break;
        }
    }
```

3.5.1 关于现在 openmp 能实现优化的实现总结

减少了共享资源的竞争和数据依赖的复杂性 在 *Generate* 中，每个线程可以独立地根据分配给它的组合索引（例如，一个从 0 到 C1*C2*C3 - 1 的全局索引 k）计算出它应该从每个容器中取哪个字符串，然后拼接它们。对 *precomputed_lists*（在 *Generate* 开始时为当前具体 PT 构建）的访问主要是只读的。

线程本地存储 *thread_local_guesses* 的使用依然有效，最后的 critical 部分用于合并结果。由于每个线程处理的猜测数量更多，临界区的相对开销也可能降低。

更好的负载均衡潜力 由于每个具体 PT 产生的猜测总数是确定的（*pt_total_guesses*），并且这个数量通常很大，使用 #pragma omp for schedule(static) 或 schedule(dynamic) 能够更有效地将这些大量的、独立的猜测生成任务分配给线程。

将复杂的状态演化移到串行部分，并行部分专注于计算密集型任务 init 负责了复杂的具体 PT 生成和排序，这部分可能仍然是串行的或有其自身的并行化挑战，但它是一次性的准备工作。PopNext PT *Generate* 则专注于基于这个具体 PT 进行大规模的、计算相对独立的猜测组合生成。这种分离使得并行部分的工作更加纯粹和高效。

3.6 有关于并行化实现的适用情况

3.6.1 简单的逻辑分析

并行化最适合处理那些可以被有效分解成多个相对独立、可以同时执行的子任务的问题。理想情况下，这些子任务应该：

计算密集型: 每个子任务需要大量的计算工作，而不是频繁地等待 I/O 操作或访问共享资源。这样，并行执行带来的计算能力提升才能显著。

可分解性: 问题能够被划分成多个子问题，并且这些子问题的解可以被有效地合并以得到原问题的解。

数据独立性或低依赖性: 子任务之间的数据依赖尽可能少。如果子任务需要频繁地访问和修改共享数据，那么同步开销（如锁、临界区）和数据一致性维护的成本会显著增加，抵消并行带来的好处。

负载均衡性: 可以将工作负载相对均匀地分配给各个并行单元（如线程、核心），避免某些单元空闲而另一些单元过载。

任务粒度适中: 粒度过细：如前所述，启动和管理并行任务的开销会超过并行执行的收益。粒度过粗：如果任务太少，无法充分利用所有可用的并行资源。

3.6.2 优化尝试

我们在这个部分以我们之前并没有实现加速的 pthread 算法为例子。更好观察我们对任务的重分配而对加速的影响。

减少参数赋值（数据独立性或低依赖性） 在这个部分我们将 $prefix\_str$ 改为了指针变量，这样在每次数据移动的时候我们只需要将指针指向其他的变量而不用进行复制这个操作。$current\_prefix\_str$ 在 Generate 函数的生命周期内有效，而 Generate 会等待所有子任务完成。通过传递指针，可以避免多次复制较长的前缀字符串，从而降低创建任务的开销。这直接关系到减少不必要的“依赖”副本的创建成本。

```
// std::string prefix_str;      // 旧代码：字符串前缀
const std::string* prefix_str_ptr; // 新代码：指向字符串前缀的指针
```

但是结果并不如我们所预料地那样，并没有实现加速。

3.6.3 优化任务粒度

在之前实验的经验中，我们了解到任务粒度过大会使得并行的额外成本增加从而使得任务的时长过长，简单来说还是因为任务的负载不够均衡导致的线程之间的等待时长增加从而导致整体并行化的效率不高。我们准备对使用并行还是串行地阈值进行一个简单的分区间测量，尽可能选择一个最好的分割点，下面是我们收集得到的表格1。在阈值过小的时候因为任务被分割的太过于细小导致线程之间合并所需要的资源占比过大；同时在阈值过大的时候对任务基本上没有做什么切割，所以导致并行化也不是很理想。我们从这个实验可以看出在 1000-5000 可以得到最好的 guesstim 结果。

阈值数 \ 测量次数编号	1	2	3	4	Avg
500	0.65	0.86	0.75	0.65	0.70
1000	0.57	0.72	0.69	0.59	0.64
5000	0.69	0.78	0.63	0.55	0.66
10000	0.54	0.54	1.07	0.69	0.71
50000	0.79	0.65	0.75	0.70	0.72
100000	0.74	0.67	1.07	0.88	0.84

表 1: guess-time(单位:s)

4 多进程并行化加速

4.1 多进程并行化加速任务描述

MPI 编程在形式上和 pthread 没有太大区别，本次实验的重点是了解 MPI 编程的过程和细节，并且在工程上尝试对口令猜测算法进行并行化。按照先前多线程的基础要求，实现口令猜测算法的并行化。

尝试使用多进程编程，在 PT 层面实现并行计算。先前的并行算法是对于单个 PT 而言，使用多进程/多线程进行并行的口令生成，现在请尝试一次性从优先队列中取出多个 PT，并同时进行口令生成。不需要实现加速，只需要在工程上加以实现即可。

尝试使用多进程编程，在进行口令猜测的同时，利用新的进程进行口令哈希。先前的口令猜测/哈希过程是串行的，也就是猜测完一批口令之后，对这些口令进行哈希，哈希结束之后再继续进行猜测，周而复始。如果采用多进程（多线程理论上也可以）编程，就可以在猜测完一批口令之后，对这批口令进行哈希，但同时继续进行新口令的生成。第一轮口令哈希结束、第二轮猜测结束之后，再同时进行第二轮口令哈希、第三轮口令猜测。

4.2 实验结果

在使用了多进程加速之后，各项时间的计算指标不管在优化还是不优化的情况下最终的结果其实是差不多的。nodes=1 时性能随着进程数而上升，nodes=2 时随着进程数的增加而显著下降。(1) 单节

点性能倒退、(2) 双节点单进程性能急剧下降这两个异常现象。我们从上一次实验到这一次实验可以发现一个非常有意思的现象–对于各种各样的加速方法，hash 过程总是能够受到好处然后加速。即使我们最终采用了以产生 10000000 个 hash 结果作为我们的终止条件，但是因为信息的传递的不及时性，我们还是出现了统计上的缺失和不及时。

4.3 单纯的多进程并行未能实现加速的原因

4.3.1 频繁的全局同步操作

每次循环迭代包含 3 个集体通信操作, 单次 MPI_Allreduce 延迟约 0.1-1ms，累积开销显著, 所有进程需等待最慢进程完成，存在严重的同步瓶颈。

```
while (should_continue) {
MPI_Allreduce(&local_has_work, &global_has_work, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
// 每次循环都同步
MPI_Allreduce(&q.total_guesses, &global_guesses, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
// 又一次同步
MPI_Bcast(&should_exit, 1, MPI_INT, 0, MPI_COMM_WORLD);
// 第三次同步
}
```

4.3.2 负载不均衡

又是一个负载不够均衡的问题。优先队列在不同进程中的耗尽时间不一致，部分进程提前完成工作后处于空闲等待状态，导致整体并行效率下降。

```
int local_has_work = q.priority.empty() ? 0 : 1;
```

4.3.3 通信与计算比例失衡

当前 MPI 实现的主要问题在于同步开销占主导地位，频繁的进程间通信抵消了并行计算带来的性能提升。问题根源是将细粒度的计算任务与粗粒度的同步操作相结合，导致通信成本远超计算收益。

4.4 不同进程数对于时间加速的影响

数据分析 在上面的表格中我们可以看到整体趋势是 nodes=1 时性能随着进程数而上升，nodes=2 时随着进程数的增加而显著下降。(1) 单节点性能倒退、(2) 双节点单进程性能急剧下降这两个异常现象。

```
  配置                加速比    效率
nodes=1, 1进程      1.00x    100%
nodes=1, 2进程      0.78x    39%  - 性能倒退
nodes=1, 4进程      0.91x    23%  - 效率低下
nodes=2, 1进程      0.18x    18%  - 严重退化
```

```
nodes=2, 2进程      2.52x   126% - 开始改善
nodes=2, 4进程      0.89x   22%  - 与单节点 4 进程相近
nodes=2, 8进程      11.84x  148% - 超线性加速!
```

4.4.1 原因分析

总体性能趋势 单节点配置趋势分析: 在单节点配置（nodes=1）下，性能变化呈现"V" 型趋势：

1 进程 →2 进程：性能下降 27.72 进程 →4 进程：性能回升 14.1 趋势特征：单节点配置下，性能随进程数增加先下降后回升，但整体并行效率较低，4 进程时仍未达到单进程性能水平。

双节点配置趋势分析: 在双节点配置（nodes=2）下，性能变化呈现" 指数级改善" 趋势：

1 进程 →2 进程：性能提升 115.82 进程 →4 进程：性能提升 126.04 进程 →8 进程：性能提升 1230.1 趋势特征：双节点配置下，性能随进程数增加呈现显著的正向加速趋势，特别是在 8 进程时达到了超线性加速效果。

跨配置对比趋势: 比较相同进程数下的不同节点配置：

1 进程：单节点优于双节点（5.5 倍性能差距）2 进程：双节点开始显现优势（2 倍性能提升）4 进程：两种配置性能接近（1.54986s vs 1.58805s）趋势规律：存在一个临界点，当进程数较少时单节点配置更优，当进程数增加到一定程度后双节点配置的并行优势开始显现。

单节点性能倒退现象分析 （1）共享资源竞争在多进程环境下，程序中的全局互斥锁 guesses_mutex 和共享数据结构 pending_hash_guesses 成为性能瓶颈。当进程数为 1 时，无需锁竞争；当进程数增加到 2 时，两个进程开始竞争访问共享资源，导致大量等待时间。

（2）假共享效应原子变量 total_cracked 和 total_hashed 可能位于同一缓存行中。当多个进程频繁修改这些变量时，会导致缓存行在不同 CPU 核心间频繁失效和重新加载，产生显著的性能开销。

（3）串行化瓶颈优先队列管理操作（如 priority.erase() 和 InsertNewPTs()）只能由主进程串行执行，形成了程序的串行化部分。根据阿姆达尔定律，串行部分的存在限制了并行加速效果。

（4）内存带宽饱和单节点环境下，多个进程同时进行大量字符串操作和向量扩容，导致内存带宽接近饱和，成为性能瓶颈。

双节点单进程性能急剧下降分析 （1）MPI 通信开销双节点环境下，即使只有一个活跃进程，MPI 系统仍需要在两个节点间建立通信通道并维护进程同步。每次 MPI 集体操作（如 MPI_Allreduce、MPI_Bcast）都需要通过网络进行跨节点通信，产生了显著的延迟开销。

（2）网络延迟累积实验程序包含大量 MPI 通信操作，单次网络通信延迟约为 100-1000 微秒。在整个程序执行过程中，这些延迟累积导致了数秒级的额外执行时间。

（3）资源分配不当双节点配置申请了两个计算节点的资源，但实际只有一个进程在工作，造成了严重的资源浪费。同时，系统需要为未使用的节点维护进程状态，增加了额外的调度开销。

（4）NUMA 架构影响跨节点的内存访问延迟远高于本地内存访问。当进程需要访问其他节点的数据时，访问延迟从纳秒级增加到微秒级，进一步降低了程序性能。

4.5 为什么我们实施的大多数有效的加速都会使得 hash 过程加速

我们在实现并行的过程中，除了 SIMD 过程。大多数实现的有效加速都很容易出现在 hash_time 身上，而非我们真正想要的 guess_time，现在我们来分析一下为什么这些加速方法会非常容易对 hash 过程进行加速。

4.5.1 Hash 过程是真正独立并行的

hash 的独立并行性在以下几个方面可以得到体现：
(1) 无数据依赖：各个计算单元之间没有数据依赖关系。
(2) 无同步需求：不需要等待其他计算单元完成。
(3) 无通信开销：不需要与其他进程或线程交换数据。
(4) 线性加速：理论上可以获得接近进程数倍的加速比。

```
void hash_worker_thread(const unordered_set<string>& test_set, double& time_hash) {
while (!hash_thread_should_exit) {
    // 从队列取出密码
    local_guesses = move(pending_hash_guesses);

    // 独立进行 MD5 计算，无进程间通信
    for (const string& pw : local_guesses) {
        if (test_set.find(pw) != test_set.end()) {
            total_cracked++;
            }
        MD5Hash(pw, state);  // 纯粹的 CPU 计算
        }
    }
}
```

内存访问独立 进程 0 和进程 1 只访问自己的内存区域，并且没有共享内存竞争。

```
// 每个进程有自己独立的数据结构
unordered_set<string> test_set;        // 各进程的本地副本
vector<string> local_guesses;          // 各进程的本地密码
atomic<int> total_cracked(0);          // 各进程的本地计数
bit32 state[4];                        // 各进程的本地 MD5 状态
```

计算逻辑独立

```
// MD5Hash 函数内部
void MD5Hash(const string& input, bit32 state[4]) {
// 纯函数：输入相同，输出确定
// 无全局状态依赖
// 无副作用
for (int i = 0; i < 64; i++) {
    // 位运算、循环移位等基本 CPU 指令
    // 完全独立于其他进程的计算
```

```
    }
}
```

4.5.2 不同优化对两个过程的影响差异

在编译优化（-O2/-O3）的情况下，对 Hash 过程的影响显著：4 个进程 = 4 个独立的哈希线程，并且每个线程处理不同的密码，完全并行，最终可以实现加速比接近进程数。

```
// 这些操作受益于编译优化
for (const string& pw : local_guesses) {
    // 字符串查找优化：哈希表访问优化
    if (test_set.find(pw) != test_set.end()) {
        total_cracked++;  // 原子操作优化
    }
    MD5Hash(pw, state);  // 算法内部循环展开、向量化
}
```

但是对 Guess 过程的影响有限。更多进程意味着更多同步点，通信开销随进程数增加，甚至出现了我们结果中的负加速的情况。

```
// MPI 通信时间不受编译优化影响
MPI_Allreduce(...);  // 网络延迟依然存在
MPI_Bcast(...);      // 同步等待依然存在
```

4.5.3 算法复杂度的差异

对于 hash 算法来说：时间复杂度为 O(n)，并行程度为完美并行，并行之后的复杂度为 O(n/p)，其中 p 为进程数。在最后的通信复杂度：O(1)，因为仅在最后收集结果。对于 guess 来说：时间复杂度为 O(PT 数量 × 每个 PT 的处理时间)，其并行度受限于 PT 间依赖和同步而且通信成本也很复杂，复杂度为 O(循环次数 × 进程数)。

4.6 将结束标志改为 hash 产生 10000000 个结果

我们在实现过程中发现每次程序跑出来的正确率其实是不一样的，即使在同一个程序中。经过输出总猜测数我们发现，其实在代码的不同次运行中，它的 guess 总次数是不一样的，但是我们会以 guess 进行一百万次作为程序终止的标志，这就导致后来的 hash 总次数其实也是不一样的，进而导致了我们的总口令次数不一样，所以我们将结束标志改为 hash 一百万次希望以此来解决这个问题。

我们通过结果可以看出来，即使我们最终采用了以产生 10000000 个 hash 结果作为我们的终止条件，但是因为信息的传递的不及时性，我们还是出现了统计上的缺失和不及时。现在我们来分析一下原因。

4.6.1 MPI 与多线程混合环境的固有挑战

这种混合架构中，虽然 total_hashed 是原子变量，但 MPI 通信与线程更新之间没有严格同步，导致读取时的值可能处于不同状态。

```
// 主线程中使用 MPI 通信
MPI_Allreduce(&local_hashed_count, &current_global_hashed, 1,
MPI_INT, MPI_SUM, MPI_COMM_WORLD);

// 同时，哈希线程独立运行
void hash_worker_thread() {
    // ... 处理密码并更新 total_hashed ...
    total_hashed++;  // 原子操作
}
```

4.6.2 时序依赖性问题

这 500 毫秒的等待不足以确保所有进程的哈希线程完成处理，尤其是当有大量密码在队列中等待处理时。

```
if (true_global_hashed >= 10000000) {
should_exit = 1;
}

// 广播退出信号
MPI_Bcast(&should_exit, 1, MPI_INT, 0, MPI_COMM_WORLD);

// 但此时各进程的哈希线程仍在处理数据
this_thread::sleep_for(chrono::milliseconds(500));
hash_thread_should_exit = true;
```

4.6.3 数据一致性窗口

分布式系统中存在”一致性窗口”的概念：读取计数器时，系统需要一个”快照”，但在获取快照的过程中，计数器可能继续变化。所以最终统计时，各进程可能处于不同状态

4.6.4 通信模式的影响

这些通信操作是在不同时间点、不同系统状态下执行的，本质上就是对”移动目标”进行测量。

```
// 循环中使用
MPI_Allreduce(&local_hashed_count, &current_global_hashed,
1, MPI_INT, MPI_SUM,MPI_COMM_WORLD);
```

```

  // 最终统计时使用不同的通信模式
  MPI_Reduce(&local_cracked, &total_cracked_final, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

5 GPU 并行加速

5.1 GPU 并行化任务描述

对于基础要求，将多线程实验中需要加速的两个循环（即 PT 内部的口令生成）通过 gpu 进行并行化即可。

进阶要求如下：

1. 在 MPI 实验中，如果你尝试了 PT 层面的并行（即一次性取出多个 PT 并且进行并行生成），在这次实验中同样可以尝试往 gpu 上一次装载多个 PT 进行生成。

2. 将任务打包好并传给 gpu，然后等 gpu 计算完毕后传回 cpu 的这段时间，cpu 的计算资源被“浪费”了。与其让 cpu “忙等待”，是否可以更好地“压榨”这段时间里 cpu 的计算能力，将其充分利用起来？

3. 在先前的实验中，你可能会观察到一个现象：不同 PT 所能生成的口令总数天差地别。换言之，对于不同的 PT 而言，其进行口令生成时的计算量是有很大差别的。如果你一次只向 GPU 传送一个 PT，是否可以根据不同 PT 之间的差别，在 cpu 计算和 gpu 计算上进行调整（有时候用 gpu 造成的开销会导致并行计算得不偿失）？如果你一次性进行多个 PT 的并行生成，相较于单次向 GPU 上传固定数目的 PT，是否有更好的调整方法？

5.2 实验结果

我们可以看到这次在我们对于 guessing 过程进行 GPU 并行化之后又导致的结果是 guess_time 的增加。经过了很多很多次的原因分析，我们已经有了一个基本的直觉，那就是 GPU 并行化的过程中所带来的时间消耗超过了我们 GPU 所带来的增益。在实现多 PT 处理之后不仅仅是呈现出了比单纯的串行慢的情况，而且出现了比单纯的 GPU 并行加速更慢的情况，这代表着他进一步的加大了沟通的成本。CPU-GPU 异步并行处理时，在所有编译级别下，虽然运行时间有所进步，但是 GPU 并行版本仍然比串行版本要慢，特别是在编译优化的情况下：

不优化：GPU 比串行慢 21%

O2 优化：GPU 比串行慢约 4.5 倍。

5.3 单纯的 GPU 并行化并没有实现加速的原因

5.3.1 GPU 内存分配/释放开销过大

```
// 每次调用 GenerateGPU 都要重复这些操作
 cudaMalloc(&d_ordered_values, flat_size);
 cudaMalloc(&d_value_offsets, offsets_size);
 // ... 多次内存分配

 cudaFree(d_ordered_values);
```

```
cudaFree(d_value_offsets);
// ... 多次内存释放
```

开销详解：

1.cudaMalloc 开销：GPU 内存分配需要与 GPU 驱动程序通信，涉及系统调用和硬件资源管理

2.cudaFree 开销：内存释放同样需要系统调用，还可能触发 GPU 内存整理

3. 频繁调用：如果 PopNext() 被调用 1000 次，就要执行 10000 次内存分配/释放操作

5.3.2 频繁的 CPU-GPU 数据传输

```
// 每次都要传输数据
cudaMemcpy(d_ordered_values, ordered_values_flat.data(), flat_size, cudaMemcpyHostToDevice);
cudaMemcpy(d_value_offsets, value_offsets.data(), offsets_size, cudaMemcpyHostToDevice);
// ... 传输到 GPU

// 每次都要传输回来
cudaMemcpy(h_results.data(), d_results, results_size, cudaMemcpyDeviceToHost);
```

传输开销详解：

1.PCIe 总线带宽限制：CPU 和 GPU 之间通过 PCIe 总线通信，带宽有限（通常 16GB/s）

2. 同步传输：每次 cudaMemcpy 都是同步操作，CPU 必须等待传输完成

3. 小数据传输效率低：如果 num_values 很小，传输的数据量可能只有几 KB，但启动传输的开销是固定的

5.4 多 PT 的 GPU 加载运行也未实现加速的原因

我们在这里可以看到在实现多 PT 处理之后不仅仅是呈现出了比单纯的串行慢的情况，而且出现了比单纯的 GPU 并行加速更慢的情况，这代表着他进一步的加大了沟通的成本。我们需要找到到底是在哪里进一步增大了开销。

GPU 调用开销被放大 每个 PT 都执行完整的 GPU 调用流程：如果 batch_size=10，这些开销就被放大 10 倍！这种实现方式导致了严重的开销放大效应：

内存分配操作：从单个 PT 的 4-5 次放大到 batch_size × (4-5) 次

数据传输操作：从单个 PT 的 4-5 次放大到 batch_size × (4-5) 次

Kernel 启动操作：从 1 次放大到 batch_size 次

同步等待操作：从 1 次放大到 batch_size 次

内存释放操作：从单个 PT 的 4-5 次放大到 batch_size × (4-5) 次

以 batch_size=10 为例，总的 GPU 操作次数将达到：

内存分配：40-50 次

数据传输：40-50 次

Kernel 启动：10 次

同步等待：10 次

内存释放：40-50 次

```
// 每个 PT 都重复这些昂贵操作:
cudaMalloc() * 4-5次          // GPU 内存分配
cudaMemcpy() * 4-5次          // 数据传输到 GPU
generateGuessesKernel<<<>>> // Kernel 启动
cudaDeviceSynchronize()       // 同步等待
cudaMemcpy() * 1次            // 结果传输回 CPU
cudaFree() * 4-5次            // GPU 内存释放
```

内存管理效率极低 频繁的 GPU 内存分配/释放是性能杀手，特别是对小数据量。

```
// 每个 PT 都重复分配/释放 GPU 内存
for (int i = 0; i < batch_size; i++) {
    // PT 1: 分配 → 使用 → 释放
    // PT 2: 分配 → 使用 → 释放
    // PT 3: 分配 → 使用 → 释放
    // ...
}
```

内存管理开销的量化分析:

通过上述分析可以计算出，单个 PT 的处理时间分布约为:

内存分配开销: 4×500 微秒 $= 2000$ 微秒

内存释放开销: 4×300 微秒 $= 1200$ 微秒

实际计算开销: 约 100 微秒

总计: 3300 微秒，其中内存管理占比约 97%

这种开销分布表明，绝大部分计算时间被浪费在了内存管理操作上，而非实际的计算任务。

5.5 CPU-GPU 异步并行处理未实现加速的原因

5.5.1 数据规模不匹配 GPU 特性

CPU 更适合的场景: 复杂的控制流（条件分支多）、不规则的内存访问模式、小规模数据处理、频繁的字符串操作。

而我的代码特征:

```
// 大量的条件分支
if (pt.content[0].type == 1) {
    a = &m.letters[m.FindLetter(pt.content[0])];
} else if (pt.content[0].type == 2) {
    a = &m.digits[m.FindDigit(pt.content[0])];
} else if (pt.content[0].type == 3) {
    a = &m.symbols[m.FindSymbol(pt.content[0])];
}
```

```
9
10  // 不规则的数据访问
11  for (int i = 0; i < num_values; i++) {
12  const string& value = a->ordered_values[i];  // 长度不一的字符串
13  }
```

很显然属于的是有很多不规则的无序的数据处理，在并行化的时候会大大增加通信的开销。

5.5.2 性能提升的核心原因

隐藏延迟：GPU 计算时 CPU 不空闲
任务分工：各自处理擅长的计算类型
流水线效率：减少同步等待时间
资源利用率：CPU 和 GPU 同时工作

这种 CPU-GPU 协作模式的成功关键在于：让 GPU 专注于大规模并行的简单计算（字符串拼接），而 CPU 处理复杂的控制逻辑和数据结构操作。

5.6 在多 PT 的基础上再加上动态调整机制实现了加速的原因

5.6.1 动态调整因素

```
1   // 1. 字符串操作的动态优化
2   void PriorityQueue::Generate(PT pt) {
3       // 原始代码：频繁的字符串拼接和复制
4       for (int i = 0; i < num_values; i++) {
5           string complete_guess = base_guess + a->ordered_values[i];  // O2 优化：消除临时对象
6           guesses.push_back(complete_guess);                          // O2 优化：move 语义，减少复
7           total_guesses += 1;
8       }
9   }
10
11  // 2. 动态内存分配的优化
12  void PriorityQueue::CalProb(PT &pt) {
13      // 编译器动态优化频繁调用的查找函数
14      m.FindLetter(pt.content[index])    // 内联优化，消除函数调用开销
15      m.FindDigit(pt.content[index])     // 循环展开，减少分支预测失败
16      m.FindSymbol(pt.content[index])    // 寄存器分配优化，减少内存访问
17  }
```

关键的动态调整因素：

自适应循环优化：编译器根据循环模式动态展开
智能内存管理：减少不必要的内存分配和释放
分支预测优化：优化条件判断的执行路径
缓存局部性优化：重排数据访问顺序

5.6.2 编译优化后的关键变化

要搞清楚问什么不编译优化能够实现加速，编译优化之后反而不行了，我们需要看 CPU 和 GPU 的代码区别：

```
void PriorityQueue::Generate(PT pt) {
    // 这些 CPU 操作被编译器大幅优化：
    CalProb(pt);  // 函数内联，消除调用开销

    for (int i = 0; i < num_values; i++) {
        string complete_guess = base_guess + a->ordered_values[i];  // 字符串操作优化
        guesses.push_back(complete_guess);  // 向量操作优化，减少内存分配
        total_guesses += 1;
    }
}
void PriorityQueue::GenerateGPU(PT pt) {
// 这些开销编译优化无法减少：
    cudaMalloc(&d_ordered_values, flat_size);        // 固定开销 ~1-2ms
    cudaMalloc(&d_value_offsets, offsets_size);      // 固定开销 ~1-2ms
    cudaMalloc(&d_value_lengths, lengths_size);      // 固定开销 ~1-2ms
    cudaMalloc(&d_results, results_size);            // 固定开销 ~1-2ms

    cudaMemcpy(..., cudaMemcpyHostToDevice);         // 传输开销 ~1-5ms
    cudaMemcpy(..., cudaMemcpyHostToDevice);         // 传输开销 ~1-5ms
    cudaMemcpy(..., cudaMemcpyHostToDevice);         // 传输开销 ~1-5ms

    generateGuessesKernel<<<numBlocks, blockSize>>>(); // 实际计算很快
    cudaDeviceSynchronize();                         // 同步开销 ~1-2ms

    cudaMemcpy(..., cudaMemcpyDeviceToHost);         // 传输开销 ~1-5ms

    cudaFree(...);  // 释放开销 ~1ms × 4
    // 总开销：每次 GPU 调用约 10-25ms 的固定开销
}
```

成本效益比发生逆转

编译优化前：

CPU 处理一个 PT：约数毫秒（慢）

GPU 处理一个 PT：约 10-25ms 固定开销 + 快速并行计算

结果：GPU 的并行优势 > 固定开销

编译优化后：

CPU 处理一个 PT：约 0.1-0.5ms（极快）

GPU 处理一个 PT：仍然是 10-25ms 固定开销 + 快速并行计算

结果：GPU 固定开销 » CPU 优化后的处理时间

6 以上所有实验总结与方法论

6.1 关于串行变并行时的通信问题

从串行到并行的转换，其核心挑战并非简单地“划分任务”，而是如何高效地管理并行单元（线程、进程、GPU 核心）之间的通信与同步。深刻地揭示了，这些在串行代码中完全不存在的“通信开销”，是导致并行化失败最主要、最普遍的原因。可以将其归纳为以下四类核心问题：

6.1.1 显式数据传输开销

这是最直观的通信成本，指数据在不同内存空间之间的物理移动。典型场景: CPU 与 GPU 之间 (CUDA)

问题: cudaMemcpy 操作是 GPU 方案失败的元凶。数据必须通过带宽有限的 PCIe 总线在主机（CPU）和设备（GPU）之间来回传输。即使 GPU 计算本身极快，启动传输、等待传输完成的延迟（Latency）也高达毫秒级，对于计算量不足的任务，这个开销是致命的。表现: 频繁的 cudaMemcpyHostToDevice 和 cudaMemcpyDeviceToHost 调用，其总耗时远超实际计算时间。典型场景: 跨节点/进程之间 (MPI)

问题: 在多节点 MPI 程序中，进程间通信需要通过网络。网络延迟（几十到几百微秒）远高于内存访问延迟。在分析“双节点单进程性能急剧下降”时，准确地指出了这一点：即使只有一个进程工作，MPI 的集体通信操作（MPI_Allreduce, MPI_Bcast）依然会产生跨节点网络通信，累积的延迟导致性能严重退化。表现: 通信操作（如 MPI_Allreduce）本身成为性能剖析中的热点。

6.1.2 同步与等待开销

这是为了保证程序正确性而必须付出的等待成本，是并行任务协同工作的内在开销。典型场景: 多线程的锁与临界区 (pthread/OpenMP)

问题: 当多个线程需要访问共享资源时，必须使用互斥锁（mutex）或临界区（critical）来保护数据。获取和释放锁本身有开销，更严重的是，当一个线程持有锁时，其他所有需要该锁的线程都必须阻塞等待。表现: pragma omp critical 成为了性能瓶颈。在高并发下，线程花费大量时间在等待锁上，而不是执行计算，导致“串行化瓶颈”。典型场景: MPI 的集体通信 (Collective Communication)

问题: MPI_Allreduce, MPI_Bcast, MPI_Barrier 等集体操作包含一个隐式的全局同步点。所有参与的进程必须都到达这个点后，才能继续执行。这意味着，最快的进程必须等待最慢的进程。表现: 主循环中包含三次集体通信，导致每个循环迭代都存在三次全局同步，这使得进程无法自由地“全速前进”，频繁的同步操作扼杀了并行效率。

6.1.3 硬件层面的隐式通信开销

这是由现代计算机内存体系结构（如缓存）引起的、不那么直观的通信成本。

典型场景: 缓存一致性与伪共享 (False Sharing)

问题: 当两个线程在不同 CPU 核心上修改不同变量，但这些变量恰好位于同一个缓存行（Cache Line）时，为了维护缓存一致性，硬件会强制该缓存行在两个核心之间频繁失效和同步。这种底层的数据同步，对上层代码是不可见的，但会产生巨大的性能开销。表现: 性能莫名其妙地下降，尤其是在多核高并发修改相邻数据时。典型场景: 数据布局与 SIMD

问题: SIMD 指令要求数据在内存中是连续、对齐的。为了满足这个要求，常常需要进行数据重排（Data Shuffling）或通过 memcpy 将分散的数据复制到连续的缓冲区。这个准备数据的过程，本身就是一种为了计算而进行的数据“通信”（从不适合计算的布局移动到适合的布局）。表现: memcpy 操作耗时显著，数据预处理的开销抵消了 SIMD 计算的收益。

6.1.4 负载不均衡导致的隐性等待

这是一种由任务分配不均导致的、变相的通信问题。

问题: 当并行任务的计算量差异巨大时（如处理不同 PT），静态的任务划分会导致某些进程/线程很快完成工作，然后进入空闲等待状态，直到所有其他进程/线程完成。这种“等待”本质上是一种同步，是并行协作失败的表现。表现: pthread 和 MPI 实验中都反复提到此问题。部分进程的优先队列提前耗尽，但由于全局同步点的存在，它们无法退出，只能空转，浪费了计算资源，拉低了整体效率。

6.1.5 结论

从串行到并行的转变，本质上是用计算换通信。我们期望通过增加计算资源（更多核心）来缩短时间，但代价是引入了复杂的通信开销。成功的并行化，就是一场精密的“成本效益分析”：必须确保并行计算带来的收益，远大于上述四类通信开销的总和。在多数情况下，若不精心设计，通信的“成本”会轻易压倒并行的“收益”。

6.2 并行的优化方式的共性之处

6.2.1 最小化通信与同步

这是并行优化中最核心、最首要的原则。通信是并行程序相对于串行程序新增的最大成本。

共性体现：GPU (CUDA): 减少 cudaMemcpy 的调用次数和传输的数据量，是优化 GPU 性能的第一步。多进程 (MPI): 避免在紧密循环中调用 MPI_Allreduce 等全局同步操作，是降低网络延迟影响的关键。多线程 (OpenMP/pthread): 缩小临界区（Critical Section）的范围，减少锁的持有时间，降低线程间的等待，是提升多线程效率的核心。本质: 无论是物理的数据传输（CPU-GPU, 跨节点网络），还是逻辑上的同步等待（锁，路障），都是“通信”。优化的目标就是让每个计算单元尽可能长时间地“埋头干活”，而不是“抬头看路”或“等人同步”。

6.2.2 优化任务粒度与负载均衡

任务粒度决定了计算和开销的比例，是并行效率的关键调节阀。

共性体现: 过细的粒度: 所有实验都证明，当任务被切分得太小时（如处理一个只有少量组合的 PT），启动和管理并行任务（线程创建、GPU Kernel 启动）的固定开销会超过计算本身带来的收益。不均的粒度: 当任务量差异巨大时（不同 PT 生成的口令数天差地别），静态的任务分配会导致严重的负载不均衡。部分核心早早完成任务后进入空闲等待，造成资源浪费。本质: 寻找一个“最佳平衡点”（Sweet Spot）。任务粒度需要足够大，以摊薄并行开销；同时又要足够细和均匀，以保证所有计算资源都能被持续利用。pthread 实验中通过调整阈值来寻找这个平衡点，就是这一原则的直接实践。

6.2.3 提升数据局部性与独立性

让数据尽可能靠近处理它的计算单元，让计算单元尽可能独立工作。

共性体现: 数据独立: 在多线程中，使用 thread_local 存储为每个线程创建独立的本地结果容器（thread_local_guesses），避免了对全局共享容器的频繁加锁，是典型的提升独立性的优化。数据局部: SIMD: 将数据预处理成连续、对齐的内存块，是为了让数据能被高效地加载到 SIMD 寄存器中，这是最高级别的数据局部性。多线程: 避免“伪共享”（False Sharing）就是一种维护数据局部性的手段，确保不同线程操作的数据在物理上位于不同的缓存行，避免了不必要的缓存同步。本质: 现代计算机的内存访问速度远慢于计算速度。将数据放在离核心更近的地方（寄存器 > L1/L2/L3 缓存 > 主内存 > 另一节点的内存），并减少对共享数据的依赖，可以直接降低访存延迟和同步开销。

6.2.4 隐藏延迟

当通信和数据准备不可避免时，尝试用有用的计算去掩盖它。

共性体现: GPU (CUDA): CPU-GPU 异步并行处理是这一原则的典范。使用 CUDA 流（Streams），可以在 GPU 执行计算任务（Kernel）的同时，让 CPU 去准备下一批要处理的数据，或者处理上一批 GPU 返回的结果。多进程/多线程: 在一个进程/线程等待 I/O 或网络数据时，操作系统会将其切换出去，让其他就绪的进程/线程运行。这在更宏观的层面上也是一种延迟隐藏。本质: 将程序的执行流程从串行的“准备-> 传输-> 计算-> 返回”模式，改造为并行的“流水线”（Pipeline）模式。通过重叠计算和通信，提升硬件的综合利用率。

总结: 无论使用何种并行技术，优化的过程万变不离其宗，都是在这四个维度上进行权衡和改进。一个成功的并行程序，必然是一个在通信成本、任务粒度、数据局部性和延迟隐藏之间取得了精妙平衡的系统。

7 此部分工作为前面历次作业没有涉及的全新工作

我们之前已经进行过的优化工作有：动态均衡负载、生产者-消费者模型、改变存储结构、一次性多个 PT、hash-guess 并行进行、GPU-CPU 压榨进行工作、动态调整 GPU、CPU 的工作量。现在我们对于我们之前已经得到的优秀模型进行模型融合、取长补短，争取得到加速最大化的模型。

在之前的实验中我们在 guessing.cpp 的 init 函数中在初始化队列的时候保持优先队列中的按概率降序排序，同时对 hash 过程进行 SIMD 加速，成功完成了在优化编译的情况下对于 hash 过程和 guessing 过程的同时加速。在多进程的实验中，我们对于 hash 和 guessing 过程呈串行的运行的方式不甚满意，所以我们将这两个过程本身进行并行化，使得时间进一步缩短。在本次实验中我们将其进行最终的融合，尝试使得这些加速一起实现，最终得到我们最终加速效果最大的结果。

7.1 模型融合描述

7.1.1 SIMD 加速 HASH

在我们进行 SIMD 加速的实验中，我们对于密集型数据使用了 SIMD 即一次性处理四个数据的方式对其进行加速。并且探究出了结论：在不优化编译的情况下 SIMD 因为开销的问题并未能实现加速，不过在 o1，o2 优化编译的情况下可以实现对于 hash-time 的大幅度加速。所以我们选择 SIMD 作为我们加速的第一步。

```
1 #define FF_SIMD(a, b, c, d, x, s, ac) { \
2 (a) = vaddq_u32((a), vaddq_u32(vaddq_u32(F_SIMD((b), (c), (d)), (x)), vdupq_n_u32(ac))); \
3 (a) = ROTATELEFT_SIMD((a), (s)); \
```

```
(a) = vaddq_u32((a), (b)); \
}
void MD5Hash_SIMD(const string inputs[4], bit32 states[4][4]);
```

7.1.2 使用有序排列优化 guessing 中的 Init 函数

在 openmp 实验中因为 openmp 加速在编译优化的情况下并不能实现并行加速，所以我们对 guessing 文件中的 init 函数进行了改造，使得在编译优化的情况下仍然能够对 hash 和 guessing 进行加速。

```
bool inserted = false;
for (auto iter = priority.begin(); iter != priority.end(); ++iter) {
    if (pt.prob > iter->prob) {
        priority.insert(iter, pt);
        inserted = true;
        break;
    }
}
if (!inserted) {
    priority.emplace_back(pt);
}
```

7.1.3 使用多进程 MPI 优化 guessing 中的 Generate 函数

在我们专门选题的实验中，对于 guessing 的加速操作主要都集中于 generate 函数的最后两个循环，包括 pthread，openmp，MPI，GPU 等等，这里我们经过反复地试验发现 MPI 对于这个过程的加速与其他的加速方式融合的最为出色，所以我们这个部分使用 MPI 来进行并行化操作。

```
// MPI 并行化：将工作分配给不同进程
int total_values = pt.max_indices[pt.content.size() - 1];
int values_per_process = total_values / mpi_size;
int remainder = total_values % mpi_size;

int start_idx = mpi_rank * values_per_process + min(mpi_rank, remainder);
int end_idx = start_idx + values_per_process + (mpi_rank < remainder ? 1 : 0);

// 每个进程处理自己的部分
for (int i = start_idx; i < end_idx; i++)
{
    string temp = guess + a->ordered_values[i];
    guesses.emplace_back(temp);
    total_guesses += 1;
}
```

7.1.4 使用一次性处理多个 PT 进行并行化加速

在 GPU 实验中，我们可以发现一次性使用多个 PT 对其进行加速能够使得非优化和优化编译之后运行的时间得到显著的下降，所以我们将这个方法也加入进了我们的优化行列清单中。

```
// 每个进程分配的 PT 索引
vector<int> pt_indices_per_process(mpi_size, -1);
vector<PT> pts_to_process(mpi_size);

// 主进程分配 PT 给各进程
if (mpi_rank == 0) {
    for (int i = 0; i < actual_batch_size && i < mpi_size; i++) {
        pt_indices_per_process[i] = i;
        pts_to_process[i] = priority[i];
    }
}
```

7.1.5 对于 SIMD 的调用在主 cpp 中进行封装并使用 openmp 进行多线程加速

为了避免之前任务中繁杂的数据分批次处理，我们直接采用多线程 openmp 对于其 SIMD 的分割数据过程进行调用。

```
#pragma omp parallel for reduction(+:total_cracked)
for (size_t i = 0; i < complete_batches; i++) {
    string inputs[4];
    bit32 states[4][4];

    // 准备 4 个密码
    for (int j = 0; j < 4; j++) {
        nputs[j] = local_guesses[i*4 + j];
    }

    // 使用 SIMD 版本哈希计算
    MD5Hash_SIMD(inputs, states);

    // 检查是否破解
    for (int j = 0; j < 4; j++) {
        if (test_set.find(inputs[j]) != test_set.end()) {
            #pragma omp atomic
            total_cracked++;
        }
    }
```

```
21  }
```

7.1.6 使用 hash-guess 流水线方式对整体过程进行优化

利用多进程编程，在进行口令猜测的同时，利用新进程对口令进行哈希。实际情况中，进程间的通讯和 workload 传递会产生一定的开销。

之前的优化都只是在 hash 或者 guess 内部对其进行优化，现在我们总整体流程出发，尝试使用多进程编程，在进行口令猜测的同时，利用新的进程进行口令哈希。即采用多进程（多线程理论上也可以）编程，在猜测完一批口令之后，对这批口令进行哈希，同时继续进行新口令的生成。第一轮口令哈希结束、第二轮猜测结束之后，再同时进行第二轮口令哈希、第三轮口令猜测。

```
1  串行模式: 生成 1 - 哈希 1 - 生成 2 - 哈希 2 - ...
2  重叠模式: 生成 1 - 生成 2 - 生成 3 - ...
3                -    -     -
4                哈希 1 - 哈希 2 - 哈希 3 - ...
```

7.2 结果展示

猜测流程 \ 优化编译方式	不优化编译	o1 优化	o2 优化
hash	17.703	9.690	8.132
guess	6.908	0.734	0.537

表 2: 串行版本 (单位:s)

猜测流程 \ 优化编译方式	不优化编译	o1 优化	o2 优化
hash	3.172	1.933	1.707
guess	0.264	0.011	0.088

表 3: 并行版本 (单位:s)

我们可以看到我们的优化效果非常的显著。最快的 guess-time 已经达到了 11ms 的级别，已经远远低于最开始的 0.7s 的数量级可以说

7.2.1 优化效果分析

编译优化的显著提升 无论串行还是并行版本，编译优化 (-O1 和-O2) 都带来了显著的性能提升在串行版本中，hash 操作从不优化的 17.703s 降至-O2 优化的 8.132s，提升了约 54guess 操作优化效果更为明显，从 6.908s 降至 0.537s，提升了约 92

O1 与 O2 优化对比 在大多数情况下，-O2 优化比-O1 提供更好的性能特例：并行版本的 guess 操作在-O1 优化下达到了惊人的 0.011s，比-O2 优化 (0.088s) 快 8 倍

并行版本 vs 串行版本 hash 操作：并行版本在-O2 优化下比串行版本快约 4.8 倍 (1.707s vs 8.132s) guess 操作：并行版本在-O2 优化下比串行版本快约 6.1 倍 (0.088s vs 0.537s) 在-O1 优化下，guess 操作的并行加速比高达 66.7 倍 (0.011s vs 0.734s)

并行化效率 guess 操作的并行化效率明显高于 hash 操作这表明 guess 操作更适合并行处理，可能是因为其任务更易于划分且具有更少的同步需求

7.2.2 执行时间分布分析

整体耗时组成 从测试输出可见，训练时间 (28.6136s) 远高于猜测和哈希时间在最优情况下 (-O2 并行)，实际破解过程只占总执行时间的约 6.3

任务间平衡 在并行版本中，hash 操作 (1.707s) 显著高于 guess 操作 (0.088s) 这表明哈希计算成为了瓶颈，未来优化应聚焦于进一步优化 MD5 哈希计算

7.2.3 破解效率分析

破解成功率 成功破解了 2,477,307 个密码考虑到测试集约 100 万个密码，这说明许多密码被重复破解了多次

性能-结果比 在最佳配置下 (并行版本 +-O2 优化)，每秒约可处理 5,687 个密码 (10,257,595/1.8s) 训练过程虽然耗时较长，但在实际应用中是一次性成本

7.3 结果分析

7.3.1 SIMD 加速

SIMD 指令允许 CPU 同时对 4 个密码进行处理，理论上可提供近 4 倍的计算吞吐量在未优化编译时，SIMD 的初始化和数据准备开销抵消了并行计算的优势在 O1/O2 优化编译下，编译器能够更好地优化 SIMD 指令序列，减少调用开销 MD5 算法中的位运算和数学运算非常适合 SIMD 并行处理使用 vaddq_u32 等 NEON 指令集操作可以充分利用现代处理器的向量处理单元

7.3.2 有序排列优化 Init 函数

传统方法通常是先插入再排序，而有序插入避免了全局排序操作对于频繁更新的优先队列，维持有序状态的增量成本低于周期性完全排序编译器更容易优化简单的线性搜索和插入操作此方法减少了内存碎片和缓存未命中，改善了缓存局部性在 O1/O2 优化下，编译器可进一步内联此类简单函数，减少函数调用开销

7.3.3 MPI 优化 Generate 函数

密码生成是高度可并行的任务，各索引之间没有依赖关系 MPI 实现了进程级并行，每个进程处理总工作量的一部分动态计算的负载均衡方案 (values_per_process 和 remainder 处理) 确保各进程工作量近似均等 MPI 相比其他并行方法 (如 OpenMP) 扩展性更好，可跨节点部署进程间独立内存空间减少了线程同步和缓存一致性问题

7.3.4 一次性处理多个 PT 并行化加速

传统单 PT 处理方式造成并行资源浪费，同时处理多个 PT 提高了资源利用率增大了计算粒度，减少了并行开销与实际计算的比率批量任务分配减少了进程间通信频率静态任务分配 (pt_indices_per_process) 避免了动态调度开销此方法与 MPI 模型契合，每个进程可以独立处理一个完整 PT

7.3.5 OpenMP 加速 SIMD 调用

形成了两级并行架构：SIMD(数据级) 和 OpenMP(线程级) pragma omp parallel for reduction(+:total_cracked) 自动处理结果合并，避免了手动同步 pragma omp atomic 确保线程安全的计数器更新批量处理 (complete_batches) 提高了并行效率，减少了线程管理开销优化了非 4 整除情况的处理 (remainder 部分)，确保全部数据都被处理

7.3.6 Hash-Guess 流水线优化

传统串行模式 CPU 利用率低，密码生成和哈希验证交替空闲流水线模式将两个阶段并行化，提高了资源利用率使用独立线程处理哈希计算，主线程专注于密码生成使用 move 语义和互斥锁高效传递数据，减少复制开销 pending_hash_guesses 作为生产者-消费者模型的缓冲区，平衡了两个阶段的速度差异

8 总结

8.1 对“性能优化”的认知重塑：从代码到系统

最深刻的教训是，在现代计算中，编译器是最高效的“并行化工具”。在几乎所有实验中，一个经过充分优化（-O2/-O3）的串行程序，其性能提升远超盲目引入并行化所带来的收益。这让我们明白，任何并行优化的第一步，都应该是建立一个最强的、经过编译器优化的串行基准。

性能瓶颈的不可预测性：我们从“想当然”地认为计算是瓶颈，到通过性能剖析（Profiling）发现真正的瓶颈往往在别处：内存复制、数据对齐、锁竞争、网络延迟、PCIe 传输……这培养了一种“测量，别猜测”（Measure, Don’t Guess）的科学方法论，是性能工程的基石。

从算法思维到体系结构思维：最初我们可能只关注循环和算法逻辑，但整个过程迫使我们思考更深层的问题：数据是如何在缓存（Cache）中流动的？伪共享（False Sharing）是如何发生的？CPU 和 GPU 之间的数据通道有多宽？多节点间的网络延迟是多少？这种从代码逻辑到硬件体系结构的思维转变，是本次实验最有价值的收获之一。

8.2 对“并行计算”的深刻理解

并行必有开销：我们亲身体会到，并行化并非简单的“人多力量大”，而是引入了一系列新的、在串行世界中不存在的开销。无论是线程创建与销毁、上下文切换、锁的获取与释放，还是 MPI 的通信延迟、GPU 的数据传输，这些都是并行计算的“税”。

任务特性决定并行上限：通过对 guess_time 和 hash_time 的对比，我们清晰地看到问题的“可并行度”是天生的。hash_time 因为其任务独立、无数据依赖的特性，几乎总能从并行中受益。而 guess_time 因为其复杂的控制流、不规则的内存访问和内在依赖，导致其并行化困难重重。这让我们学会了在动手前，先评估一个问题是否“值得”并行化。

“工具”与“问题”的匹配艺术：我们不再迷信任何一种“银弹”技术。SIMD 适合数据级并行，多线程适合共享内存下的任务级并行，MPI 适合跨节点的粗粒度并行，而 GPU 则为大规模数据并行而生。为错误的问题选择错误的工具，效果只会适得其反。例如，将一个数据量小、逻辑复杂的任务强行交给 GPU，就是最典型的错配。

8.3 工程实践与解决问题能力的提升

从失败中学习的能力：本次实验的大部分尝试在初期都“失败”了（未能实现加速）。但最有价值的知识，恰恰来自于对这些失败的刨根问底。分析为什么 pthread 比串行慢，为什么 GPU 在优化后反而性能更差，这个过程锻炼了我们分析问题、定位根源并提出改进方案的系统性能力。

代码设计的“并行友好”意识：在经历了种种困难后，我们未来在编写代码时会自然地带上“并行”的视角。例如，会倾向于使用局部变量（Thread-Local）来减少共享；在设计数据结构时会考虑缓存行对齐以避免伪共享；会尽量将数据处理设计成批处理模式，以适应 GPU 或 MPI 的通信模型。

对“加速比”的批判性看待：我们不再简单地追求一个漂亮的“加速比”数字，而是会去分析其背后的构成。一个看似不错的整体加速，可能是因为某个“易并行”的部分被大幅优化，而核心瓶颈依然存在。这种对结果的审慎和批判性思维，是成为优秀工程师的重要特质。

9 致谢

感谢张逸飞学长的耐心解答和王刚老师的倾情讲授。

10 代码链接

Github 代码链接 https://github.com/Mercycoffee12138/guess