南開大學
Nankai University

计算机学院

并行**Lab04**

多进程并行加速的口令猜测算法

姓名：刘迪乘

学号：2312810

专业：计算机科学与技术

2025 年 6 月 14 日

目录

1 实验要求

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

本次问题：利用 MPI 对 PCFG 算法的猜测产生过程进行多进程并行化优化加速。

2 实验环境

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

具体硬件参数如下：

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

表 1: AMD R7 9700X 硬件参数

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

表 2: OpenEuler 服务器硬件参数

编译器版本：

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

依据课程要求，本次实验代码开源在 https://github.com/July-h5kf3/NKU-parallel-computing

3 基础要求

MPI 编程在形式上与 pthread 没有太大的区别。本此实验的重点是了解 MPI 编程的过程和细节，并且对口令猜测算法进行并行化。

在多线程实验中是针对一个 PT 的最后一个 Segment 进行填充时，进行多线程填充。同样地，在本次实验中，对最后一个 Segment 进行多**进程**填充。

3.1 算法设计

本算法采用主从（Master-Worker）并行计算模式，结合数据并行的思想，对基于概率上下文无关文法（PCFG, Probabilistic Context-Free Grammar）的口令猜测过程进行加速优化。

程序启动时，通过 `MPI_Init()` 初始化 MPI 并行环境，确保所有进程加入同一通信域。其中，编号为 0 的进程被指定为主进程（Master），负责处理串行控制逻辑，包括初始化与维护核心优先队列、

选取概率最高的密码模板（PT）、分发生成任务，以及汇总与输出最终结果和进度信息。其余进程（包括主进程自身）作为工作进程（Worker），共同参与计算密集型的猜测生成过程。

为简化进程间的通信操作并避免传递复杂的 C++ 对象（例如包含 `std::vector` 的 PT 模板类），每个进程在初始化阶段都会独立加载并训练一份只读的 PCFG 模型副本。虽然这在启动阶段带来一定的资源开销，但它确保了所有进程在后续计算中拥有完全一致的模型数据，从而避免了频繁的通信，提高了并行效率。

并行化的核心在于对 `Generate()` 函数进行重构，重点加速每个 PT 中最后一个 Segment 的填充过程。当主进程从优先队列中取出概率最高的 PT 时，即启动一轮并行生成，具体过程如下所述：

- **任务解构与广播：** 由于 PT 是复杂对象，不能直接通过 MPI 进行广播，因此主进程提取出生成猜测所需的关键信息，包括：
  - 公共前缀（Common Prefix）：即除最后一个 Segment 外的已实例化部分，作为所有猜测的前缀；
  - 最后一个 Segment 的类型与长度：用于在本地模型中定位；
  - 迭代范围（Iteration Range）：该 Segment 所包含的候选值总数，决定并行循环范围。

  这些数据被打包后通过 `MPI_Bcast` 广播至所有进程。
- **并行任务划分与生成：** 各工作进程接收到广播数据后，采用块划分（Block Partitioning）策略，将候选值集合划分为若干不重叠子区间。例如，若某 Segment 包含 N 个候选值，则编号为 i 的进程负责处理索引区间 $[i \cdot N/P, (i+1) \cdot N/P)$，其中 P 为总进程数。每个进程在自己的分区内，将公共前缀与对应候选值拼接，生成部分猜测并存入本地向量 `local_guesses`。
- **结果汇总与收集：** 由于各进程生成的猜测数量可能不等，主进程使用 `MPI_Gatherv` 操作收集结果。每个进程首先将本地字符串向量序列化为以空字符（`\0`）分隔的长字符串，然后发送至主进程。主进程收到所有数据后反序列化，合并为全局猜测集合 `guesses`，完成该轮生成。

通过上述设计，原本在单进程中串行执行的口令生成过程被有效拆解为多个进程并行执行，大大提升了生成效率。主进程主导控制逻辑与任务分配，所有进程协同完成计算密集型任务，从而在保证功能正确性的同时实现显著的性能提升。

3.2 代码实现

首先是初始化，由于 MPI_init 只能调用一次的特性，将其放在了 main.cpp 中，并对串行部分的代码进行包裹，由于训练文件并不好直接在各个进程之间通过通信传播，因此各个进程先各自完成自己进程的训练，实际上这也减少了后续在并行部分的通信，这样主程序只需要传递需要填充的 Segment 对应的 index 然后各个子进程就可以知道需要填充的集合。然后在主循环部分，只有主进程会进行数量统计，各个步骤的计时以及 MD5hash 过程。

此外由于原本的代码框架中，当产生的数量到达一定数目时就会停止产生新的口令以及 SIMD 过程，因此我设计了一个布尔类型参量 master_is_done 来提示各个进程产生口令是否结束，在每次循环中都进行一次广播。

```
1  int main(int argc, char** argv)
2  {
```

```
// ofstream logFile("top1e6.txt");
MPI_Init(&argc, &argv);
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
double time_hash = 0;  // 用于 MD5 哈希的时间
double time_guess = 0; // 哈希和猜测的总时长
double time_train = 0; // 模型训练的总时长
PriorityQueue q;
auto start_train = system_clock::now();
q.m.train("/home/s2312810/homework/data/Rockyou.txt");
q.m.order();
auto end_train = system_clock::now();
if(world_rank == 0)
{
    auto duration_train = duration_cast<microseconds>(end_train - start_train);
    time_train = double(duration_train.count()) * microseconds::period::num / microseconds::pe
    q.init();
    cout << "here" << endl;
}
int curr_num = 0;
auto start = system_clock::now();
int history = 0;
bool master_is_done = false;
while (1)
{
    // master_is_done = false;
    MPI_Bcast(&master_is_done, 1, MPI_C_BOOL, 0, MPI_COMM_WORLD);
    if (master_is_done)
    {
        break;
    }
    q.PopNext();
    if(world_rank == 0)
    {
        ...//计时操作以及 log
        int generate_n=10000000;
        if (history + q.total_guesses > generate_n && master_is_done != 1)
        {
            ....
            master_is_done = 1;
        }
```

```
45      }
46      // 为了避免内存超限，我们在 q.guesses 中口令达到一定数目时，将其中的所有口令取出并且进行哈希
47      // 然后，q.guesses 将会被清空。为了有效记录已经生成的口令总数，维护一个 history 变量来进行记录
48      if (curr_num > 1000000)
49      {
50          Hash
51      }
52      }
53  }
54  MPI_Finalize();
55  return 0;
56  }
```

对于各个进程都会进入的 PopNext 函数需要保证其它串行部分都是只有主线程执行。并在主线程中确定需要进行填充产生口令的 PT。需要注意的是这个过程不需要把 PT 广播到各个子进程中，这是因为每个子进程进行填充的任务需要的参数只有 pre_terminal,values,num, 而不需要整个 PT，因此广播任务我将其放在了 Generate 函数中，即 PT 已经确认的情况下，在主进程中求出这些参数，并广播至各个进程。

```
1
2   void PriorityQueue::Generate(PT pt)
3   {
4   int rank;
5   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6
7   string guess;
8   long long broadcast_data[3];
9
10  if (rank == 0)
11  {
12      if (!pt.content.empty())
13      {
14          //计算前缀
15          if (pt.content.size() > 1)
16          {
17              for (size_t i = 0; i < pt.content.size() - 1; ++i)
18              {
19                  int seg_idx_val = pt.curr_indices[i];
20                  if (pt.content[i].type == 1) guess += m.letters[m.FindLetter(pt.content[i])].o
21                  if (pt.content[i].type == 2) guess += m.digits[m.FindDigit(pt.content[i])].ord
22                  if (pt.content[i].type == 3) guess += m.symbols[m.FindSymbol(pt.content[i])].o
23              }
```

```
24          }
25          // 准备最后一个 segment 的信息
26          int last_idx = pt.content.size() - 1;
27          const segment& last_seg = pt.content[last_idx];
28
29          broadcast_data[0] = last_seg.type;
30          broadcast_data[2] = pt.max_indices[last_idx];
31
32          if(last_seg.type == 1) broadcast_data[1] = m.FindLetter(last_seg);
33          else if(last_seg.type == 2) broadcast_data[1] = m.FindDigit(last_seg);
34          else if(last_seg.type == 3) broadcast_data[1] = m.FindSymbol(last_seg);
35          else broadcast_data[1] = -1;
36      }
37      else
38      {
39          broadcast_data[0] = -1; // 标记为无效 PT
40      }
41
42  }
43  MPI_Bcast(broadcast_data, 3, MPI_LONG_LONG, 0, MPI_COMM_WORLD);
44
45  if (broadcast_data[0] == -1)
46  {
47      return;
48  }
49
50  int guess_len = (rank == 0) ? guess.length() : 0;
51  MPI_Bcast(&guess_len, 1, MPI_INT, 0, MPI_COMM_WORLD);
52
53  vector<char> guess_buffer(guess_len + 1);
54  if (rank == 0)
55  {
56      strcpy(guess_buffer.data(), guess.c_str());
57  }
58  MPI_Bcast(guess_buffer.data(), guess_len + 1, MPI_CHAR, 0, MPI_COMM_WORLD);
59  guess = string(guess_buffer.data());
60
61  int last_seg_type = broadcast_data[0];
62  int index_in_model = broadcast_data[1];
63  int loop_bound = broadcast_data[2];
64
65  segment* a;
```

```
66  if(index_in_model != - 1)
67  {
68      if (last_seg_type == 1)       a = &m.letters[index_in_model];
69      else if (last_seg_type == 2)  a = &m.digits[index_in_model];
70      else if (last_seg_type == 3)  a = &m.symbols[index_in_model];
71  }
72  fill_preterminal(guess, a, loop_bound);
73  }
```

其中每个子进程进行填充的函数 fill_preterminal 的内容与多线程任务中的思路是一样的，就是各自维护一个 local_guesses 将填充后的结果先各自保存在这些容器中，最后打包给主进程与 guesses 合并得到完整的猜测集合。

```
1
2   void PriorityQueue::fill_preterminal(const std::string& base,segment* a,int num)
3
4   int rank,size;
5   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
6   MPI_Comm_size(MPI_COMM_WORLD,&size);
7   vector<string> local_guesses;
8
9   if (num <= 0 || a == nullptr)
10  {}
11  else
12  {
13      long long n_total = num;
14      long long start_idx = (long long)rank * n_total / size;
15      long long end_idx = (long long)(rank + 1) * n_total / size;
16
17      local_guesses.reserve(end_idx - start_idx);
18
19      for (long long i = start_idx; i < end_idx; ++i)
20      {
21          local_guesses.emplace_back(base + a->ordered_values[i]);
22      }
23  }
24  //将每个进程的 local_guesses 链接
25  size_t total_local_len = 0;
26  for (const auto& s : local_guesses)
27  {
28      total_local_len += s.length() + 1;
29  }
```

```
30  string concat_str;
31  if (total_local_len > 0)
32  {
33      concat_str.reserve(total_local_len);
34      for (const auto& s : local_guesses)
35      {
36          concat_str.append(s);
37          concat_str.push_back('\0');
38      }
39  }
40  //在主线程收集这些链接后的字符串大小
41  int local_size = concat_str.length();
42  vector<int> all_sizes;
43  if (rank == 0)
44  {
45      all_sizes.resize(size);
46  }
47  MPI_Gather(&local_size, 1, MPI_INT, all_sizes.data(), 1, MPI_INT, 0, MPI_COMM_WORLD);
48
49  //接受数据
50  vector<int> displacements;
51  vector<char> recv_buffer;
52  if (rank == 0)
53  {
54      displacements.resize(size, 0);
55      size_t total_recv_size = 0;
56      for (int i = 0; i < size; ++i)
57      {
58          displacements[i] = total_recv_size;
59          total_recv_size += all_sizes[i];
60      }
61      if (total_recv_size > 0)
62      {
63          recv_buffer.resize(total_recv_size);
64      }
65  }
66  MPI_Gatherv(concat_str.data(), local_size, MPI_CHAR,
67  recv_buffer.data(), all_sizes.data(), displacements.data(),
68  MPI_CHAR, 0, MPI_COMM_WORLD);
69  //主进程解包缓冲区并将其添加到主'guesses'
70  if (rank == 0 && !recv_buffer.empty())
71  {
```

```
72      size_t current_pos = 0;
73      while (current_pos < recv_buffer.size())
74      {
75          string s(&recv_buffer[current_pos]);
76          guesses.emplace_back(std::move(s));
77          current_pos += guesses.back().length() + 1;
78      }
79  }
80
```

3.3 性能测试

由于基础要求较为简单，因此只在不同进程数进行了测试，实验结果如下。

	串行	2	4	6	8
Time	0.705241s	0.717653s	0.609036s	0.572228s	0.555973s

3.4 结果分析

在对并行化程序进行性能测试后可以发现，尽管相较于串行版本实现了一定程度的加速，但其加速比（Speedup）并未随着进程数量的增加呈现出理想的线性增长。这一现象反映出当前并行策略在特定任务场景下的内在局限性，主要原因可归结为以下三点：

1. 通信开销主导的细粒度并行瓶颈 本实现采用的是细粒度并行（Fine-Grained Parallelism），即对每一个从优先队列中取出的预终结结构（PT）进行一次任务划分，仅对其最后一个 segment 的候选值进行并行填充。整个过程中，每个 PT 的处理均需经历完整的通信流程：主进程广播任务参数（如前缀字符串、segment 类型、模型索引等），所有子进程执行拼接任务，最终再将结果通过 `MPI_Gatherv` 汇总回主进程。

然而，当 segment 的候选值数量较少时（如数百或几千个），每个进程实际承担的计算负载极低，而通信延迟依然不可避免。在这种计算通信比（Computation-to-Communication Ratio）极低的场景中，通信成本反而成为主导因素，显著抑制了并行计算带来的收益。

2. 主从模式下的串行瓶颈与进程负载不均衡 本方案采用经典的主从模式（Master-Slave Model），其中主进程（Rank 0）承担了所有与优先队列相关的串行任务，包括计算新 PT 的概率、插入队列、Pop 操作等。在这一过程中，其它子进程处于阻塞等待状态，系统整体的执行进度受限于主进程的性能瓶颈，符合阿姆达尔定律（Amdahl's Law）中对于串行部分的限制描述。

此外，`PopNext` 函数作为隐式同步点，要求所有进程协同参与任务执行。这种高耦合的并行模式导致即便任务划分内部较为均衡，也无法充分释放多进程的计算潜力，最终形成资源利用效率偏低的问题。

3. 粗粒度并行作为优化方向 为进一步提升并行效率，应考虑采用粗粒度并行（Coarse-Grained Parallelism）策略。与当前多个进程协作处理单个 PT 不同，新的策略建议将任务粒度提升到 PT 层级，即每个子进程独立负责多个 PT 的完整处理过程。

这一策略已在本文后续实现中得到初步尝试。具体而言：

- 主进程转变为任务调度器，从优先队列中批量取出多个 PT 并分发给空闲进程；
- 子进程本地执行完整的填充与哈希过程，极大提升计算密度；
- 通信操作频率显著下降，仅在任务分发与结果回传阶段进行，通信开销可被摊薄；
- 系统总体的并行性与扩展性得到显著提升。

该模型本质上是一种任务池（Task Pool）或任务队列（Work Queue）架构，能够有效避免频繁同步与通信带来的瓶颈，体现了在并行设计中“合理粒度划分”的关键性原则。

综上所述，尽管本项目在并行化方面初步实现了任务加速，但通过实验结果与瓶颈分析可见，仅在 segment 层级进行并行是不足以充分发挥多核系统计算能力的。未来工作的重点应聚焦于任务粒度提升、通信模式优化与进程调度策略的进一步改进。

4 进阶要求：PT 层面的并行加速算法

4.1 算法设计

在介绍具体的算法前，我们再次回顾一下线程与进程之间的关系。线程是操作系统调度的最小单位，而进程则是资源分配的基本单位。一个进程可以包含多个线程，这些线程共享该进程的内存空间和资源，因此线程间通信效率较高，适合轻量级并行计算。而进程之间相互独立，具有各自的内存空间，通信需要依赖特定的机制如消息传递（Message Passing），这正是 MPI 所采用的并行模型。相比多线程，MPI 更适用于大规模分布式系统，能够跨节点并行执行任务，但其通信成本也更高。因此，在设计并行算法时，需要综合考虑任务的粒度、通信频率与负载平衡等因素，合理选择并行模型，以实现最优性能。

图 4.1: 进程与线程概念

因此在设计并行化优化时，我们不应将目光局限于某一种并行范式。实际上，进程并行与线程并行各有优势，二者可以互为补充。对于计算密集型任务，可以通过多线程共享内存、降低通信成本；而在任务划分明确、节点间需要解耦的场景中，则更适合采用多进程并行。

具体到本项目中，可以考虑在主进程通过 MPI 实现不同结构 (PT) 之间的并行分工，而每个进程内部再通过多线程进一步加速每个结构下口令的填充与生成。这种混合并行的方式（Hybrid Parallelism）能够在保证分布式扩展能力的同时，充分利用单节点的多核计算资源，从而达到更高的性能提升。换言之，可以让主进程先确定每个子进程需要填充的 PT，然后每个子进程则负责各个 PT 的填充，在填充时应用先前实现的多线程优化（这里应该是多线程，不是多进程），从而将通信成本交给开销较小的多线程并行。

那么我第一个要解决的问题就是，主进程从优先队列中，取出了若干个 PT，我该如何将他们传给各个子进程。在基础要求中提到，我们无法将整个类广播给各个子进程。但是注意到，在 PT 的填充任务中，**PT 中的所有成员并不是都必须的**。我们需要利用的内容有：

- **结构信息（`content`）**：即生成模板（如 L6D2S1），每个 segment 包括类型（L/D/S）与长度，均可表示为一个整数对（`type`, `length`）。
- **当前索引（`curr_indices`）**：表示已实例化 prefix 的位置，用于标识当前填充进度。
- **最大索引（`max_indices`）**：表示每段 segment 可选值总数，尤其最后一段的取值数量决定了并行填充的规模。

基于以上分析，我们可以设计一个高效的 PT 序列化与分发方案。主进程（rank 0）从优先队列中取出 N 个 PT（N 为 MPI 进程数），然后将每个 PT 对象“扁平化”为一个整数向量。这个向量包含了重建 PT 并执行生成任务所需的所有最小信息。具体的序列化格式可以设计如下：

$$\Big[\texttt{len}_c,\ \texttt{len}_{cur},\ \texttt{len}_{max},\ \underbrace{\texttt{content}_1,\ \ldots,\ \texttt{content}_{\texttt{len}_c}}_{\text{结构信息}},\ \underbrace{\texttt{curr}_1,\ \ldots,\ \texttt{curr}_{\texttt{len}_{cur}}}_{\text{当前索引}},\ \underbrace{\texttt{max}_1,\ \ldots,\ \texttt{max}_{\texttt{len}_{max}}}_{\text{最大索引}}\Big]$$

主进程将这 N 个序列化后的整数向量拼接成一个大的发送缓冲区 send_buffer。然后，利用 MPI 的集合通信函数 MPI_Scatterv，将这个大缓冲区的不同部分精确地分发给每个对应的子进程。每个子进程接收到属于自己的整数序列后，执行反序列化操作，在本地内存中重建出轻量级的 PT 对象。

一旦子进程拥有了各自的 PT 任务，它就可以调用内部的、使用 OpenMP 优化过的 Generate 函数。该函数利用多线程并行化最后一个 segment 的填充循环，将计算密集型任务分解到节点内的多个核心上，从而实现二级并行加速。最终，所有进程生成的口令猜测再通过 MPI_Gatherv 汇总到主进程进行后续处理。通过这种方式，我们成功地将 MPI 用于任务间（Inter-task）的粗粒度并行，将 OpenMP 用于任务内（Intra-task）的细粒度并行，实现了高效的混合并行计算模型。

4.2 代码设计

主要是对两个部分进行了更改，一个是 PopNext 函数，增添了主进程分发 pt，以及各个子进程接受解包的过程. 另一个是 generate 函数，由于 MPI 的限制，我们所有的猜测都是存储在主进程的队列中，因此需要将各个进程得到的猜测进行合并，故把 generate 函数的返回值由空转变为了 string 数组

由于 MPI 无法直接传输复杂的 C++类对象，我们需要设计一种机制来序列化和反序列化 `PT` 对象。我们仅提取生成口令所必需的最小信息集，并将其“扁平化”为一个整数向量。

$$\Big[\texttt{len}_c,\ \texttt{len}_{cur},\ \texttt{len}_{max},\ \underbrace{\texttt{content}_1,\ \ldots,\ \texttt{content}_{\texttt{len}_c}}_{\text{结构信息}},\ \underbrace{\texttt{curr}_1,\ \ldots,\ \texttt{curr}_{\texttt{len}_{cur}}}_{\text{当前索引}},\ \underbrace{\texttt{max}_1,\ \ldots,\ \texttt{max}_{\texttt{len}_{max}}}_{\text{最大索引}}\Big]$$

主进程负责从优先队列中取出 PT，进行序列化，并使用 `MPI_Scatterv` 将这些序列化后的数据块分发给所有工作进程。以下代码展示了主进程中序列化和准备分发参数的核心逻辑。

```
// 在 rank == 0 的主进程中
vector<int> send_buffer;
vector<int> sendcounts(size, 0);
```

```
vector<int> displacements(size, 0);
int current_displacement = 0;

// batch_pts 是从优先队列中取出的 PT 批次
for (int i = 0; i < batch_pts.size(); ++i) {
    const auto& pt = batch_pts[i];
    vector<int> temp_buffer;

    // 1. 序列化元数据
    temp_buffer.push_back(pt.content.size());
    temp_buffer.push_back(pt.curr_indices.size());
    temp_buffer.push_back(pt.max_indices.size());

    // 2. 序列化 content
    for(const auto& seg : pt.content) {
        temp_buffer.push_back(seg.type);
        temp_buffer.push_back(seg.length);
    }

    // 3. 序列化 indices
    temp_buffer.insert(temp_buffer.end(), pt.curr_indices.begin(), pt.curr_indices.end());
    temp_buffer.insert(temp_buffer.end(), pt.max_indices.begin(), pt.max_indices.end());

    // 4. 更新发送缓冲区和分发参数
    send_buffer.insert(send_buffer.end(), temp_buffer.begin(), temp_buffer.end());
    sendcounts[i] = temp_buffer.size();
    displacements[i] = current_displacement;
    current_displacement += sendcounts[i];
}
```

所有进程通过 MPI_Scatterv 接收到各自的数据后，进行反序列化，在本地重建 PT 对象，并调用内部的并行生成函数。

```
// 所有进程执行
int recv_count;
MPI_Scatter(sendcounts.data(), 1, MPI_INT, &recv_count, 1, MPI_INT, 0, MPI_COMM_WORLD);

vector<int> recv_buffer(recv_count);
MPI_Scatterv(send_buffer.data(), sendcounts.data(), displacements.data(), MPI_INT,
             recv_buffer.data(), recv_count, MPI_INT, 0, MPI_COMM_WORLD);

if (recv_count > 0) {
```

```
    // 反序列化，重建 my_pt ...
    PT my_pt;
    // (此处省略反序列化代码，与报告上一部分相同)

    // 调用 OpenMP 优化的生成函数
    vector<string> local_guesses = Generate_single_omp(my_pt);
    // 后续将 local_guesses 汇总到主进程
}
```

在每个 MPI 进程内部，我们利用 OpenMP 对计算最密集的 for 循环进行并行化。为确保线程安全并获得最佳性能，我们采用了“线程私有存储”策略。每个 OpenMP 线程将生成的口令存入各自的私有 vector 中，避免了在循环中对共享数据进行加锁，从而消除了竞争条件和同步开销。

以下是 Generate_single_omp 函数中并行化部分的核心实现：

```
// 假设已准备好前缀 guess_prefix 和最后一个 segment 指针 a
const int n_total = pt.max_indices.back();
vector<string> final_guesses;
vector<vector<string>> private_results; // 存储每个线程的私有结果

#pragma omp parallel
{
    #pragma omp single
    {
        // 仅一个线程执行：初始化私有结果容器
        private_results.resize(omp_get_num_threads());
    }

    int thread_id = omp_get_thread_num();

    // 并行化 for 循环，每个线程写入自己的私有 vector
    #pragma omp for schedule(static)
    for (int i = 0; i < n_total; i++) {
        private_results[thread_id].emplace_back(guess_prefix + a->ordered_values[i]);
    }
} // -- 并行区域结束 --

// 串行合并所有线程的结果
size_t total_size = 0;
for(const auto& vec : private_results) total_size += vec.size();
final_guesses.reserve(total_size);
for(const auto& vec : private_results) {
    final_guesses.insert(final_guesses.end(), vec.begin(), vec.end());
```

```
29 }
30
31 return final_guesses;
```

通过上述设计，我们成功构建了一个两级并行模型。MPI 负责在节点间进行粗粒度的任务划分，分发不同的口令结构（PT），实现了良好的可扩展性。OpenMP 则在每个节点内部，对计算密集的口令生成循环进行细粒度的并行，充分压榨了多核 CPU 的计算潜力。这种混合并行策略，将通信开销较大的进程间通信用于高层级的任务分配，而将通信成本极低（共享内存）的线程间并行用于底层的循环加速，从而在理论上能达到比单一并行范式更优的性能表现。

4.3 性能分析

我们尝试了在不同线程数和进程数下的实验结果，为了保证相对的高性能，保证进程数和线程数的乘积为 24, 依据服务器的要求进程数不多于 8。

	np = 2	np = 4	np = 6	np=8
time	19.0521s	8.39746s	7.98566s	6.52873s

4.4 结果分析

可以看到，虽然在工程上实现了 PT 层面的并行，但实验结果却出现了性能下降的现象，这是我们在并行优化中无法接受的结果。以下是对性能损失的简要分析。

严重的资源竞争导致性能不升反降 当在一个 MPI 进程中再启动多个 OpenMP 线程时，会引发显著的内存带宽竞争问题：同一进程内的所有 OpenMP 线程共享内存空间及其访问总线。由于字符串拼接属于内存密集型操作，它需要频繁地从内存中读取前缀字符串和候选值，并将新生成的字符串写回内存。当多个线程并发、高强度地进行读写时，会迅速占满内存带宽，导致线程间的互相阻塞，仿佛多条车道上的车辆都在同时试图通过一个狭窄的收费口。最终，线程数的增加并未带来加速，反而因内存访问冲突而显著拖慢程序运行。

负载不均衡问题 在 MPI 层面上进行 PT 并行处理时，本身就存在负载不均衡的风险：某些 PT 的候选值极多，处理时间长，而另一些 PT 候选值极少，处理时间短。引入 OpenMP 后，这一问题进一步放大。例如，一个候选值极少的“小” PT，其对应的 MPI 进程很快就能完成任务，但其内部的 OpenMP 线程可能刚刚初始化完毕，几乎未参与任何有效计算；而一个候选值极多的“大” PT，即使其内部线程在运行，也由于资源竞争问题，加速效果依然不佳。这种跨层次的不均衡，严重影响整体的并行性能。

任务粒度过小，OpenMP 开销反客为主 OpenMP 并行化所作用的目标为 `Generate` 函数中的 `for` 循环，而该循环的计算逻辑本身非常简单，仅包括字符串拼接（`base + value`）。在现代 CPU 中，这类操作计算量极低，瓶颈主要在于内存访问，而非算力。同时，OpenMP 本身并非“零成本”，其引入了包括线程创建与销毁、调度、内存同步、以及 `#pragma` 指令的运行时管理等大量开销。

当循环体内的任务过于轻量时，这些并行控制带来的开销往往超过并行所带来的收益。这种情况就像是为了搬运一箱羽毛而动用了重型起重机，准备工作远比实际任务更费时间。在本实验中，正是这种“准备成本过高”压倒了“计算收益”，使得多线程方案非但未优化，反而拖慢了程序运行。