ASC-RNA 优化思路总结

学科背景

RNA m5C 是一道典型的生物信息处理赛题。拿到题目的初期，我们面对一堆复杂的bash脚本和步骤说明，最困惑的是：整个工作流究竟在做什么？带着这种好奇心，我们阅读相关论文并补充了背景知识，最终将繁琐流程抽象为三个核心步骤：配对（Alignment）、去重（Deduplication）、统计（Counting）。

在讲述核心算法前，我们首先需要理解输入数据的产生与结构：
研究者从生物样本中提取 RNA，经 bisulfite 化学处理，将未修饰的 C 转换为 U，而 m5C 保持不变，再通过逆转录 PCR（RT-PCR）将 RNA 转录为 cDNA，再进行建库与高通量测序。最终得到的 .fastq 文件记录的是 RNA 衍生的碱基序列。以预赛数据集的一个case为例，包含约 8,000 万条 reads，每条长度约 50–200 bp。

接下来，这些reads会经过以下三个核心步骤：

1. 配对（Alignment）
   本质上是特殊的字符串匹配。给定每一条 RNA read，返回它对应到参考基因组的哪个位置。该步骤使用 HISAT-3N 完成。
2. 去重（Deduplication）
   PCR 会对原始 RNA 片段进行指数级扩增，测序后的许多片段其实是同一个原始片段的重复。我们使用 UMICollapse 工具，结合 UMI（可以理解为在扩增前给每一个片段接上一段随机的、独立的便签）和对齐信息，将属于同一原始 RNA 片段的扩增片段聚类，只保留一个代表序列。
3. 统计（Counting）
   统计实质上是一种**桶计数**过程：参考基因组上的每个碱基位置视为一个“桶”，当某条 RNA read 在特定位置上携带 m5C 修饰时，对应桶的计数 +1。最终生成一个表格，显示每个位点累积的 m5C 修饰数。该步骤由 hisat-3n-table 工具实现。

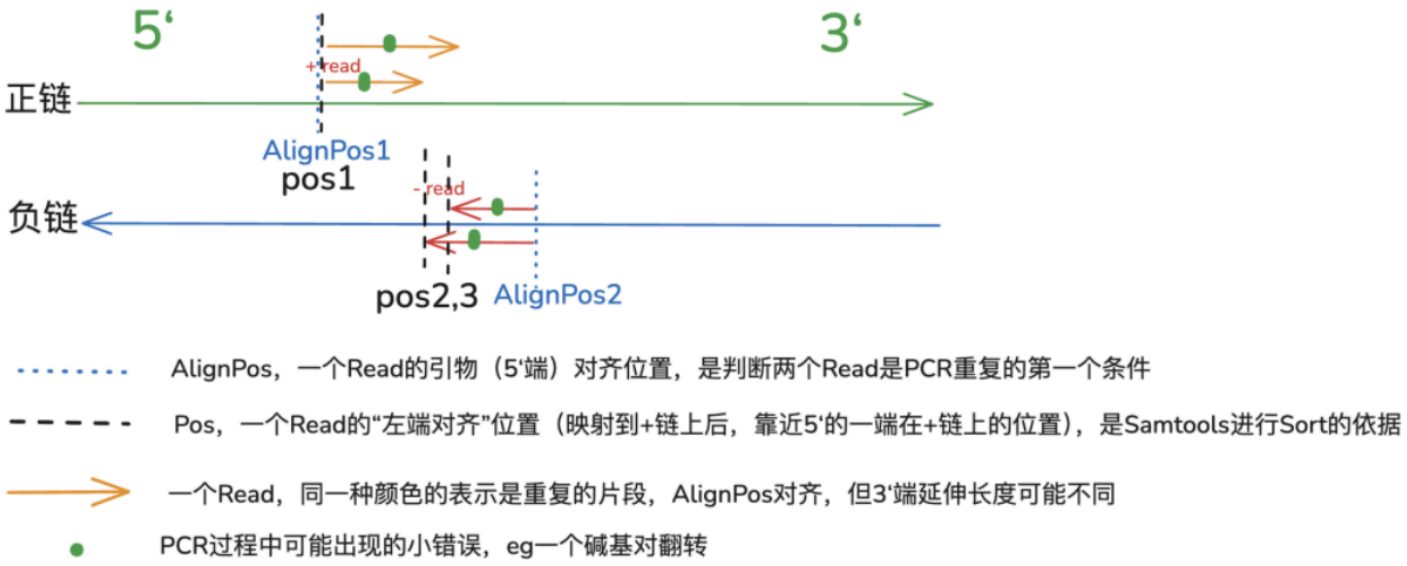


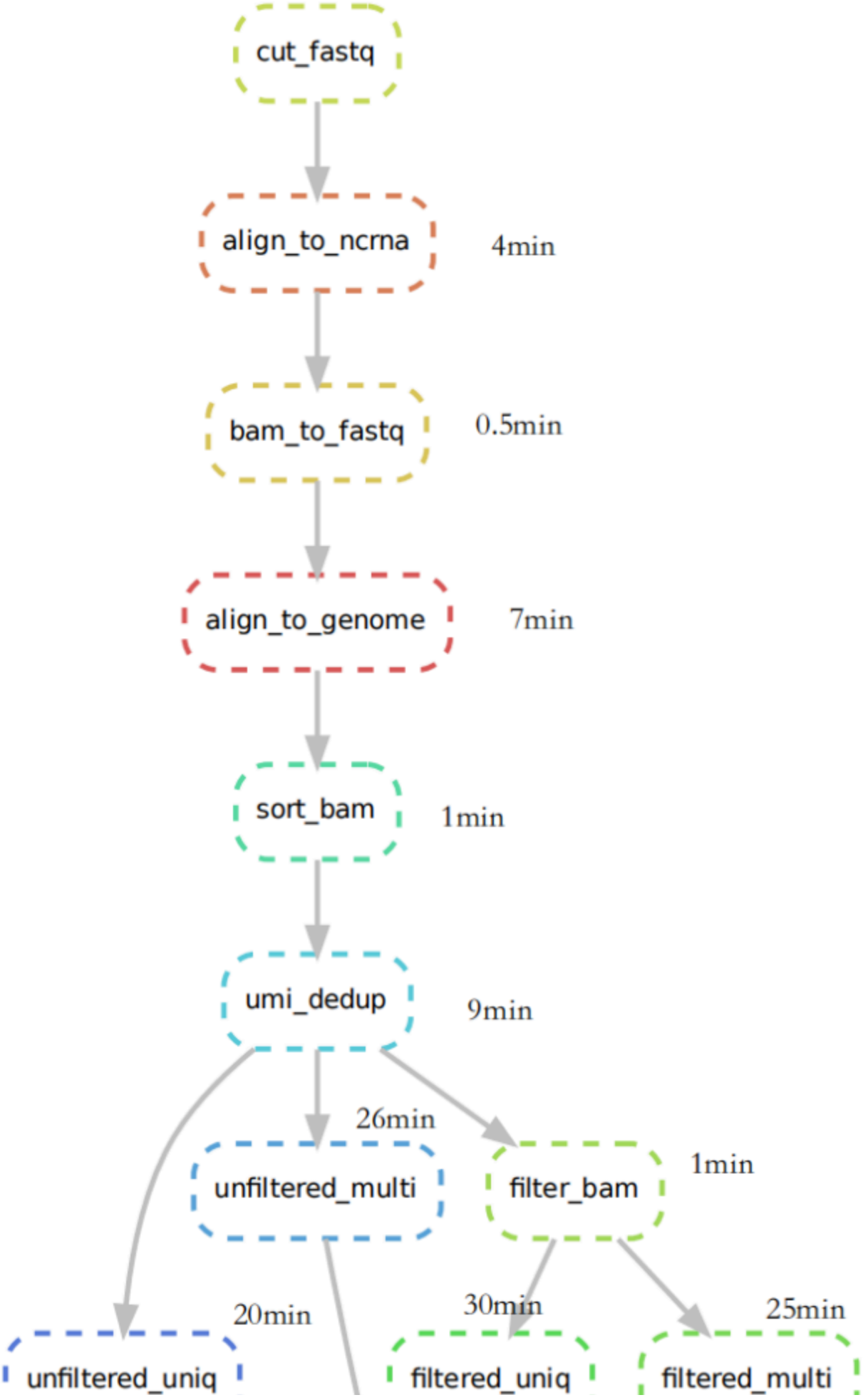
图1展示了配对与去重模块的整体流程

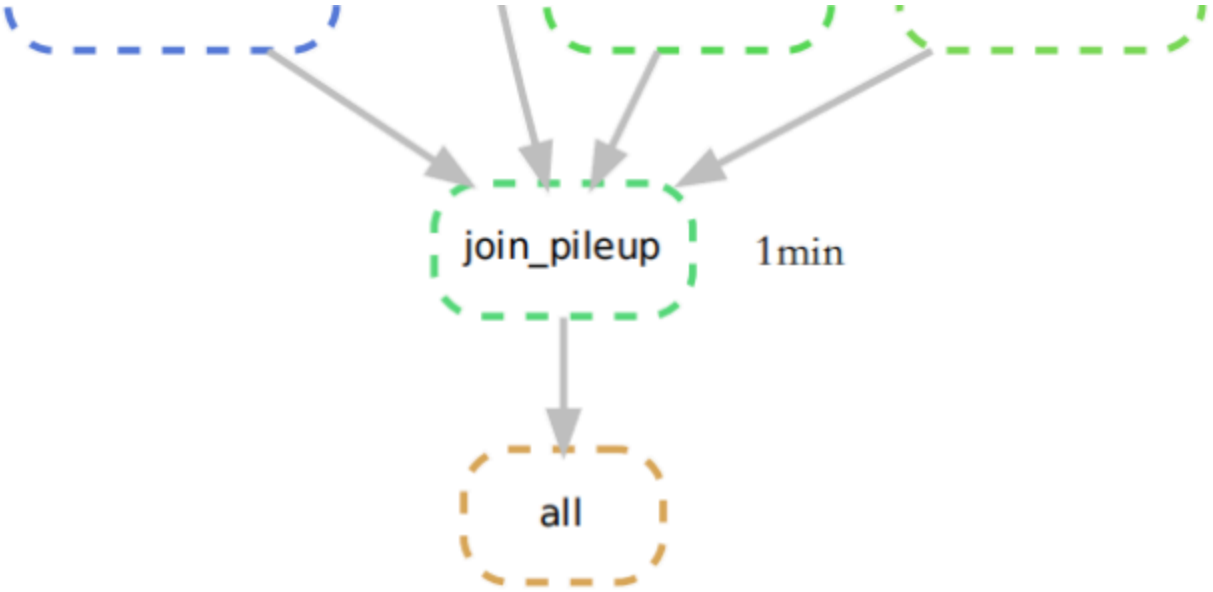
对背景流程的深入理解，使得我们在后续性能优化过程中有一个清晰的全局意识。从HISAT-3N、hisat-3n-table的计算量估计，到UMICollapse的算法特性理解，再到并行方案的重新设计，这种全局意识给我们持续的优化提供了很好的指引。

1. 基于工作流图的瓶颈识别

在赛题提示下，我们发现并调研了 Snakemake——一个专门为生物信息学数据分析而设计的 Python 化工作流管理工具。它类似 GNU Make，但采用 Python 语法，使得流程更加灵活且易于扩展。Snakemake 自动识别输入输出文件之间的依赖关系，从而构建一个 有向无环图（DAG）来控制任务调度与并行执行。

更为重要的是，Snakemake 提供内建的可视化与日志功能：将赛题提供的脚本转换为 Snakefile 后，通过 DAG 可视化，我们清晰地看到各步骤之间的依赖和并行度；再结合运行时 log 将耗时标注在节点边，我们就可以非常直观的识别出整个流程的瓶颈，如图2所示。


cut_fastq
align_to_ncrna
4min
bam_to_fastq
0.5min
align_to_genome
7min
sort_bam
1min
umi_dedup
9min
26min
unfiltered_multi
filter_bam
1min
20min
30min
25min
unfiltered_uniq
filtered_uniq
filtered_multi



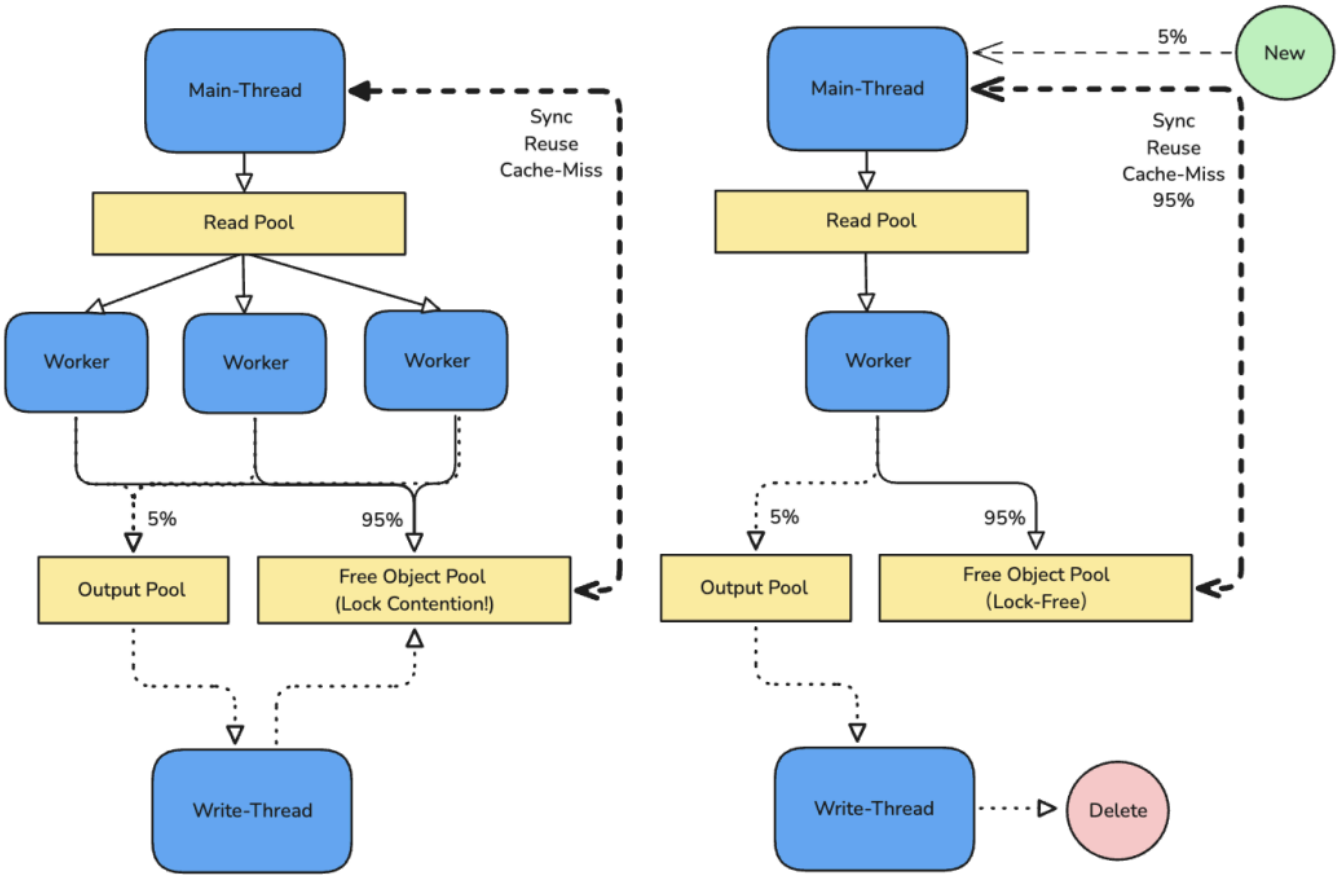
2. 基于profile结果的性能优化

hisat-3n-table的数据流分析

我们定位到的第一个性能瓶颈出现在 hisat-3n-table 模块。该步骤在初始实现中耗时占据了整个工作流约 50%。通过 VTune 的 flame graph 采样分析，我们发现程序执行过程中存在大量锁争抢，大量CPU时间消耗在同步等待而非有效计算。

为解决这一问题，我们进行了两方面优化：

1. 减少锁争抢
   为了定位和解决锁争抢问题，我们对数据流进行了分析，如图3所示。我们重构了原始实现中频繁同步的生产者-消费者模式，保留占比95%的worker Thread 回收路径，而将仅占比5%的output Thread的对象回收路径改为delete → (re)new模式，同时将worker的数量调整为一。这样，Free Object Pool就可以更换为无锁实现，解决了锁争抢的问题，实现了约5倍的加速。



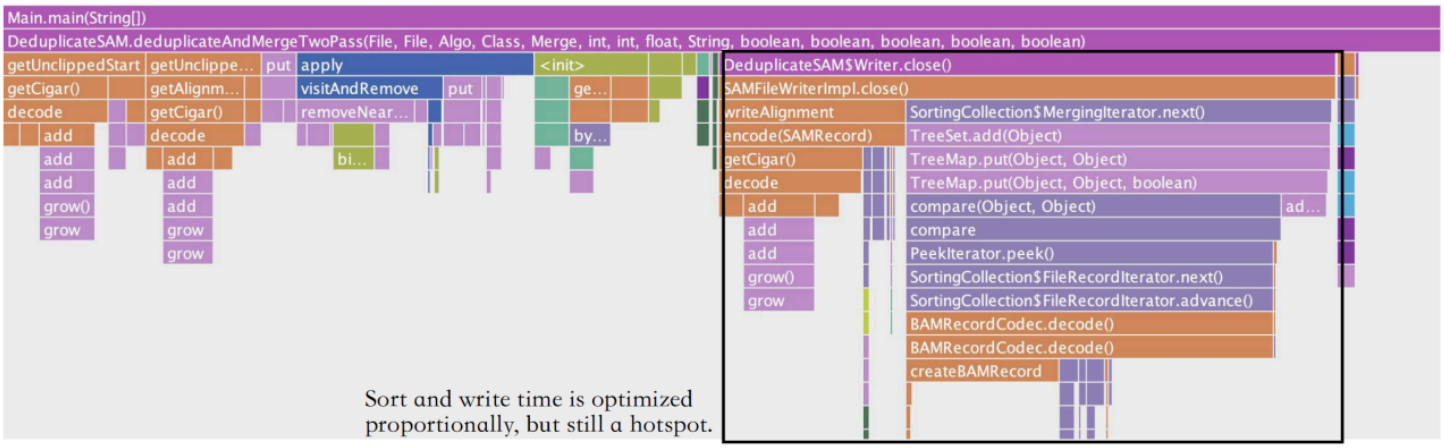
2. 调整并行策略，优化访存
   初步优化后我们发现，对象池相关访存的 LLCMiss 率仍然偏高。为进一步降低访存延迟，我们删除了Position 对象中在后续工作流中不再用到的成员变量，同时减小对象池中的对象数，显著降低了 LLC Miss率,但 L1 Miss率 仍然较高。分析发现，这是由于 Main Thread 与 Worker Thread 交替读写同一内存池，导致核私有的L1缓存行被频繁无效化，即典型的 False Sharing（伪共享） 现象。
   为彻底消除该问题，我们将 Main Thread 与 Worker Thread 合并为单线程执行，同时将并行策略改为 按染色体维度进行数据并行，减少共享对象的跨线程写入冲突，避免 False Sharing，大大降低了L1 Miss率。优化后实现约 6 倍额外加速，结合第一点优化，整个模块在预赛数据集上的执行时间从 30 分钟降至 1 分钟。

Umicollapse IO 与 去重算法优化

在大幅优化 hisat-3n-table 的性能后，回顾DAG，Umicollapse模块成为了新的瓶颈。为了定位问题，我们使用了 JDK Mission Control（JMC） 进行采样分析，结果显示，Umicollapse 在读写 SAM/BAM 时调用的 HTSJDK 库效率很低，是性能瓶颈的源头。

我们在HTSJDK的仓库中，找到了若干未被合并的性能优化相关的PR，并手动在BAMReader中调用了AsyncBufferedIterator，使得BAM文件的读取和解析能够异步进行，同时，我们手动合并了一个异步排序的PR `AsyncWriteSortingCollection` 。此外，我们还对运行时参数进行了一系列调整，如调低输入输出的压缩级别，使用ZGC（Z Garbage Collector）并适当调大堆内存以降低GC频率与开销等，这让端

到端实现了3倍左右的性能提升。然而，JMC的采样结果表明，即使经过了上述优化，IO仍然构成本模块的瓶颈，如图4所示。



在进一步的调研过程中，我们发现了一个使用C++编写的BAM/SAM解析库htslib，在IO性能上较HTSJDK有十几倍的提升。但由于Umicollapse的主体逻辑使用的是Java语言，我们决定对Umicollapse整体用C++进行重写并接入htslib，从根本上解决HTSJDK带来的IO瓶颈问题。

在重写的同时，我们也对去重算法进行了优化。原始代码的逻辑是，在读到某个对齐位置的最后一条记录后，对这个对齐位置的所有 reads 去重。我们实现了 mark dedup，具体操作是，第一遍扫描只标记重复，第二遍再删除重复，这样线性地访存能更好的利用Cache。此外，我们还采用了等距码编码碱基对；并查集代替map等细节优化，最终较原始实现获得10倍左右的端到端性能提升。

3. 并行方案与现场调度

输入拆分与染色体并行

在对hisat-3n-table优化时，我们就已经开始尝试通过改变并行策略来实现更好的拓展性。随着优化的进行，我们决定将这一想法拓展到整个工作流，我们发现：在 配对 过程中，每一条 read 之间都是独立的，因此理论上可以实现纯线性的扩展。我们选择在不改变 `hista-3n` 代码的情况下，将输入切分成若干份，运行若干 `hisat-3n-align` 进程，最后再做合并。

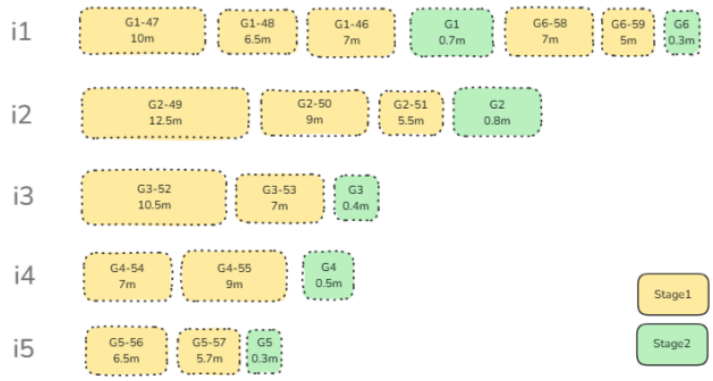
而在完成 配对 后，所有 reads 被对应到不同的染色体（参考基因序列）上，不同染色体之间的 去重 和 统计 又是独立的，因此我们可以通过按染色体切分来实现粗粒度并行，并将每条染色体 去重 和 统计 打通，进一步减少同步开销。整体流程如图5所示。

在实现了以上所有修改后，我们在单机上又获得了2倍的加速比。

现场调度

在决赛赛场上，我们拿到了 6 组共 14 个 case，并使用提前准备好的 文件大小 -> 运行时间 拟合曲线，对每个case进行了耗时预估。

为了在5台机器上实现负载均衡，我们将不同组之间的case打乱，贪心地调整任务分配策略，并利用第一轮实际运行的日志进一步优化调度策略，最终在 24 分钟内完成了所有的 14 个 case。



imbalanced scheduling

greedily rescheduled

4. 结语

在整个赛题的调优过程中，我们选择使用 Git 进行版本管理与协作，方便交叉复现，同步优化思路。点开下面的仓库链接，你可能会看到一些“幼稚”的commit——比如一次失败的cherry-pick尝试，但这或许恰恰是超算的意义所在。

最后，还是欢迎大家访问我们的项目仓库，一起讨论、改进和创新：https://github.com/orgs/asc-rna