**密级:**



**硕士学位论文**

**基于Spark的二代测序数据分析流程的加速研究**

**作者姓名： 李红印**

**指导教师： 谭光明 研究员**

**中国科学院计算技术研究所**

**学位类别： 工学硕士**

**学科专业： 计算机科学与技术**

**研究所： 中国科学院计算技术研究所**

**2016年5月**

**Research of Spark Based**

**NGS data processing acceleration**

A Thesis Submitted to

**The University of Chinese Academy of Sciences**

in partial fulfillment of the requirement

for the degree of

**Master of Science**

in

**Computer Science and Technology**

by

**Hongyin Li**

**Thesis Supervisor: Professor Guangming Tan**

Institute of Computing Technology

Chinese Academy of Sciences

May, 2016

**声 明**

我声明本论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，本论文中不包含其他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

作者签名： 日期：

**论文版权使用授权书**

本人授权中国科学院计算技术研究所可以保留并向国家有关部门或机构送交本论文的复印件和电子文档，允许本论文被查阅和借阅，可以将本论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编本论文。

（保密论文在解密后适用本授权书。）

作者签名： 导师签名： 日期：

摘 要

随着基因测序中测序技术的不断进步，测序数据量增长迅速，原有的基因测序数据分析流程逐渐不能满足这种快速增长的数据需求；而另一方面，计算机领域中大数据技术的持续发展，为处理海量数据提供了稳定、高效的计算框架。在可以预见的未来，人类基因测序数据分析流程必将和大数据的发展不断融合。

本文以基因测序中的二代测序数据分析流程GATK为基础，通过对其流程的原理分析和各个环节的测试，找出阻碍整个流程高效运行的瓶颈。针对整个流程中存在的不同瓶颈，我们提出了使用大数据计算框架Spark作为加速整个流程的基础；通过详细学习和分析GATK流程中的各个环节的算法原理和具体实现，我们对不同环节提出不同的改写方法。最终，通过对整个流程的改写，在保证正确性的前提下，整个流程的运行速度和扩展性有了极大提升。

本文的主要工作和贡献如下：

1. 提出了基于大数据计算框架Spark来加速二代测序数据分析流程。通过对二代测序数据分析流程GATK的测试和原理分析，确定其瓶颈所在，然后针对其瓶颈问题提出了使用Spark来加速的方法。
2. 数据预处理算法的改写。数据预处理是GATK流程的开始部分，由于在原有流程中使用了串行算法，耗时较多。通过对算法的并行化改写，极大的提高了其运行速度和扩展性。
3. Scatter-Gather操作的并行优化。GATK流程的后续步骤中使用网格计算的方法实现了并行加速；但是，由于其每个步骤都需要不断的切割合并（Scatter-Gather）文件，造成了不必要的浪费。通过对Scatter-Gather操作的并行优化，消除了其瓶颈问题。

**关键词：**基因测序，大数据，二代测序，GATK，Spark，并行优化

**Abstract**

Along with the advance of gene sequencing technology and the rapid growth of the sequencing data. The original genetic sequencing data processing process gradually cannot meet the demand of the rapid growth of data; On the other hand, in computer field, big data technology continues to develop, to deal with huge amounts of data to provide a stable and efficient computing framework. In the foreseeable future, the human genome sequencing data analysis process and the development of the large data will fusion.

In this paper, we use the NGS sequencing data analysis process GATK as the foundation, through analyzing the principle of the process and do test for each step, find out the bottleneck of the obstacles to whole process run efficiently. For different bottleneck existed in the work of the entire process, we proposed to use big data computing framework Spark as the basis of accelerating the process; through detailed study and analysis in the GATK process each step of the principle of the algorithm and the concrete implementation, we proposed different methods for different step. Finally, through the rewriting of the whole process , on the premise of guarantee the accuracy, the whole process of running speed and scalability are greatly promoted.

In this paper, the main work and contributions are as follows:

(1) Proposing to use the big data framework Spark to speed up the next generation sequencing data analysis process. Through the theory analysis and test of the next generation sequencing data process GATK, determine its bottlenecks, then put forward the method to accelerate the Spark for the bottleneck.

(2) Rewrite of the data pre-processing algorithm. Data pre-processing is the beginning of the GATK process, it used the serial algorithm in original process, so it was time consuming. Through using the parallel algorithm, greatly improved the speed and scalability.

(3) The parallel optimization of the Scatter-Gather operation. GATK process used grid computing method to implement the parallel acceleration; However, due to its every step needs cutting and merger (Scatter - Gather) file, causing unnecessary waste. Through the parallel optimization of Scatter-Gather operation, eliminating the bottleneck problem.

**Key words:** Gene Sequencing, Big Data, Next Generation Sequencing, GATK, Spark, Parallel Optimization

目录

**[摘 要 I](#_Toc582)**

**[目录 V](#_Toc11957)**

**[图目录 VII](#_Toc28670)**

**[表目录 IX](#_Toc3025)**

**[第一章 引言 1](#_Toc16127)**

[1.1 基因测序技术的发展 1](#_Toc24689)

[1.2 大数据处理技术的发展 3](#_Toc7769)

[1.3 研究动机 5](#_Toc17494)

[1.4 本文主要贡献 6](#_Toc8182)

[1.5 本文组织结构 7](#_Toc7748)

**[第二章 背景与相关研究 9](#_Toc16892)**

[2.1 二代测序数据分析流程 9](#_Toc17499)

[2.2 GATK典型流程 10](#_Toc22622)

[2.2.1 序列比对 11](#_Toc24546)

[2.2.2 数据清理 12](#_Toc32731)

[2.2.3 多样性发现 13](#_Toc14547)

[2.3 基于硬件的优化 13](#_Toc19955)

[2.4 流程并行优化 14](#_Toc28267)

[2.5 本章小结 16](#_Toc19876)

**[第三章 GATK典型流程的分析和评估 17](#_Toc3463)**

[3.1 流程的相关测试 17](#_Toc5865)

[3.2 流程分析及算法详解 19](#_Toc14995)

[3.2.1 Mark Duplicates&Sort 20](#_Toc32592)

[3.2.2 Indel Realignment 21](#_Toc5211)

[3.2.3 Base Recalibration 22](#_Toc5155)

[3.3.4 Haplotype Caller 23](#_Toc24064)

[3.3 瓶颈分析 24](#_Toc21241)

[3.3.1 大文件的切分合并 24](#_Toc12755)

[3.3.2 Picard中串行步骤 25](#_Toc31654)

[3.3.3 GATK中Scatter-Gather 25](#_Toc19444)

[3.4 平台选择 25](#_Toc28395)

[3.5 本章小结 26](#_Toc13015)

**[第四章 GATK-Spark的设计与实现 27](#_Toc26445)**

[4.1 平台架构与整体设计 27](#_Toc29995)

[4.2 数据交换的优化 28](#_Toc1533)

[4.3 预处理算法的并行优化 30](#_Toc12211)

[4.4 Scatter-Gather的并行 33](#_Toc15294)

[4.4.1 Spark版Indel Realignment 33](#_Toc25351)

[4.4.2 Spark版Base Recalibration 35](#_Toc5666)

[4.4.3 Spark版Haplotype Caller 37](#_Toc8290)

[4.5 本章小结 39](#_Toc31644)

**[第五章 性能评测及实验分析 41](#_Toc3562)**

[5.1 实验配置 41](#_Toc30011)

[5.2 实验结果与分析 42](#_Toc1548)

[5.2.1 结果正确性测试 42](#_Toc23295)

[5.2.2 性能测试 43](#_Toc29411)

[5.2.3 加速比分析 46](#_Toc8239)

[5.2.4 其它测试 49](#_Toc19917)

[5.3 本章小结 51](#_Toc10017)

**[第六章 结束语 53](#_Toc11406)**

[6.1 本文工作总结 53](#_Toc62)

[6.2下一步工作 53](#_Toc20664)

**[参考文献 55](#_Toc22106)**

**[致谢 i](#_Toc4917)**

**[作者简介 iii](#_Toc6802)**

# 图目录

[图 1.1 第一代DNA测序：荧光自动测序技术 1](#_Toc28488)

[图 1.2 第二代DNA测序：Illumina公司最新的HiSeq 4000 2](#_Toc3356)

[图 1.3 第三代DNA测序：新型纳米孔测序法 3](#_Toc69)

[图 1.4 Hadoop中MapReduce原理图 4](#_Toc18340)

[图 1.5 Spark RDD操作示例图 5](#_Toc28392)

[图 1.6 测序数据量随时间的变化 6](#_Toc17182)

[图 2.1 DNA测序genotype-calling非商业软件 9](#_Toc7344)

[图 2.2 RNA测序分析工具 10](#_Toc29322)

[图 2.3 GATK的典型流程 10](#_Toc18240)

[图 2.4 GATK的典型流程的分类表示 11](#_Toc18525)

[图 2.5 字符串googol的后缀数组和轮排索引 12](#_Toc31054)

[图 2.6 PCR扩增过程 12](#_Toc20098)

[图 2.7 Churchill中切分BAM的示意图 15](#_Toc699)

[图 3.1 GATK-Queue的流程示意图 17](#_Toc32671)

[图 3.2 GATK-Queue运行资源利用率 18](#_Toc15363)

[图 3.3 GATK-Queue各部分运行时间比例 18](#_Toc17817)

[图 3.4 GATK-Queue各步骤CPU Time和IO Time占比 19](#_Toc6360)

[图 3.5 Picard中Sort的原理图 20](#_Toc16730)

[图 3.6 Picard中Mark Duplicates的原理图 21](#_Toc19978)

[图 3.7 GATK中Indel Realignment流程图 21](#_Toc16338)

[图 3.8 GATK质量分数重校正前后的结果对比 22](#_Toc3754)

[图 3.9 GATK中Haplotype Caller原理图 23](#_Toc59)

[图 3.10 GATK-Queue瓶颈分析 24](#_Toc9327)

[图 4.1 Spark的架构图 27](#_Toc12194)

[图 4.2 GATK-Queue到GATK-Spark的转化 28](#_Toc26456)

[图 4.3 解决数据交换依赖及分发全局信息示意图 29](#_Toc15779)

[图 4.4 Spark中RDD解决数据依赖原理图 30](#_Toc22675)

[图 4.5 GATK-Spark中区分Pair-End和Fragment序列 30](#_Toc28276)

[图 4.6 GATK-Spark中Pair-End序列标记重复 31](#_Toc25758)

[图 4.7 Spark版Mark Duplicates原理图 32](#_Toc20005)

[图 4.8 两种Spark版的Sort 32](#_Toc4268)

[图 4.9 Spark版Indel Realignment原理图 33](#_Toc26917)

[图 4.10 Spark版Indel Realignment中找出局部区域流程图 34](#_Toc5962)

[图 4.11 Spark版Indel Realignment核心代码 34](#_Toc28169)

[图 4.12 Spark版Base Recalibration原理图 35](#_Toc14402)

[图 4.13 Spark版Base Recalibration计算qualityTable 35](#_Toc7305)

[图 4.14 Spark版Base Recalibration中统计全局信息表流程图 36](#_Toc16320)

[图 4.15 Spark版Haplotype Caller原理图 37](#_Toc9675)

[图 4.16 Spark版Haplotype Caller中Overlap机制 38](#_Toc2227)

[图 4.17 Spark版Haplotype Caller核心代码 38](#_Toc25061)

[图 5.1 GATK-Spark和GATK-Queue结果正确性对比 42](#_Toc32374)

[图 5.2 GATK-Spark和GATK-Queue的cpu利用率 44](#_Toc32413)

[图 5.3 不同运行方式下流程的运行时间随cpu个数的变化 44](#_Toc25008)

[图 5.4 GATK-Spark各个步骤运行时间随cpu个数的变化 45](#_Toc2930)

[图 5.5 不同运行方式下的加速比 46](#_Toc21625)

[图 5.6 GATK-Spark各部分运行时间比例随CPU个数的变化 47](#_Toc1761)

[图 5.7 GATK-Spark各步骤运行时间和并行度随CPU个数的变化 48](#_Toc29118)

[图 5.8 垃圾回收时间随着单cpu内存量的变化 49](#_Toc20705)

[图 5.9 16 cpu下GATK-Spark中map执行图 50](#_Toc29670)

[图 5.10 256 cpu下GATK-Spark中map执行图 50](#_Toc9904)

表目录

[表 2.1 不同硬件方式下Pair-HMM的加速效果 14](#_Toc9540)

[表 3.1 GATK典型流程不同方式下运行时间 18](#_Toc15895)

[表 5.1 单台机器的配置参数 41](#_Toc14230)

[表 5.2 测试数据的相关参数 42](#_Toc23176)

[表 5.3 不同方法得到的多样性位点与GATK-Queue的差别 43](#_Toc25799)

[表 5.4 不同运行方式下流程的详细运行时间 45](#_Toc18683)

[表 5.5 不同数据量下GATK-Spark运行时间 51](#_Toc9524)

# 引言

作为人类三大科学计划之一的人类基因组计划，通过十几年的努力，揭开了人体的30亿个碱基对。自此，基因测序的研究进入了一个快速发展的通道，其在科学、商业以及人类健康方面发挥越来越重要的作用。近十年来，随着测序技术的不断进步，测序数据量呈现快速增长的趋势，原有的测序数据分析流程逐渐不能满足日益增长的数据量的需求，因此，需要我们提出一种新的基因数据处理的方法。

## 1.1 基因测序技术的发展

基因测序又叫DNA测序，是现代分子生物研究中常用的技术，从其技术的发展来看，可以分为三个阶段：第一代DNA测序技术、第二代DNA测序技术和第三代DNA测序技术。

通过传统的化学降解法、双脱氧链终止法以及在它们的基础上发展来的各种DNA测序技术统称为第一代DNA测序技术，第一代DNA测序所用到的技术包括：化学降解法、双脱氧链终止法、荧光自动测序技术和杂交测序技术等。第一代DNA测序技术在分子生物学研究领域中发挥过重要的作用，包括像人类基因组计划在内的早期分子生物学研究领域主要基于第一代DNA测序技术。目前基于荧光标记和Sanger的双脱氧链终止法原理的荧光自动测序仪仍被广泛的应用[1]。

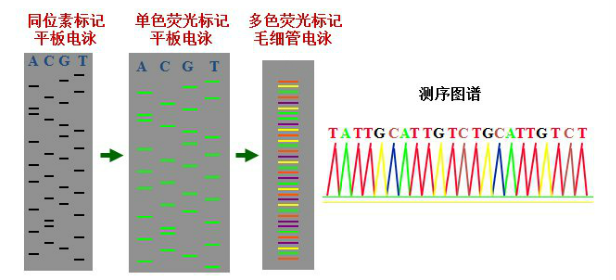


图 1.1 第一代DNA测序：荧光自动测序技术 [1]

随着人类基因组计划的完成，人们进入了后基因组时代，原有的测序技术和方法已经不能满足大规模基因测序的需求，这使得新一代DNA测序技术的产生，也就是第二代DNA测序技术。第二代DNA测序技术又叫二代测序（NGS）或高通量测序，其典型的特点有两个：吞吐量大和读长短。吞吐量大是指一次能并行的对几十万到几百万条DNA分析进行序列测定，读长短是指其测得的序列是很多条read，每条read长度在100-300bp之间。二代测序由于其一次可以对几万到几百万条DNA分子进行序列测序，使得对于一个物种的转录组的测序或者基因组的深度测序变得更加方便、更加容易，也正是这个原因使得其在商业化的道路上走得异常顺利。

从2005年Life Sciences公司发布其454 FLX平台开始，近十几年，二代测序的测序平台商业化主要被三家公司所占领，包括Applied BioSystem（ABI）公司的SOLiD平台、罗氏公司（Roche）公司的GS FLX 454平台以及Illumina公司推出的Solexa基因组分析平台。以近来发展最好的Illumina公司来看，其从2005年单次测得10万个标记物到如今单次18000亿个标记物的规模，这其中在数据量上的跨越，是极其夸张的。  


图 1.2 第二代DNA测序：Illumina公司最新的HiSeq 4000 [2]

第三代DNA测序技术是以单分子测序为特点的测序技术。目前第三代DNA测序技术在原理上主要分为两大阵营：第一阵营是单分子荧光测序，通过显微镜实时观察记录用荧光标记的脱氧核苷酸；另外一个阵营是纳米孔测序，通过电泳技术来驱动单个分子逐一通过纳米孔来实现测序。第三次DNA测序的产生主要是针对第二代测序在测序的速度、成本和准确度等问题而来，目前由于其在测序通量和成本上的问题，依旧处于研究发展阶段。

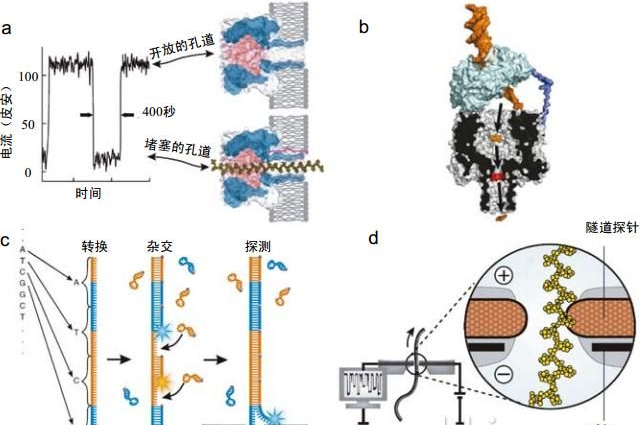


图 1.3 第三代DNA测序：新型纳米孔测序法 [1]

经过近三十年的发展，DNA测序技术跨越了三代，而这三代测序技术有其不同的优缺点。第一代DNA测序技术由于其成本高、速度慢的特点没有大规模的应用，不过对于少量序列测序来说依旧是一个不错的选择；第二代DNA测序技术伴随着其商用化的发展，正在不断的发展成熟；第三代DNA测序技术目前还处在起步阶段，距离其稳定成熟并大规模商业化还有很大的距离。总得来说，目前阶段还是三代DNA测序技术共存，第二代DNA测序技术为主的阶段。对于第二代测序来说，随着技术的不断进步，测序的通量不断提高，相应的配套数据处理流程逐渐不能与之匹配，对于二代测序数据分析流程的加速就势在必行。

## 1.2 大数据处理技术的发展

近年来，随着大数据概念的提出，其在科研领域、工业界甚至是大众生活的各个方面产生越来越深刻的影响[3]。本文中我们主要关注的是大数据技术中的大数据处理技术，也就主要是分布式计算框架。

大数据处理技术的起源要从谷歌的三篇论文说起，从2003年到2006年这三年中，谷歌发表了三篇非常有影响力的文章，分别是2003年发在SOSP的The Google File System[4]、2004年发在OSDI的MapReduce: Simplified Data Processing on Large Clusters[5]和2006年发在OSDI的Bigtable: A Distributed Storage System for Structured Data[6]，之后开源社区借鉴了谷歌论文中的方法，开发了大名鼎鼎的hadoop，自此大数据处理技术的发展如雨后春笋般席卷开来。

Hadoop是一个开源的可运行在大规模集群上的分布式并行编程框架，其最核心的两个部分是：HDFS和MapReduce，HDFS为海量数据提供了分布式存储，MapReduce为海量数据提供了分布式计算。通过使用hadoop，可以很容易的编写处理海量数据的分布式并行程序，并将其运行在成千上万的节点组成的集群上。

基于MapReduce的编程框架使得编写分布式程序变得简单，只需要设计和实现Map和Reduce的相关部分即可。并行编程中的各种问题，如网络通信、负载均衡、作业调度、容错处理以及分布式存储，统统交给HDFS和MapReduce来负责处理。我们只需要关注业务逻辑的实现即可，不再用关注底层的各种问题和实现，即可以写出高效的复杂的分布式并行程序。

MapReduce的原理相对来说也比较简单，其核心步骤主要是两部分：Map和Reduce。当我们向MapReduce提交一个任务时，它首先会把任务分拆成很多个小的任务分配到不同节点去执行，这也正好是map task阶段完成的。当map阶段完成，会生成很多个小的中间文件，这些中间文件将会作为reduce阶段的输入被reduce读取并执行reduce阶段的任务，reduce也可以有很多个任务，最后汇总所有结果，完成整个流程。

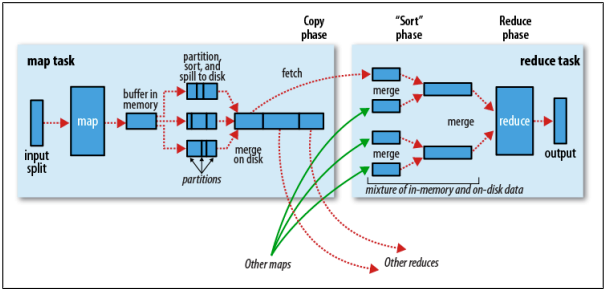


图 1.4 Hadoop中MapReduce原理图 [32]

Hadoop的出现极大的方便了人们在大规模并行计算上的编程，但是它也存在一定的局限和不足：操作少，hadoop只有map和reduce两个操作，表达力非常差，需要将很多逻辑转化为map reduce的形式；中间结果存放在HDFS上，迭代数据处理性能差；reduce阶段需要所有map任务结束之后才能开始。因此在hadoop推出之后，出现了很多对其局限和不足进行改进的研究，其中最有名也是当前最有前景的当属Spark。2009年，Spark诞生在伯克利大学的AMPLab。Spark作为一个研究性质的项目，于2010年开源，在2013年成为Apache基金项目，并且在2014年成为Apache基金的顶级项目，短短几年迅速发展壮大。

Spark是一个基于内存计算的分布式计算框架，其主要特点就是提供了一个集群的分布式内存抽象，这个抽象就是RDD（Resilient Distributed Dataset），它表示不可变的带分区信息并能够并行操作的数据集合。RDD也是Spark中的编程模型，通过在其上的操作来实现分布式计算，这些操作主要分为两类：转换和动作，包括map、reduce、union、groupBy、join、sortBy等在内的一百多个操作。所有这些转换都是惰性计算的，通过有向无环图（DAG）来构造出依赖关系，确定阶段（Stage）、分区（Partition）、流水线（Pipeline）、任务（Task）和缓存（Cache），从而来实现整个计算过程的优化。由于每次迭代的数据都在内存中（当内存不足是溢出到本地硬盘），使得迭代数据处理的性能相对hadoop提高很多。

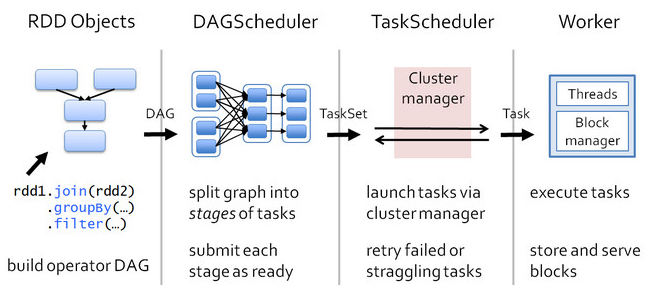


图 1.5 Spark RDD操作示例图 [30]

从谷歌的论文和开源社区的hadoop实现为大规模集群计算提供了简单有效的编程框架到后来基于内存计算的Spark的迅速发展壮大，大数据处理技术的不断进步和发展为大规模计算机集群快速高效的编程和运用提供了基础，也为我们处理大规模数据提供了保障。

## 1.3 研究动机

随着基因测速技术的不断发展，测序技术从第一代发展到如今的第三代，测序数据量呈现一种快速增长的趋势。一方面，测序技术的进步使得单次单样本的测序价格迅速下降，另一方面，为了得到更加精确的数据，测序的样本覆盖率也在不断提升，这就使得所需要处理的数据的规模越来越大。

图 1.6 测序数据量随时间的变化

测序数据量的迅速增长，而相应的数据处理分析流程软件却依旧停留在早期的处理方法的阶段，处理的并行度不高、资源利用率差，这也造成了处理速度不能与测序数据量的不断发展相配套。另一方面，随着大数据概念的普及以及大数据处理技术（分布式计算）的快速发展，其对于大量数据的处理已经显现出不俗的能力。

利用大数据处理技术的方法处理基因测序数据分析流程，这为我们更加快速高效的得到基因测序的结果，为疾病预防和诊断提供及时有效的评测。未来，大数据技术在基因测序数据处理中的作用将越来越重要。

## 1.4 本文主要贡献

本文从基因测序的快速发展切入，通过对一个全基因测序（WGS）数据分析流程GATK典型流程的具体分析和评测，找出导致整个流程并行度差、运行时间长的原因。本文针对原有流程存在的问题，提出了以Spark计算框架改写其中主要步骤的方法，通过改写和迁移的方式，使得整个流程高效稳定的运行在Spark计算框架上，在保证正确性的前提下，提高了整个流程的并行度，大大降低了流程的运行时间。论文的主要工作和贡献如下：

(1) 提出了使用大数据计算框架Spark来加速二代测序数据分析流程。通过对二代测序数据分析流程GATK的测试和原理分析，确定其瓶颈所在，然后针对其瓶颈问题提出了使用Spark来加速的方法。

(2) 数据预处理算法的改写。数据预处理是GATK流程的开始部分，由于在原有流程中使用了串行算法，耗时较多。通过对算法的并行化改写，极大的提高了其运行速度和扩展性。

(3) Scatter-Gather操作的并行优化。GATK流程的后续步骤中使用网格计算的方法实现了并行加速；但是，由于其每个步骤都需要不断的切割合并（Scatter-Gather）文件，造成了不必要的浪费。通过对Scatter-Gather操作的并行优化，消除了其瓶颈问题。

## 1.5 本文组织结构

本论文总共分为六章，具体组织结构如下：

1. 作为本文的引言，通过对基因测序技术和大数据处理技术的介绍，为本文的研究动机和研究意义做了一个很好的诠释，另外还介绍了本文的主要贡献。
2. 主要介绍背景和相关研究，通过多二代测序流程的大致了解，选择了GATK典型流程作为我们并行加速的重点，经过对GATK典型流程的介绍和现在已有的一些关于GATK并行加速的相关研究为我们后续的并行加速提供基础。
3. 重点是对GATK典型流程的详细分析和评估，通过对GATK-Queue方式的相关测试和各个步骤详细算法的探讨，找出阻碍其性能提升的瓶颈所在，然后提出了基于Spark改写的方法。
4. 是GATK-Spark系统的设计与实现，经过我们整体的一个考量，提出了不同步骤使用不同改写的方法，对各个步骤进行加速优化，最终得到可以在Spark上运行的GATK典型流程。
5. 进行了性能评测和实验分析，通过正确性、性能和加速比方面的测试工作和分析，更加清楚的证明了GATK-Spark在性能和扩展性方面的优势。
6. 是总结和展望，总结了整个课题的情况，并提出在未来可能的研究方向。

# 背景与相关研究

基因测序得到的碱基序列需要通过各种数据分析流程来获得我们需要的各类信息，而这些流程中GATK的典型流程是非常有名而且也非常重要的一个流程。随着基因测序技术的不断发展，数据量的快速增长也促使一些对于GATK的并行加速的研究的出现，但是这些并行加速的研究由于各种各样的问题，目前尚没有一个成熟稳定并且高效的并行方法。

## 2.1 二代测序数据分析流程

二代测序通过荧光和生物试剂等手段获得人类遗传物质基因的碱基序列，但由于这些获得的信息是大量片段性质的，而且存在一定误差，因此想要准确完整的确定其所处位置和真实的碱基类型需要我们使用计算机的手段来进行分析，这些分析就是我们通常使用的基因测序数据分析流程[7]。

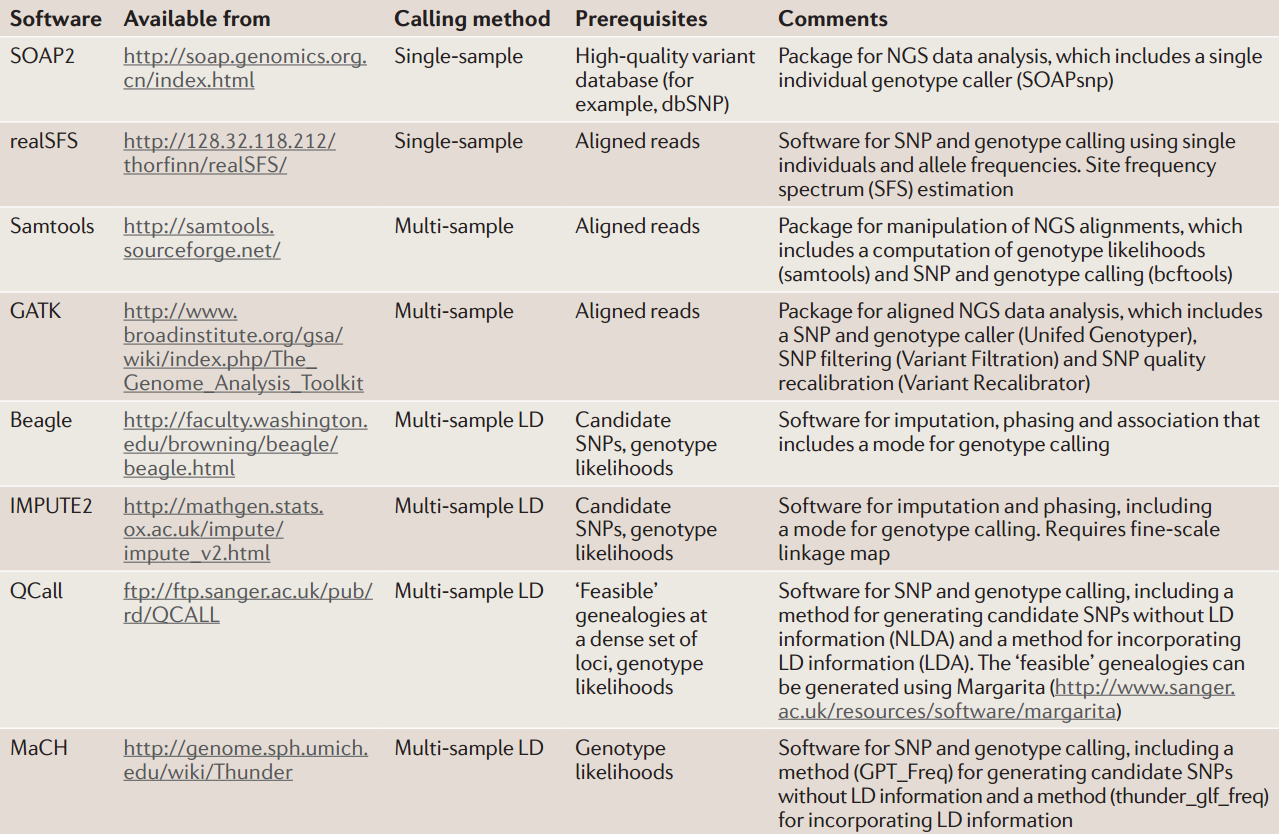


图 2.1 DNA测序genotype-calling非商业软件 [8]

二代测序主要分为DNA测序和RNA测序，在不同的测序数据类型中，针对不同的用途和目的也有很多不同的数据分析流程。图2.1列出了在DNA测序中针对genotype-calling的非商业软件[8]，图2.2列出了RNA测序中的一些分析工具，当然对于其他不同用途，还有很多不同的分析流程[9]。

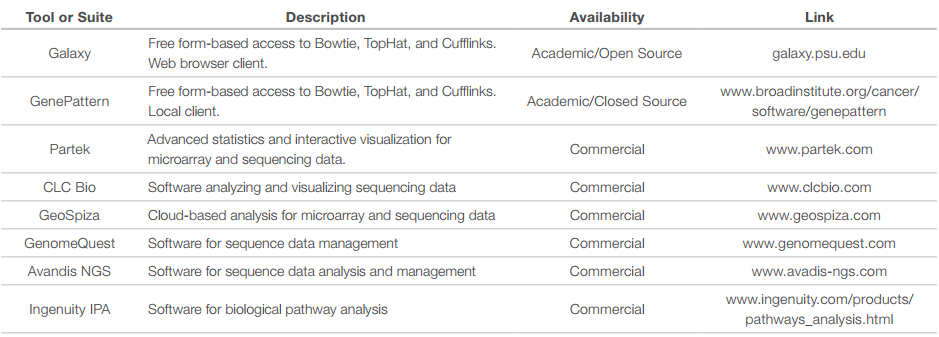


图 2.2 RNA测序分析工具 [9]

随着测序技术和计算技术的进步，基因测序的作用逐渐显现，未来的前景也越来越被人们看好，所以对于基因测序数据分析流程的研究也不断增多。目前，由于基因测序还处在初级阶段，测序数据的获得和配套的数据分析流程还不够成熟，所以关于基因测序数据分析流程的方法和手段还在探索中，没有一定的标准，但是在不同的应用中还是出现一些被大家所认可的佼佼者。在RNA测序中，TopHat和Cufflinks基本成为RNA-Seq中必备的步骤[10]。在DNA测序中，GATK被广泛应用到1000 Genomes Project[11]和Cancer Genome Atlas等国际化大项目中，而且被各大研究所和一些基因公司广泛使用[12]。

本文我们主要针对GATK的数据分析流程来进行分析，找出阻碍其运行速度的瓶颈，通过改进和修改其中的一些步骤来实现对其并行加速的目的。

## 2.2 GATK典型流程

GATK（The Genome Analysis Toolkit）[13]是由Broad Institute开发的一套发现多样性位点的工具集，它主要是为了来发现DNA测序和RNA测序数据中的SNP（Single Nucleotide Polymorphisms）和indel（insert&delete）[14]。除了多样性位点的发现之外，GATK中还包括一些相关的工具，包括预处理工具和控制测序数据质量的工具。

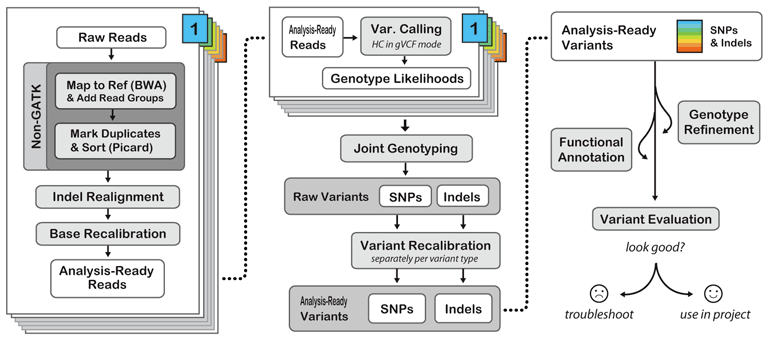


图 2.3 GATK的典型流程 [13]

通过图2.3可以看到GATK的典型流程主要由7个步骤组成[14]，根据各个步骤的功能大概可以把整个流程分成三个部分（见图2.4）。第一部分是序列比对，主要由BWA完成；第二部分是数据清理，主要包括Mark Duplicates、Sort、Indel Realignment和Base Recalibration这几个步骤；第三部分是多样性发现，主要由HaplotypeCaller以及其后的Joint Genotyping和Variant Recalibration。

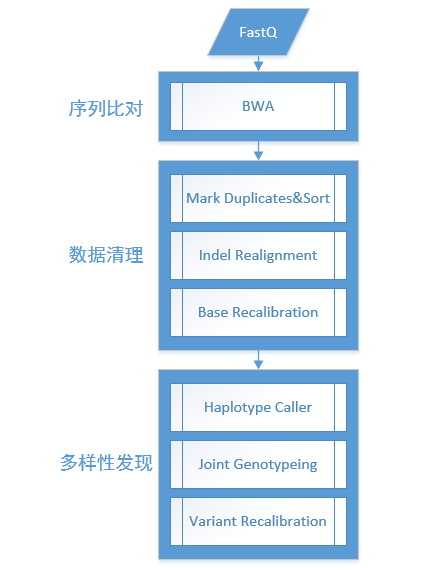


图 2.4 GATK的典型流程的分类表示

接下来我们对各个部分中的每个步骤进行详细的描述来充分认识GATK典型流程的原理和功能。

### 2.2.1 序列比对

DNA测序中通过测序仪我们得到了大量的成对的DNA分子片段（长度大约100-300bp），接下来我们首先要做的就是把这些片段比对到参考序列上。参考序列是通过人类基因组计划测得的人类的基因序列的参考标准。比对的软件有很多，主要是以Seed Methods和Burrows-Wheeler transform Methods两种方法为基础[15]，在GATK典型流程中使用的BWA就是Burrows-Wheeler transform Methods的一种实现。

BWA使用的是通过轮排索引压缩索引数据进行搜索的方式来进行比对，图2.5中列出了字符串googol建立的后缀数组和轮排索引字符串[16]。BWA本身带有多线程的加速方式，另外由于其计算密集的特点，对于其进行并行加速可以作为一个专门的课题来研究，所以在本文中，我们以BWA的结果为起始来做并行加速。

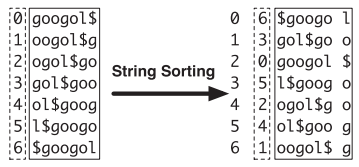


图 2.5 字符串googol的后缀数组和轮排索引 [16]

### 2.2.2 数据清理

由于在测序过程中得到的数据是原始的碱基序列，因此想要用到我们的数据分析流程中之前需要进行数据清理，GATK典型流程中数据清理部分主要包括三方面的内容：重复标记、indel局部重新比对和碱基质量重校正。

测序时在制备文库过程中，PCR的扩增会产生一些偏差，这样序列就会被过量扩增，图2.6说明了PCR的扩增过程[17]。在比对中，这些扩增的序列就会比对到同一个位置，但是这些扩增的序列并不是基因组本身固有的，因此不能作为检测变异的证据，所以我们需要去除掉这些重复的序列，这个过程就叫做Mark Duplicates。这一步主要通过使用picard软件来完成，把序列中的duplicate flag设置为true来表示重复的序列。

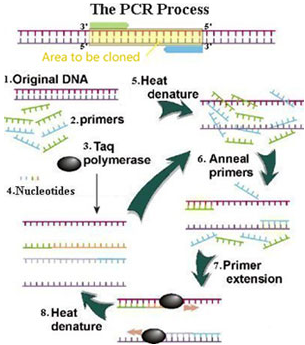


图 2.6 PCR扩增过程 [17]

一般来说，需要进行重新比对的基因组区域，都是因为有indel的存在，因为在indel附近的序列比对可能会出现很多碱基错配，这些碱基错配很容易被认为是SNP。另外，在序列比对中，比对算法是对每一条序列单独进行比对的，不可能把多条序列组合在一起和参考序列进行错误纠正。因此就需要我们进行indel局部重新比对，通过对由indel导致错配的区域进行重新比对，来实现降低indel附近的错误率，这个过程就是Indel Realignment的主要工作。

在后续的多样性发现中，我们主要使用质量分数在Q25之上的碱基，但实际上质量分数在Q25的碱基的错误率在1%左右，也就是说质量分数只有Q20，这样会对我们后续的变异检测的可信度产生影响；另外，测序过程是边合成边测序的，在序列末端的碱基错误率要比起始端高很多；再加上不同的碱基的出错率的不同，A（腺嘌呤）C（胞嘧啶）的质量分数往往低于T（胸腺嘧啶）G（鸟嘌呤）[18]。因此对碱基的质量分数的校正就非常有必要，Base Recalibration主要完成的就是这项工作。

数据清理是变异检测的基础，通过对数据的修剪和校正，删减掉冗余和可信度不高的数据，修正了数据中可能存在的错误，为后续的变异检测的准确性提供了高质量的可靠保障。

### 2.2.3 多样性发现

多样性发现是整个GATK典型流程的核心，主要包括Haplotype Caller及其后的Joint Genotyping和Variant Recalibration，通过对比对并且清理后的序列数据与参考序列之间的分析评估，找出可能的变异位点，并对这些变异位点进行详细的校正和分析[19]。

Haplotype Caller是整个GATK典型流程中最重要也是最复杂的一步，由于我们得到的序列是片段的，而且其数据存在一定的错误率，所以找出变异位点的方法也需要以一种不确定的方式。GATK的Haplotype Caller中主要使用了Pair-HMM的方法来对位点进行评估，确定变异位点的可能性。随后的Joint Genotyping主要把超过一定阈值的位点（也就是变异位点）进行合并，过滤掉正常位点，最后通过Variant Recalibration来对变异位点进行校正，并确定SNP位点和indel位点以及它们相对应的得分[14]。

## 2.3 基于硬件的优化

GATK典型流程中存在一些计算密集的步骤或算法，这些步骤或算法耗时久并且计算量非常大，通过多线程或者多核的加速方式虽然能在一定程度上改善其时间消耗，但是随着各步骤的并行度的提升，这些部分依旧是整个流程的瓶颈所在，所以使用硬件的加速方式来提高其性能就成为一个必然的方向，这种密集计算的步骤或算法中最重要的两个要数BWA和Pair-HMM了。

早在2012年，剑桥大学的Petr Klus等人就开发了BarraCUDA[20]，通过将比对算法改写成CUDA的处理方式，使用GPU来加速比对过程，在时间上获得了很大的提升，并且一定程度大大节省了CPU资源的使用。

Pair-HMM作为一个计算量非常大的部分，GATK开发团队从一开始就提出了对这部分加速的想法。通过与来自Green Mountain Computing Systems的Scott Thibault的合作实现了在FPGA上计算Pair-HMM的代码，实现了高速的IO带宽和高的并行度；与来自IMPA-RJ的Diego Nehab合作完成了在GPU上的算法实现，除了高的并行度外还实现了快速的浮点精度计算；通过与Intel公司合作实现了使用AVX指令完成Pair-HMM计算的方式，实现了在原有资源的基础上对性能的提升的方式。可以从表2.1中看到不同方式对于Pair-HMM算法加速的效果。

表 2.1 不同硬件方式下Pair-HMM的加速效果 [21]

|  |  |  |  |
| --- | --- | --- | --- |
| TECH | Hardware | Runtime  (seconds) | Improvement  (fold) |
| AVX | Intel Xeon 24-core | 15 | 720x |
| GPU | NVidia Tesla K40 | 160 | 67x |
| GPU | Nvidia GeForce GTX Titan | 161 | 67x |
| GPU | Nvidia GeForce GTX 480 | 190 | 56x |
| GPU | Nvidia GeForce GTX 680 | 274 | 40x |
| GPU | Nvidia GeForce GTX 670 | 288 | 38x |
| AVX | Intel Xeon 1-core | 309 | 35x |
| FPGA | Convey Computers HC2 | 834 | 13x |

## 2.4 流程并行优化

除了在一些密集计算步骤上的并行加速外，对于整个流程的并行优化也有一定的研究，但是这些并行优化的方法都存在着一些问题。

GATK本身自带了Queue的加速方式，通过网格计算的方式，将大的输入文件切分成很多个小的文件，分配到不同的机器上运行，最后收集所有这些结果进行合并作为下一个步骤的输入[14]。将大任务切分成多个小任务的方式在一定程度上提升了整个流程的并行度，但是还是存在并行不够彻底的问题。首先，Mark Duplicates&Sort是picard中的软件，没有Queue的运行方式；另外，由于其不断拆分合并文件的过程，而且这个过程的串行进行的，因此会成为流程并行加速的一个新的瓶颈。

2011年，芬兰阿尔托大学的Matti Niemenmaa基于picard的JDK开发了Hadoop-BAM[22]。Hadoop-BAM是一个可以在分布式计算框架hadoop上方便快捷操作二代测序数据的软件，并且其中还加入了一些简单的操作，例如Sort。但是由于其方法和实现上的原因，Hadoop-BAM的功能太简单并且扩展性差。

随着Spark的产生和发展，加州大学伯克利分校的Matt Massie基于Hadoop-BAM开发了ADAM[23]。ADAM是在Spark中处理二代测序数据格式的一款软件，提供了在Spark中存储及操作的数据格式，另外还提供了一些预处理相关的操作。在ADAM的后续版本中，提供了发现多样性位点的avocado，但是，由于ADAM中的操作完全重写了与GATK相关的算法，而且并未与GATK保持一致性，结果也并未得到验证，还存在很大的问题。

2015年，Benjamin J Kelly等人基于切分区间的方式开发了Churchill [24]。基于区间的切分方式与GATK-Queue中切分的方式类似，如图2.7所示，但是Churchill的切分从序列比对阶段就开始了，通过中间阶段汇总一些全局性的信息，使得切分后的处理方式尽量接近不切分时的处理方式。Churchill的处理方式相对于GATK-Queue部分切分和HugeSeq[25]基于染色体的切分有了非常大的进步，性能的提升也非常大，但是由于其切分区间的方式太过直接，并未增加overlap进行拟补，所以当计算资源扩展时区间的切分非常多，对结果的影响也会相应增加。另外，Churchill的关键处理部分的代码没有开源，对其应用的广泛性会大大受影响。

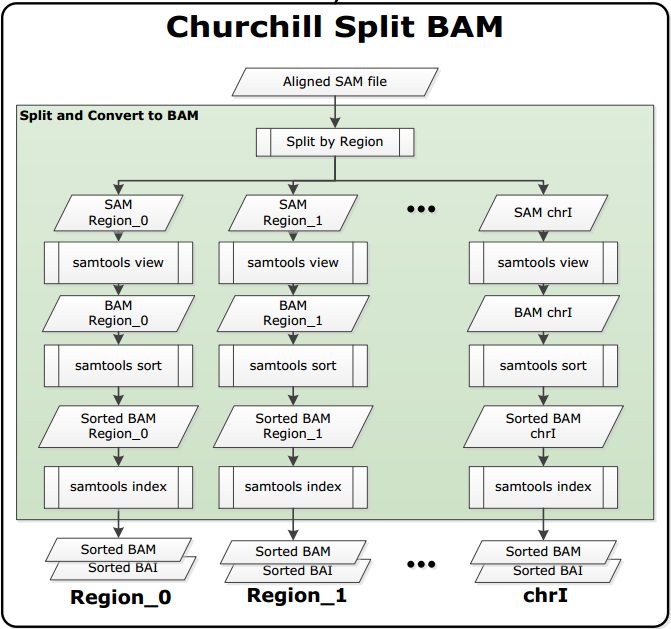


图 2.7 Churchill中切分BAM的示意图 [24]

## 2.5 本章小结

本章以各种二代测序数据分析流程切入，通过对目前二代测序方面存在的各种流程的简要介绍，对二代测序数据分析流程有了大概的了解。随后深入的介绍和分析了本文主要关注的GATK典型流程，对其三个部分序列比对、数据处理和多样性发现进行了详细描述和说明，使得对GATK典型流程有了粗略的认识。接下来介绍了目前阶段已有的一些与GATK典型流程相关的一些优化研究，为之后我们选择GATK典型流程的优化方法提供了帮助。

# GATK典型流程的分析和评估

GATK典型流程是GATK开发团队为使用者提供的一个经常使用且非常行之有效的典型流程。在GATK典型流程的使用中，为了更为快速的得到运行结果，一般会使用GATK的Queue方式，如图3.1所示，GATK-Queue主要是将一些步骤划分成多个小的任务进行网格计算的方式。本章节我们主要通过分析GATK-Queue运行时的各类指标，找出阻碍GATK典型流程并行加速的瓶颈所在。本文中我们主要关注序列比对后的流程加速，关于序列比对的并行加速是另外一个较为重要的课题。

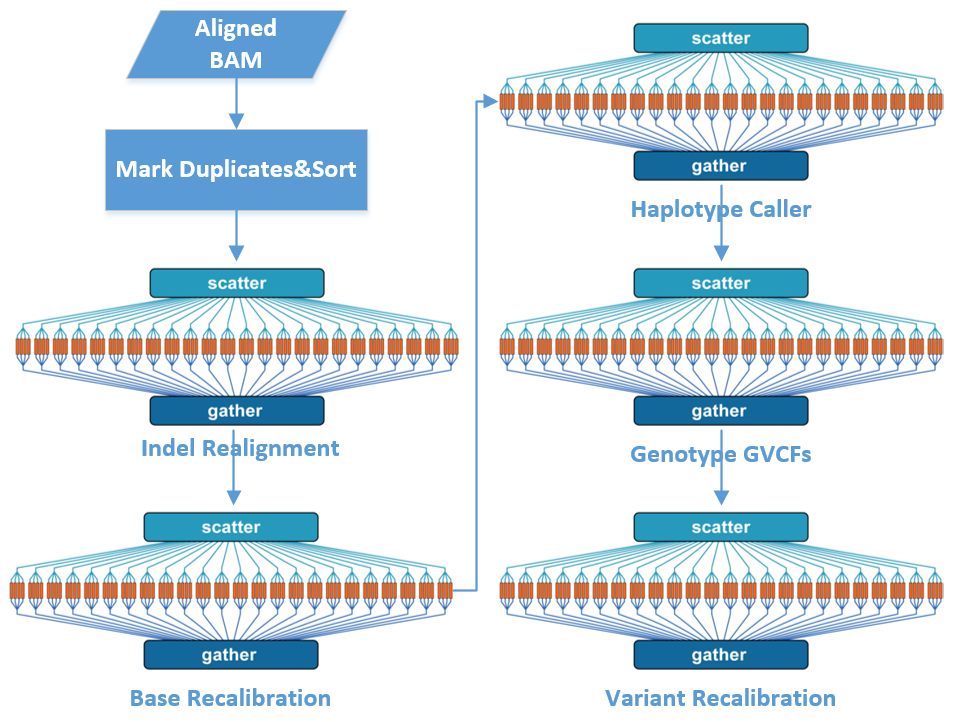


图 3.1 GATK-Queue的流程示意图

## 3.1 流程的相关测试

GATK-Queue是运行在多个节点上的一种运行方式，由于其使用网格计算的模式，所以需要在多个节点之间使用同一个共享的文件目录，我们采用了NFS的方法[26]。另外，为了分发任务及任务调度，GATK-Queue中使用了SGE的任务调度工具。通过对GATK典型流程使用串行方式和Queue的方式进行测试，得到了表3.1中的结果，测试的步骤依据图3.1。根据表格中的结果我们可以看到，GATK-Queue方式相对于串行运行方式来说性能提升非常多，但是，其加速的比例以及扩展性效果并不好。

表 3.1 GATK典型流程不同方式下运行时间

|  |  |  |  |
| --- | --- | --- | --- |
| Serial | GATK-Queue | | |
| 10 cpu | 20 cpu | 40 cpu |
| 19.4 h | 8.2 h | 6.5 h | 5.9 h |

通过使用UC Berkeley发起的开源集群监控软件ganglia，我们统计了在4个节点上使用GATK-Queue时运行的资源利用率，如图3.2所示。根据图示我们可以发现，在GATK-Queue的运行任务中，大多时候计算任务是串行执行的，只有少部分任务的资源利用率比较高，而且这部分任务的占比相对较少。由此可以看出，GATK-Queue运行方式的并行度不高，其扩展性也难于达到好的效果。

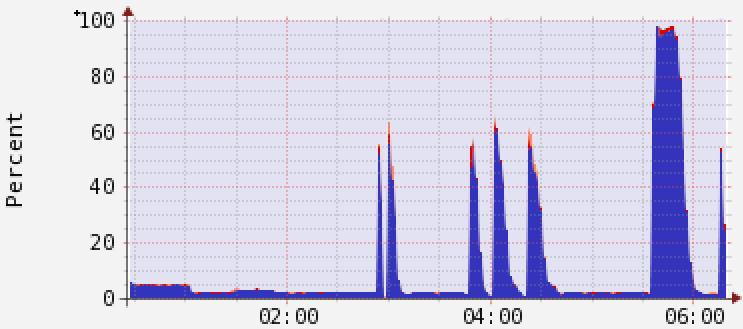


图 3.2 GATK-Queue运行资源利用率

对GATK-Queue运行方式的资源利用率的分析，让我们了解到非并行的步骤在整个流程中占了相当大的比重，因此，统计出这些非并行的步骤，并对其进行详细分析就很有必要。我们把picard中Mark Duplicates&Sort的运行合并为一部分，GATK中的Scatter-Gather和并行计算分别作为一部分，这样统计三个部分所占比例如图3.3所示，我们可以看到并行计算部分所占比较只有26%，Mark Duplicates&Sort和Scatter-Gather两个部分的占比达到了74%，而且这两部分是非并行的。

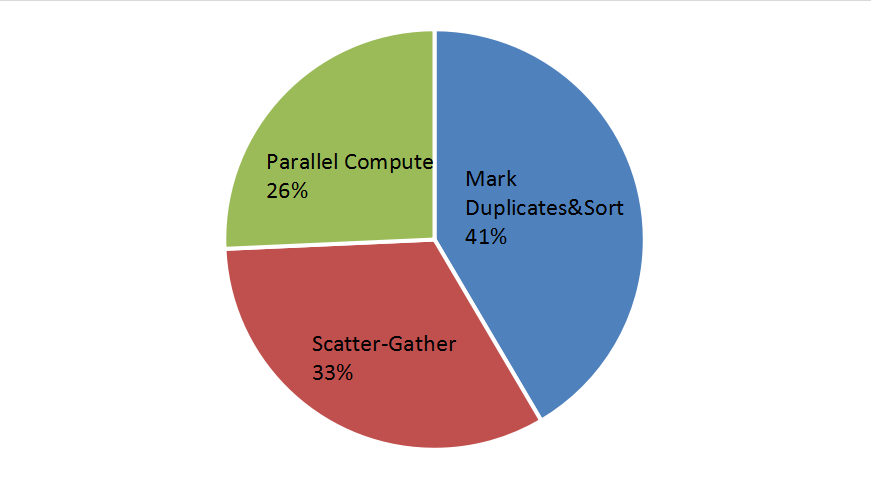


图 3.3 GATK-Queue各部分运行时间比例

另外，我们通过ganglia统计了GATK-Queue运行时各个步骤的CPU Time和IO Time的占比关系，如图3.4所示，整个流程的平均IO Time占比达到10%以上。因此，当我们通过并行的方式加速串行的步骤之后，IO的时间占比可能会成为新的瓶颈，所以在我们对整个流程并行加速之前就需要考虑到这个问题。

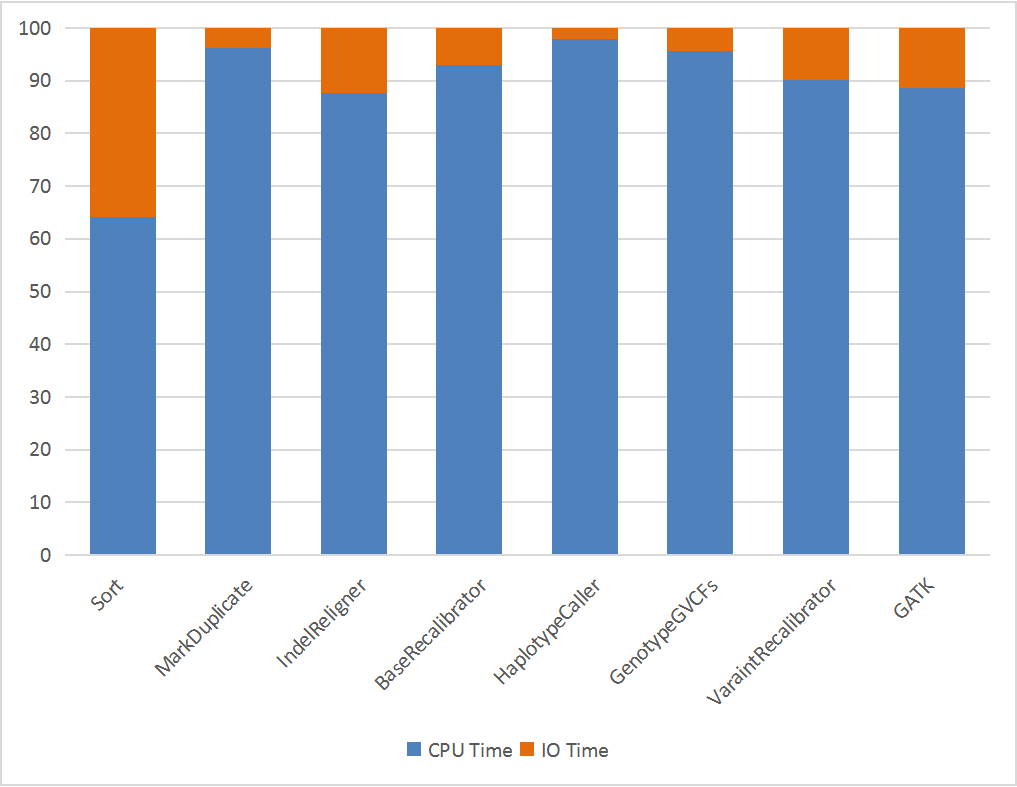


图 3.4 GATK-Queue各步骤CPU Time和IO Time占比

通过对GATK-Queue运行时间的粗略分析，我们大概可以看到影响整个流程性能的主要原因。接下来，我们会详细分析和探讨GATK-Queue中各个步骤的算法及处理过程，使我们对于各个步骤有更为直观的认识，为后续的瓶颈分析和平台选择提供坚实的依据。

## 3.2 流程分析及算法详解

如图3.1所示，GATK-Queue流程中主要包含六个步骤，其中Mark Duplicates&Sort是Picard中的工具，而后续5个步骤都是GATK中的工具。Picard是Broad Institute开发的一套操作高通量测序数据（HTS）的工具集，以htsjdk为基础，与Samtool的功能类似，广泛应用在基因数据处理过程中[27]。由于Haplotype Caller以及其后的Genotype GVCFs和Variant Recalibration是在Haplotype Caller基础上增加的操作，原理及数据关联都比较密切，因此本文中我们把这三个步骤合并在一起分析和改写。所以，我们的流程分析中主要对四个步骤进行分析和详解，分别是Mark Duplicates&Sort、Indel Realignment、Base Recalibration和Haplotype Caller。

### 3.2.1 Mark Duplicates&Sort

序列比对之后需要进行数据清理相关的工作，在进行Mark Duplicates之前，我们需要首先进行Sort，需要以排好序的序列进行分组并标记重复。Picard中的Sort是串行执行的，由于基因测序数据量非常大，普通机器的内存难以容纳，所以Picard中的Sort采用分块排序然后进行合并的方式来进行排序。如图3.5所示，Picard依次处理内存可容纳大小的数据进行排序，每一部分都是一个局部排序后的临时文件，接下来开始合并这些临时文件，通过采用类似于归并排序的方法不断将临时文件进行合并，直到最终合并成一个大的且全局有序的文件，则Sort部分的工作完成。

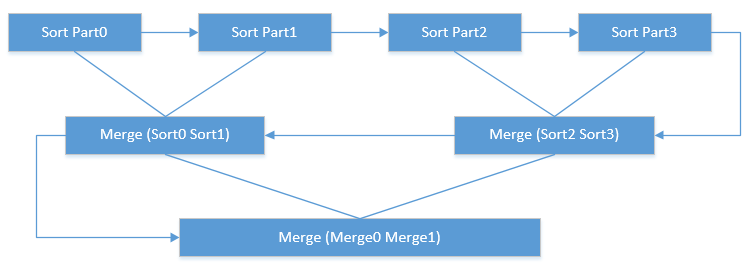


图 3.5 Picard中Sort的原理图

Mark Duplicates也是Picard中的工具，主要完成的工作是标记在测序中由于PCR过量扩增而产生的重复序列。Mark Duplicates是以排序好的序列为基础，对于双端测序方法得到的序列来说要分两种情况进行处理，一个是Pair-Ends的情况，另一个是Fragment的情况，如图3.6所示。

在Mark Duplicates的具体实现中，整个算法流程需要对所有序列和中间文件进行四次遍历。第一次遍历排序后的所有序列，构建生成标记重复所需要的信息，这些信息包括Pair-Ends的有序数组pairSort和所有Fragment组成的有序数组fragSort；第二次遍历上一步生成的pairSort，记录所有成对序列的冗余位置；第三次遍历第一步生成的fragSort，记录Fragment的冗余位置，其中Fragment中一旦有Pair-Ends序列的存在则整个组都被标记为冗余序列；第四次遍历，也是最后一次遍历，再次遍历所有序列，根据第二步和第三步中记录的冗余位置信息来对序列进行重复标记，最后写入目标文件。

Picard中的Sort和Mark Duplicates都是串行执行的，官方并未提供多线程或者多进程的运行方式，因此在GATK-Queue的运行方式中，这一步骤所占用的时间比较多，是我们优先进行并行优化的任务之一。

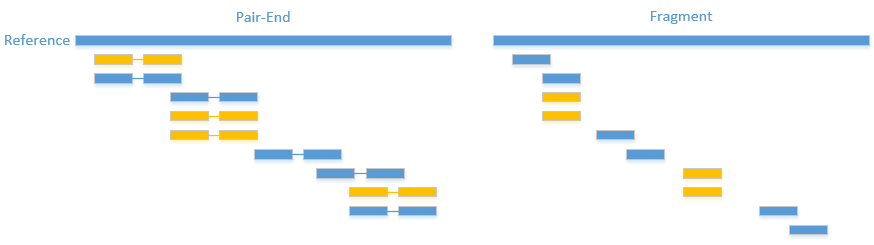


图 3.6 Picard中Mark Duplicates的原理图

### 3.2.2 Indel Realignment

Indel Realignment是GATK工具集中的第一个工具，这一步骤的主要任务是通过局部的indel重新校正来修正序列比对阶段的错误，另外通过多个序列的重复覆盖来使得序列indel的错误率更低。Indel Realignment分为两个步骤，如图3.7所示，首先使用RealignerTargetCreator来创建Target intervals，接下来根据Target intervals信息对序列进行矫正。

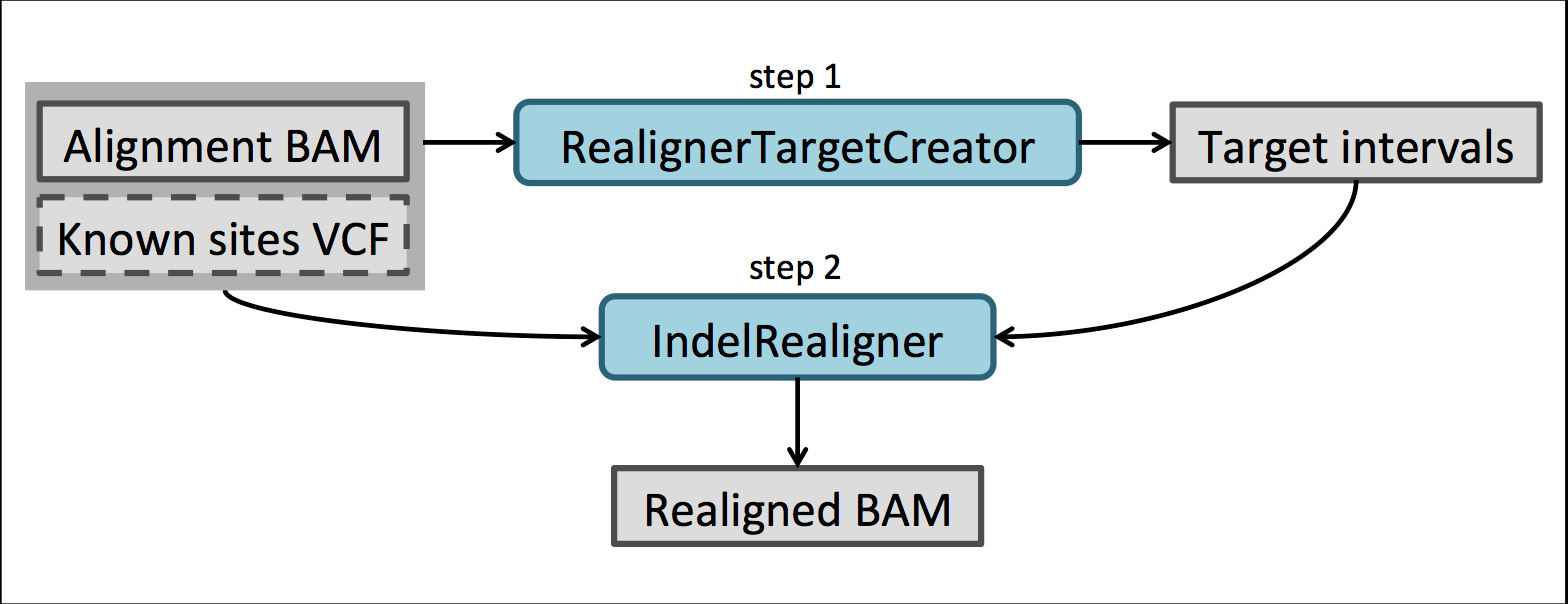


图 3.7 GATK中Indel Realignment流程图 [14]

第一步RealignerTargetCreator是找出可能存在indel的区间Target intervals，为下一步做局部的重新校正提供基础。寻找Target intervals的方法较为简单，对于已知的一些indel位点和snp位点直接对那些位置进行标记，而对于序列中可能存在的indel和snp则使用一种评估其出现可能性的熵的方式来表示，最终通过判断是否有indel或Event来决定是否使用当前区间，并通过在计算熵时来扩展区间的长度，最终找出所有的Target intervals。在GATK-Queue中，为了性能的考虑，直接把排序后的序列进行切分，相当于把整个参考序列切成多个区间，然后在每个区间中独立的找出所有Target intervals，这种做法虽然有一定可能性切断本来是一个区间的部分，但是由于这种可能性比较小，并且即使存在对结果的正确性的影响也不大，所以为了性能的考虑，忽略了这种问题。在GATK-Queue中的其他步骤中，基本上也是采用了同样的策略来提升性能。

得到Target intervals后再来进行IndelRealigner就相对比较简单了，IndelRealigner是局部的重新校对，局部的位置已经可以从Target intervals得到了，那么接下来直接进行重新比对就可以了。IndelRealigner中的比对和大多数的序列比对工具不同，它使用多个重复覆盖某一区域的序列一起来判定一个可能的indel的存在。

### 3.2.3 Base Recalibration

Base Recalibration与Indel Realignment类似，也是分两个步骤，首先搜集合并一个全局信息的表格，然后根据全局表格的信息对所有序列进行重新校正。全局信息表格的搜集是由这个步骤方法所决定的，Base Recalibration主要是对由于测序仪的机器周期和其它一些问题引起的测序质量分数的偏差进行纠正。如图3.8所示，测序得到序列的质量分数存在的偏差在经过GATK的质量分数校正后得到了很好的效果。

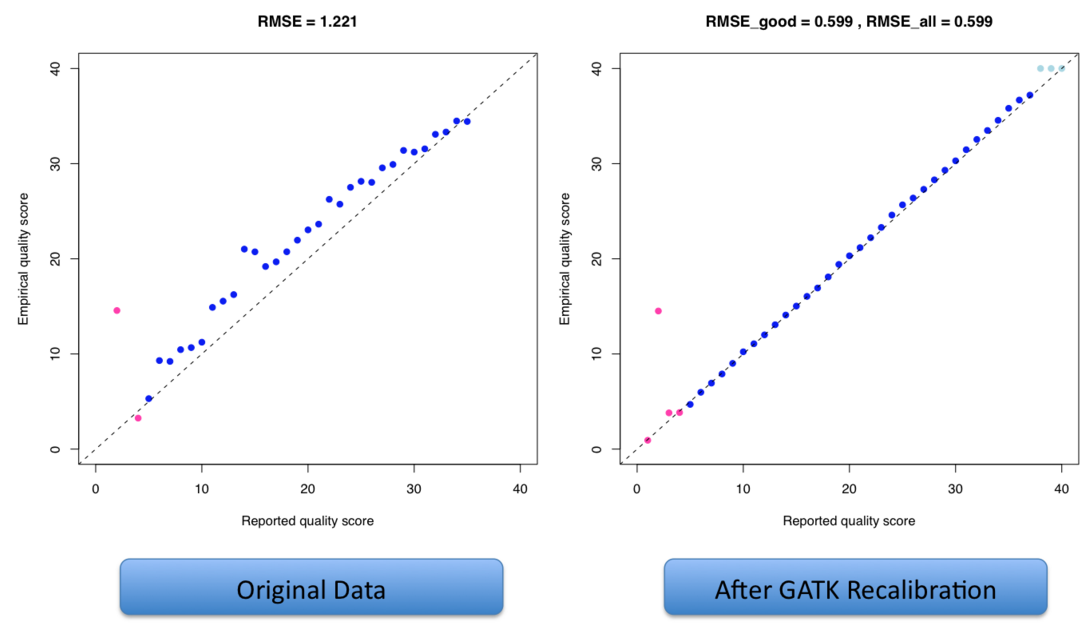


图 3.8 GATK质量分数重校正前后的结果对比 [14]

Base Recalibration算法的实现也不复杂，第一步中通过遍历所有序列，搜集并合并出三张表，分别是ReadGroup Table、Quality Score Table和Covariates Table。这三张表依次在更加细致的维度上统计了质量分数在某一个条件下按照经验应该得到的值，这些值就是接下来我们进行校正时需要参考的。第二步以第一步中统计的三张表为基础，通过对序列进行遍历并确认其归属的组别来对序列的质量分数进行校正，最终得到校正后的序列，为下一步也是最为重要的多样性发现提供更为准确有效的信息。

### 3.3.4 Haplotype Caller

多样性发现是GATK典型流程最为重要的步骤，也是其目的所在。GATK工具集中发现多样性的工具有多个，GATK典型流程中使用Haplotype Caller作为多样性发现的工具，Haplotype Caller也是GATK官方推荐使用的多样性发现的工具[28]。Haplotype Caller中发现多样性位点主要分为四个步骤，如图3.9所示，接下来分别来介绍这四个步骤。

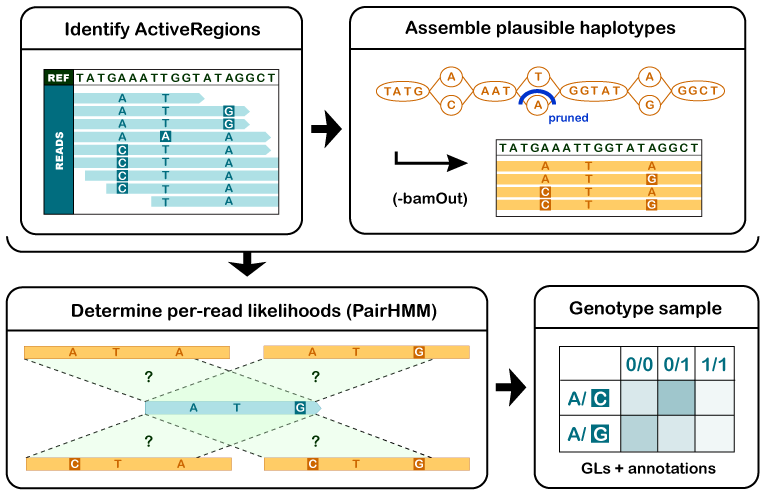


图 3.9 GATK中Haplotype Caller原理图 [14]

第一步是确定ActiveRegions（Identify ActiveRegions），ActiveRegions是可能存在多样性位点的区域。确定ActiveRegions可分为三小步，首先在每一个位置计算一个得分（activity score）来表示这个位置是多样性位点的可能性，计算的方式是使用reference-confidence模型进行计算；然后对每一个位置的得分需要做一个高斯平滑，使用高斯核函数向当前位置左右各50bp做一个横向的平滑，并把所有的平滑进行合并；最后，对所有的位置进行一个阈值过滤，凡是得分超过某一个阈值的位置，都是活动位置（activity position），把活动位置连起来就得到了ActiveRegions。得到ActiveRegions后，接下来对每个ActiveRegions都要执行下面三个步骤。

第二步是装配可能的haplotypes（Assemble plausible haplotypes），这一步中首先要把ActiveRegion构建成一个参考序列的德布林图（de Bruijn graph）来进行装配，然后对德布林图中的每一条边算一个转换概率，合并一条路径上的所有转换概率就得到了一个可能的haplotype的得分（likelihood score），最后选取N个得分最高的路径，这N个路径就是我们需要装配的可能的haplotypes。

第三步是确定每个序列的likelihoods（Determine per-read likelihoods），上一步中我们得到了得分最高的N个haplotypes，在这一步中，我们需要把这N个haplotypes的得分（haplotype likelihood score）作为可见状态，去计算每个序列的得分（read likelihood score），这里就需要使用PairHMM的方法，通过PairHMM方法的转换最终得到每条序列的得分，最后，对每个位点我们对所有可能的等位基因进行评估，计算出每个位点的多样性的得分[29]。

最后一步是为每个样例确定基因型（Genotype samples），这一步主要是针对多样例数据的合并，通过使用贝叶斯等式的变换，得出计算多样例时基因型的计算方式。通过这种方式计算出每个基因型的得分，并且选取其中最大的作为该位点的基因型。

## 3.3 瓶颈分析

通过流程的相关测试，我们可以看到GATK-Queue运行方式的性能并不高，而影响整体性能的关键就是Picard中的Mark Duplicates&Sort和GATK中并行步骤的Scatter-Gather部分。经过对流程的分析和各个步骤算法的详细了解，我们总结了造成GATK-Queue运行方式性能不高的瓶颈，如图3.10所示，我们总结了三个瓶颈，分别是大文件不断的切分合并、Picard中的工具是串行执行的、GATK中Scatter-Gather是串行执行的。

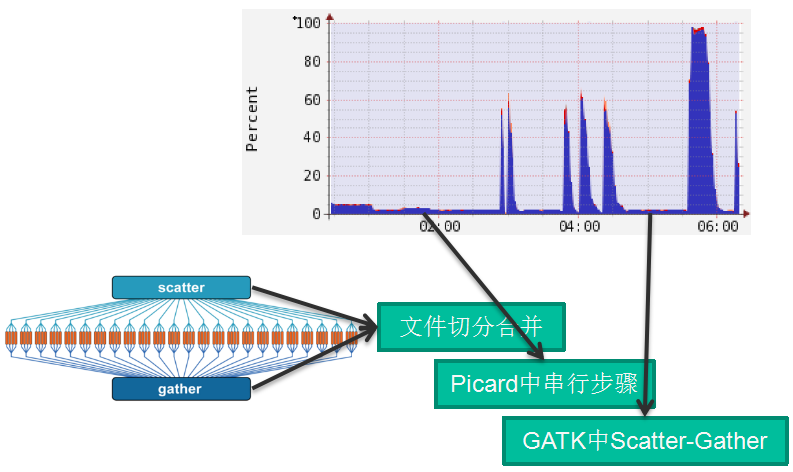


图 3.10 GATK-Queue瓶颈分析

### 3.3.1 大文件的切分合并

GATK-Queue的运行方式中，各个步骤之间通过一个大的文件作为连接。在可并行执行的GATK工具中，每次需要把大文件切分成多个小文件分发给各个节点，当任务结束时需要把所有结果小文件合并成一个大的文件。不断的切分合并文件在一定程度上浪费了资源和时间，我们可以通过减少这种切分合并来提高流程的并行性。在大文件切分成多个小文件后，后续步骤不再不断的合并切分了，而是在每个小文件中执行多个步骤，当需要全局信息或者有数据依赖时，只需要通过网络传输全局信息或者数据依赖则可以。

### 3.3.2 Picard中串行步骤

数据预处理中起始的两个步骤Mark Duplicates&Sort都是Picard中的工具，3.2.1节中我们详细描述了Mark Duplicates&Sort的算法以及细节，我们知道Mark Duplicates&Sort都是串行执行的，在序列数据相对较大时，这两个步骤的运行时间会非常长。通过图3.3我们可以看到Mark Duplicates&Sort运行时间在整个流程中的时间占比达到了41%，那么对这两个步骤的并行加速就迫在眉睫了。

### 3.3.3 GATK中Scatter-Gather

在GATK-Queue运行方式中，需要不断的切分合并文件，这种切分合并文件就叫做Scatter-Gather。当然，Scatter-Gather中除了切分合并文件外还有一些为切分合并文件提供正确性保证的相关计算，例如合并文件时需要结果是有序的。Scatter-Gather的运行时间大概占整个流程运行时间的33%，而且这部分的运行也是串行执行的，这就造成了GATK-Queue运行的性能不高的瓶颈之一。

## 3.4 平台选择

GATK-Queue运行方式中由于并行不够彻底，造成了其性能受到一定影响，运行时间和可扩展性都不是很好，所以我们提出了对其进行并行加速的想法。由于是对GATK典型流程的加速，所以我们首先需要考虑的就是性能，性能上的提升需要我们选用高效稳定的平台；另外，由于GATK典型流程中涉及到多个步骤，每个步骤的复杂性相对较高，想要在一定的时间完成整个流程的并行加速，所以我们还要考虑到效率的因素。

上一节中我们知道文件的不断切分合并是GATK-Queue运行方式的瓶颈之一，大文件的不断切分合并对资源和时间都是大的浪费，而在大数据处理平台Spark中提供了RDD的机制，基于内存的分布式冗余数据集有效的解决了大文件不断切分合并的问题，通过Spark本身实现的接口和DAG方式来完成不同步骤中数据之间的依赖关系，另外由于RDD是缓存在内存里，其读写速度和效率相比大文件都大大提高。加州大学伯克利分校开发的ADAM为在Spark上操作基因测序格式的数据提供了大大的遍历，不仅可以帮我们减少了一定的工作量，而且由于其数据格式是基于Spark的使用而定义的，其性能以及稳定性相对于htsjdk中的格式好很多。另外，Spark作为一个较为新的大数据处理平台，提供了一百多个API接口，相对于hadoop中的map reduce丰富了很多，为我们开发新的Spark程序或者改写原有程序为Spark程序提供许多便利，可以更高效率的完成开发或者改写。

基于性能和效率这两个考虑，以及RDD解决文件瓶颈、已有ADAM作为基础和Spark丰富高效的编程接口这三个原因，我们提出使用Spark的处理方式来改写GATK-Queue。通过把GATK-Queue中的计算逻辑迁移到Spark中，有数据依赖及全局信息的部分改写成Spark处理的方式来实现对GATK-Queue的并行优化，提高GATK典型流程的性能，使得我们能在相对较短时间内得到基因测序的结果。我们把这种把GATK-Queue改写成Spark处理方式的流程叫做GATK-Spark运行方式。

## 3.5 本章小结

本章我们首先对流程做了一些简单的测试工作，然后详细分析了整个流程以及流程中各个步骤的算法。通过流程测试的结果以及对流程的一些细致分析我们总结了造成GATK-Queue运行方式性能不高的三个瓶颈：大文件的切分合并、Picard中串行步骤和GATK中Scatter-Gather。最后基于性能和效率的考虑以及Spark适合我们改写的三个原因提出了基于Spark平台的GATK-Queue改写，并命名为GATK-Spark。

# GATK-Spark的设计与实现

上一章节我们根据对GATK典型流程的分析，提出了基于Spark平台的GATK流程的改写，这一章节我们就基于Spark平台的GATK流程：GATK-Spark的详细设计与具体实现。我们知道，基于Spark平台的程序需要保证平台的适用性，所以我们关于GATK-Spark的设计和实现都要考虑到这一点，另外由于性能方面的考虑，需要我们对数据结构以及相关操作的序列化及网络都有一定的考虑。接下来，我们就通过平台架构的整体设计和每一部分的具体设计来探讨GATK-Spark的设计和实现。

## 4.1 平台架构与整体设计

Spark提供了一个快速高效的分布式计算框架，我们可以自由选择存储和运行模式，如图4.1所示[30]。在存储系统的选择上，既可以使用Hadoop中的HDFS或Hbase，也可以使用一些通用的网络文件系统NFS或lustre等。另外Spark提供了多种运行方式，既可以和YARN或Mesos这些分发调度模块共同使用，也可以使用自身独立运行模式。本文中，由于我们设计实现的是一个全新的基于Spark的程序，不需要考虑由于系统中的影响，所以对于运行方式基本没有限制，我们选择了较为方便的独立运行模式。而存储系统的选择上，我们直接使用了NFS的方式，由于我们在后续的GATK-Spark的设计中对于文件的读写只有开始的读入数据和最后的写出数据两个部分，因此对于存储系统的性能要求也不高。

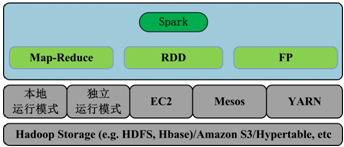


图 4.1 Spark的架构图 [30]

完全把GATK典型流程使用Spark处理方式来重新设计和实现对于整个流程来说肯定是最好的改写方式，这样可以针对算法并行化的具体细节减少一些不必要的操作，但是由于其工程量和难度都相对太大，所以可操作性比较低。基于工程效率和性能方面的考虑，我们提出了具体问题具体分析的思路，对于逻辑较为简单的步骤，我们可以通过基于Spark处理方式的完全重写，而对于逻辑复杂、代码量大的步骤，我们选择保留原有计算逻辑，只改写数据划分部分的逻辑，使其可以运行在Spark平台上。

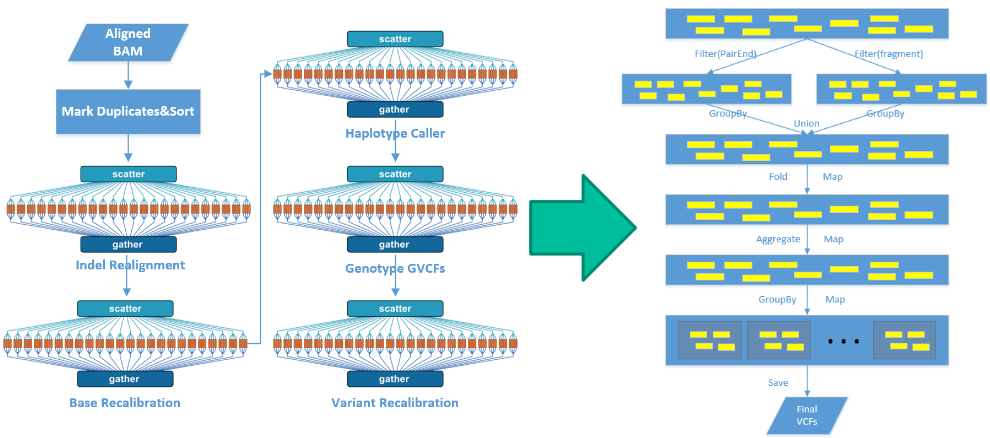


图 4.2 GATK-Queue到GATK-Spark的转化

我们知道GATK-Queue运行方式是一边读取文件一边进行计算的方式，另外流程中存在很多串行计算的部分。对于Picard中的Mark Duplicates&Sort逻辑相对较为简单，我们可以通过使用Spark处理方式完全重写的方式来实现并行优化，而对于Haplotype Caller及其后续步骤来说逻辑复杂且代码量大，我们可以通过保留其计算逻辑而改写其数据划分部分。对于中间两个步骤Indel Alignment和Base Recalibration在伯克利开发的ADAM中已经有了相关的实现，所以我们借助已有的逻辑，通过调整其中部分过程和增加部分内容，实现和GATK-Queue中一样得到相同的处理结果。另外，ADAM中对二代测序数据格式的支持已经相对稳定且高效，所以我们接下来的改写都会以其数据格式作为基础。

上一章节中，我们分析出阻碍GATK-Queue运行方式性能提高的三个瓶颈：大文件的切分合并、Picard中串行步骤和GATK中Scatter-Gather，那么我们对于GATK-Spark的设计也要针对这三个瓶颈提出优化的思路。针对大文件的切分合并，我们提出了数据交换的优化，通过Spark平台自有的RDD，大文件的不断切分合并得到有效的解决；对于Picard中的串行步骤，我们需要进行预处理算法的并行优化；GATK中Scatter-Gather部分存在的串行执行的问题，我们也提出了Scatter-Gather的并行。接下来，我们就针对这三个部分来详细介绍我们GATK-Spark具体的设计和实现。

## 4.2 数据交换的优化

GATK-Queue中每个步骤之间通过一个大的文件来进行数据的传递，对于并行步骤之间需要首先把大文件切分成多个小的文件，然后最后再把所有小文件合并成一个大文件，需要有序的时候还需要一些计算，不断的切分合并文件造成了很大的资源和时间上的浪费，我们需要来解决这种不断的切分合并文件。我们知道这种不断的切分合并其实是没有必要的，我们可以在一开始就把文件切分成多个小的文件，也就是把数据划分成多个小的部分，然后在接下来的运行中只需要在小数据集上计算就可以，当产生数据依赖或者需要一些全局信息时，我们通过网络传输的方式进行数据的交换。当然对于需要全局信息的部分，通过一个master节点搜集所有需要的部分组成一个大的全局数据，通过广播的方式把数据分发给需要的节点则可以实现。通过这种方式，大文件的不断切分合并则可以避免，可以有效的减少文件的读入和写出，减少资源和时间的浪费，图4.3中我们用示意图展示了解决数据交换依赖的大致原理和搜集分发全局信息数据的方法。

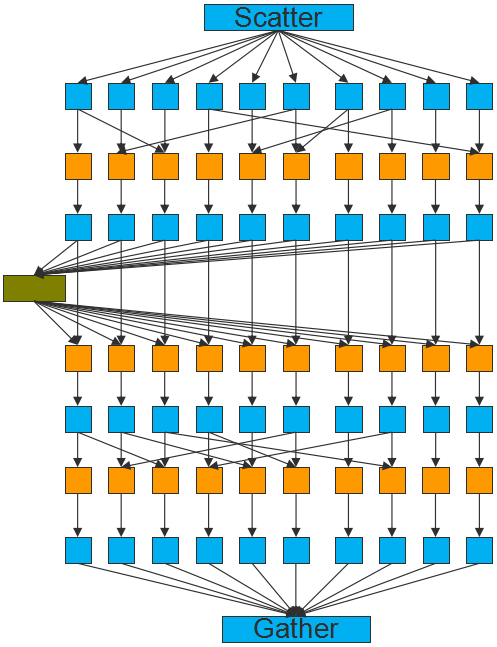


图 4.3 解决数据交换依赖及分发全局信息示意图

虽然对于大文件切分合并瓶颈问题的解决方法看起来比较复杂，但其实通过我们使用Spark平台的计算方式，已经自动帮我们解决了这个问题。Spark中采用了RDD（冗余分布式数据集）的机制，它是把数据读入成驻留在内存中的数据集，当我们调用其提供的api接口时，Spark中的DAG机制计算数据之间的关系和依赖，找出合适有效的执行路径，数据之间的依赖已经在Spark中被自动的解决，我们只需要关注算法的正确性则可以，而不再关心其详细实现的细节，图4.4中简要表述了在Spark中RDD解决数据依赖的原理[31]。另外，由于RDD是在内存中的，所以不再需要不断的读写文件，这对于性能的提升很是有效。

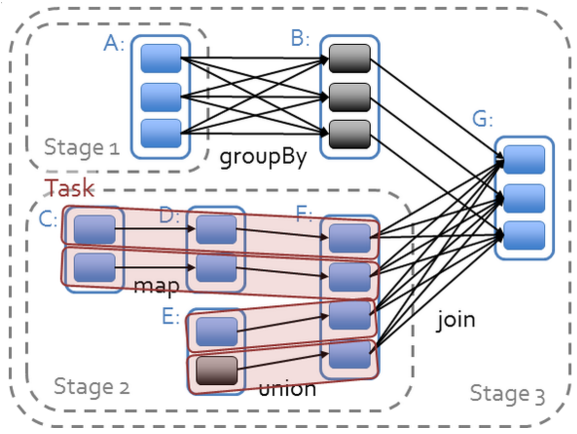


图 4.4 Spark中RDD解决数据依赖原理图 [30]

## 4.3 预处理算法的并行优化

Picard中Mark Duplicates&Sort都是串行执行的，上一章节中我们已经详细分析了其具体的算法实现，现在我们就根据串行算法的原理在Spark中重新来实现Mark Duplicates的Spark版本。对于双端法测序得到的序列，我们在标记重复时需要对Pair-End的序列和Fragment的序列区别对待，所以对于Mark Duplicates的改写首先需要对序列进行区分。

val fragments = rdd.filter(r => (r.getReadMapped && !r.getSecondaryAlignment) )

val pair-ends = fragments.filter(r => r.getMateMapped)

图 4.5 GATK-Spark中区分Pair-End和Fragment序列

对于Pair-End的序列，我们需要把一对序列作为一个整体来考虑，这个序列对在参考序列上会有两个对应的位置（在序列比对阶段在参考序列上的位置），我们把这两个位置作为序列对的键值，然后以此键值来进行分组，分到同一个组中的序列对就是我们需要进行重复标记的序列组，接下来在同一个组中对每个序列对计算一个表示其代表性能力的得分，得分最大的序列对作为整个组的代表，而组内其他所有序列对都标记为重复序列。

对于Fragment的序列和Pair-End的处理方式类似，不同的是Fragment中是每一个序列单独处理的，按照其比对到参考序列上的位置进行分组，分到同一个组中的序列就是我们需要标记重复的序列。另外一个不同之处是，如果一个序列组中存在Pair-End的序列，那么除了Pair-End的序列其他所有序列都要标记为重复，如果序列组中不存在Pair-End的序列，那么和Pair-End序列的处理方式一样，计算一个表示其代表性能力的得分，得分最大的序列作为代表，其他所有序列标记为重复。

if(i != maxScorePos) {

val nscore = pairs(i).map(score).sum

pairs(i).foreach(r => {

r.setDuplicateRead(true)

reads ::= r

})

}

图 4.6 GATK-Spark中Pair-End序列标记重复

对Pair-End和Fragment的处理方式极为相似，在GATK-Spark中我们可以把这两种处理方式放在一起来实现。Pair-End的序列对需要其对应的位置对作为键值，我们需要定义其相应的数据结构，而对于Fragment的序列只需要在参考序列上的位置作为键值就可以。基于其对应的键值我们使用Spark的groupBy接口对序列进行分组，同一个位置上的序列分配到同一个组内。在Pair-End的组内使用最大得分作为代表，其他序列标记重复的方式，而在Fragment的组内需要增加是否有Pair-End序列的考虑。最终得到标记好的序列的集合，使用Spark的union接口合并两个不同集合中的序列，得到标记重复后的序列集。我们在图4.7中描述了Spark版Mark Duplicates的整个过程，从开始序列的RDD集合，通过Filter得到Pair-End的序列和Fragment的序列，然后分别使用Group by进行分组并在组内进行得分计算和标记重复，最后通过Union合并Pair-End的序列和Fragment的序列，得到标记重复后的序列RDD，作为下一步计算的基础。

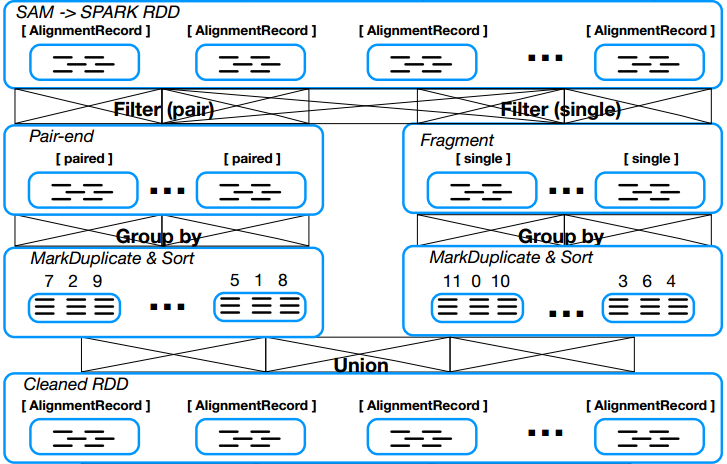


图 4.7 Spark版Mark Duplicates原理图

在GATK-Queue的运行方式中，Picard中的Mark Duplicates是在Sort的基础上进行的，需要首先对序列进行排序才能进行重复标记，而在GATK-Spark的设计中，由于我们使用了Group by的方式，不再需要序列是有序的，只需要其对应在参考序列上的位置作为键值则可以。在后续的步骤中，我们仍然需要序列是有序的，所以Spark版本的Sort还是要实现的，庆幸的是Spark的排序非常简单，Spark中提供了sortBy的接口，我们需要做的是把排序需要的比较函数就可以。由于我们需要的是基于序列比对到参考序列上的位置来进行排序，所以只需要实现对于位置的比较函数则可以，而Spark版的Sort只需要一个sortBy的函数即可。

//基于Spark中sortBy接口的排序方式

alignedRecords = alignedRecords.sortReadsByReferencePosition()

//不同区间独立sort的排序方式

Collections.sort(filteredReads, new SAMRecordCoordinateComparator());

图 4.8 两种Spark版的Sort

虽然Spark版本的Sort非常简单，但是在Spark平台的RDD中数据是根据其大小和节点个数分区的，sortBy函数对其排序后序列的顺序仅在当前步骤是保证顺序的，一旦我们对RDD使用了其他接口的操作，RDD中的序列顺序就不能保证了。在后续的步骤中，我们通过计算逻辑确定了需要进行操作的一些区间，分配到这些区间的序列由于其多个分区的不确定性是不能保证顺序的。这种情况下，这些区间通常不会太大（<300），其对应的序列也相对比较少，我们只需要把他们进行快速排序即可，由于这样的区间非常多，且其排序时间也不长，最重要的是这些区间还在不同的分区不同的节点上，所以并行度也是非常高的。

## 4.4 Scatter-Gather的并行

GATK-Queue中的并行步骤使用了网格计算的方式，通过Scatter进行分拆数据，Gather进行合并数据，Scatter-Gather的过程都是串行进行的。在上一章节中我们详细分析了这些并行步骤的算法和原理，本节我们就针对这些串行步骤中的Scatter-Gather，提出对其并行优化的方法并具体实现。需要进行Scatter-Gather并行加速的步骤主要有三个：Indel Realignment、Base Recalibration和Haplotype Caller。其中，Indel Realignment和Base Recalibration在ADAM中已经完全重新改写，我们通过阅读ADAM中相关的源码，分析了其进行Spark改写的原理，并且在其基础上做了相应的一些修改，达到和GATK中的方法得到相同的运行结果。

### 4.4.1 Spark版Indel Realignment

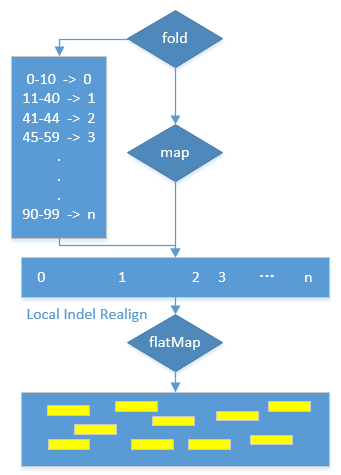


图 4.9 Spark版Indel Realignment原理图

Indel Realignment是通过在一个局部区域对序列进行重新比对来实现indel的修正，图4.9中给出了ADAM中Indel Realignment的原理图，主要分为两个部分：找出所有需要indel修正的所有局部区域、在局部区域内实现indel修正。找出所有局部区域是一个相对较为复杂的过程，图4.10中给出了其流程图，其中最为核心的步骤要数图4.11给出的核心代码部分。

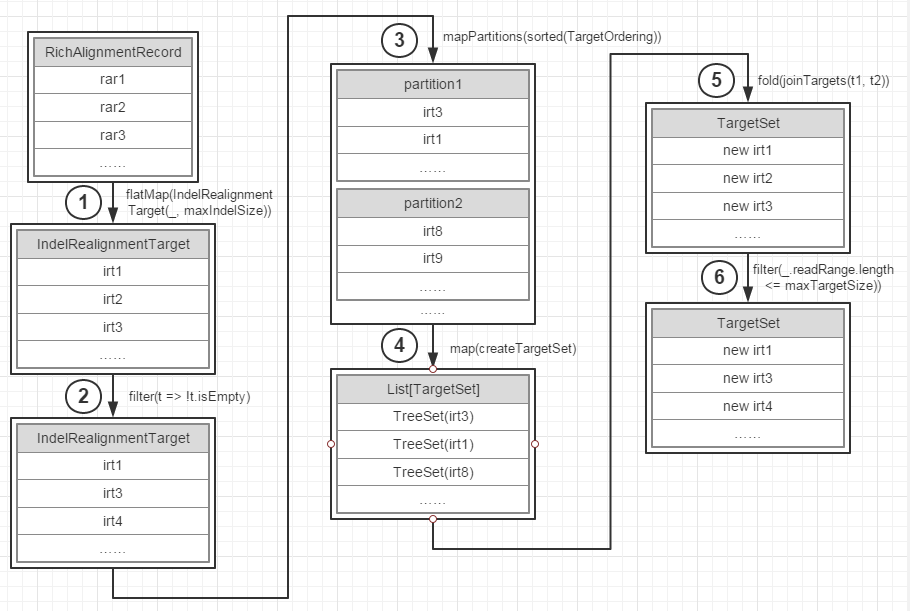


图 4.10 Spark版Indel Realignment中找出局部区域流程图

找出所有局部区域的过程大概可以分为两步，首先把所有indel出现的区间全部找出来，也就是图4.10中的1、2、3、4步所做的，然后把这些所有的区间进行合并，在长度超过一定阈值时进行截断，对应图4.10中的5、6步。找到所有局部区域后，在局部区域实现indel修正相对来说就比较简单了，与GATK-Queue中处理的方式一样，在局部区间进行indel修正时，需要多条序列作为一个整体来进行indel的修正，最终得到修正后的结果。

val targetSet : TargetSet =

TargetSet(targets.mapPartitions(iter=> iter.toArray.sorted(TargetOrdering).toIterator )

.map(createTargetSet)

.fold(TargetSet())((t1: TargetSet, t2: TargetSet) => joinTargets(t1, t2))

.set.filter(\_.readRange.length <= maxTargetSize))

图 4.11 Spark版Indel Realignment核心代码

### 4.4.2 Spark版Base Recalibration

另一个在ADAM中完全用Spark重写的步骤是Base Recalibration，这一步骤主要是为了对序列的质量分数进行重校正。在上一章节中我们已经介绍了质量分数校正的由来以及其校正的方法，主要就是统计三个包含全局信息的表，这三张表分别是globalTable、qualityTable和extrasTable，通过这三张表中统计的信息来对序列进行质量得分的重新校正。

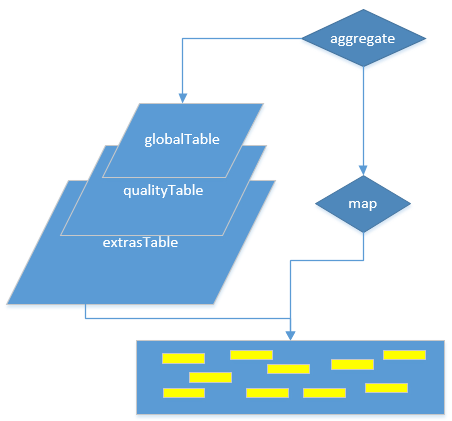


图 4.12 Spark版Base Recalibration原理图

我们在图4.12中展示了在Spark中Base Recalibration方法的原理图，其余Indel Realignment的方法类似，也是主要分为两个步骤：统计全局信息的表、对序列进行重校正。全局信息的表包含三个，在图4.13中给出了计算qualityTable的部分代码，globalTable和extrasTable的计算类似，由于篇幅原因不再一一列出。

def computeQualityTable(globalEntry: (String, Map[CovariateKey, Observation]),

space: CovariateSpace): Map[QualityScore, (Aggregate, ExtrasTables)] = {

globalEntry.\_2.groupBy(\_.\_1.quality).map(qualityEntry => {

(qualityEntry.\_1, (aggregateObservations(qualityEntry.\_2),

new ExtrasTables(computeExtrasTables(qualityEntry.\_2, space))))

}).map(identity)

}

图 4.13 Spark版Base Recalibration计算qualityTable

Base Recalibration中最为重要的就是统计全局信息的表，图4.13中只列出了计算qualityTable的计算代码，而接下来在图4.14中，我们把计算得到Recalibration Table的流程图画了出来。生成Recalibration Table大概有五个步骤，分别对应图4.14中的1到5的序号，第一步是对序列进行过滤，过滤掉不合适的序列，第二步是把序列中的信息转化为需要统计的全局信息表中的格式的数据，第三步把统计信息根据其相关性进行部分合并，第四步把不同类型的表中的数据组合到一起，最后第五步进行一个封装。

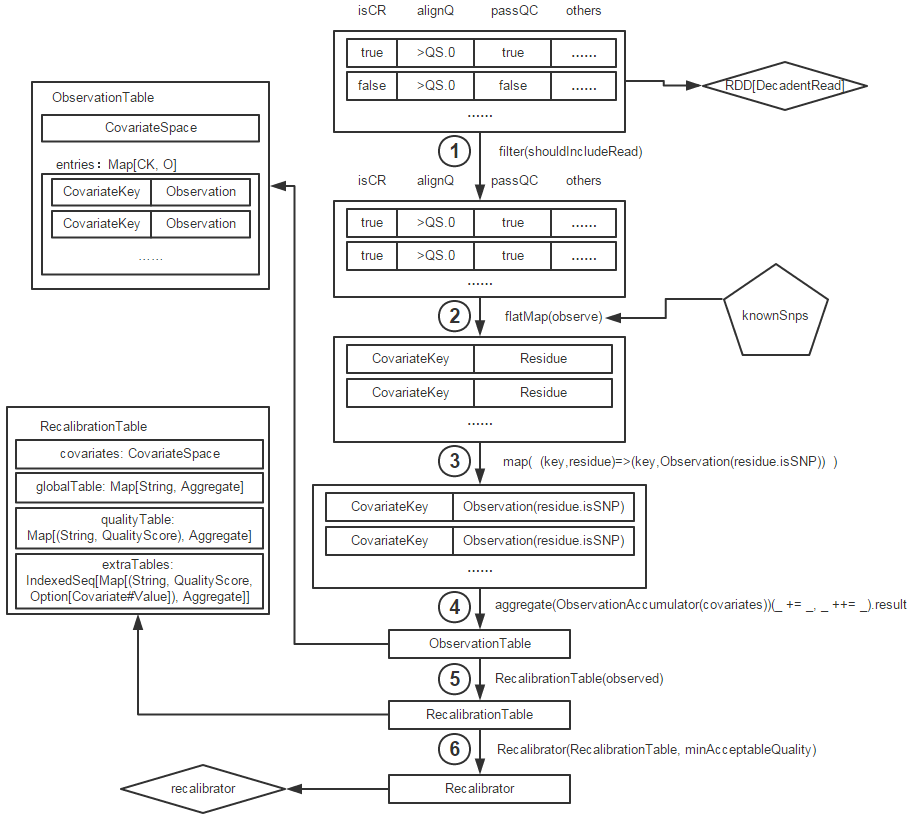


图 4.14 Spark版Base Recalibration中统计全局信息表流程图

得到全局信息的表后，我们需要把序列的质量分数重新计算，通过序列中不同位置的碱基所拥有的信息把其归属到不同的组别中，然后根据全局信息表计算一个新的得分，对于原有得分高于某个阈值的碱基得分进行更新，而低于阈值的碱基序列得分不再更新，最终得到了校正后的质量得分序列，作为下一步多样性发现的基础。

### 4.4.3 Spark版Haplotype Caller

Haplotype Caller是GATK典型流程中的核心步骤，也是GATK-Queue流程中最为复杂的一个步骤。在上一章节中，我们讨论过GATK-Queue中Haplotype Caller的原理和算法，详细分析了Haplotype Caller的四个组成部分。由于Haplotype Caller计算逻辑的复杂性，想要把Haplotype Caller像Mark Duplicates那样完全重写成Spark的处理方式在短时间内是无法完成的，因此我们提出了保留GATK中发现多样性位点的计算逻辑，通过把数据划分到不同区间来实现Scatter-Gather部分的并行性，最终改写成Spark版本的Haplotype Caller。想要在保留原有计算逻辑的基础上把Haplotype Caller改写成Spark版本的处理方式需要我们解决两个方面的困难，一个是方法层面，另外一个是实现方面。

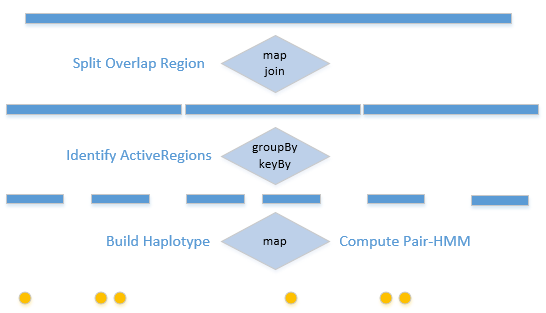


图 4.15 Spark版Haplotype Caller原理图

首先，我们从方法层面来提出解决的方案。Haplotype Caller方法中首先需要找到ActiveRegions，接下来所有并行操作都可以基于ActiveRegions来实现，那么我们想要把Haplotype Caller改写成Spark的处理方式，最重要的任务就是找出ActiveRegions部分的改写。对于找出ActiveRegions部分的改写可以有两种思路，第一种思路类似于Indel Realignment，把所有ActiveRegions相关区域找出来，合并所有这些区域并广播给所有计算节点，第二种思路类似于GATK-Queue中的处理方式，按照参考序列切分区间，在各自区间独立找出所有ActiveRegion，当然，不同于GATK-Queue我们可以通过overlap的方式对切分区间进行拟补。对于第一种思路，通过一些简单的测试我们发现了其不可行，在Indel Realignment中Indel出现的位置和区间是独立的，但是在Haplotype Caller中这些ActiveRegions却是相互依赖的，新的ActiveRegion必须是在上一个ActiveRegion之后开始找。

因此，我们采用了第二种思路，如图4.15所示，通过把参考序列切分成多个区间，然后在各个区间内部独立找出其ActiveRegions并进行其他计算。由于我们是把区间进行强制切分的，所以上一个ActiveRegion对当前的依赖没有考虑，而且有可能我们切分区间的位置上存在ActiveRegion被切断，为了解决这个问题，我们提出了Overlap的机制，如图4.16所示。通过把区间进行一个扩展，这样即使ActiveRegion位于被切分的位置上，我们也能在扩展的区间中把这个ActiveRegion找到。另外，由于ActiveRegion有不能超过300bp的限制，所以在切分处的下一个区间，在300bp位置后的影响几乎不存在了。这样我们就在方法层面找出了可行的解决方案。

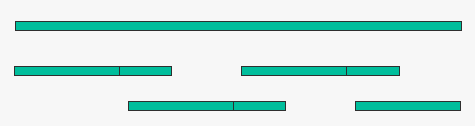


图 4.16 Spark版Haplotype Caller中Overlap机制

另外需要考虑的是实现方面的问题。Haplotype Caller中找出ActiveRegions和计算多样性位点部分的逻辑相对都比较复杂，而且在GATK-Queue代码中是一边读文件一边进行计算操作的，我们需要把计算部分的代码抽取出来，通过Spark中分发的数据来调用这些计算逻辑。抽取计算逻辑代码的工作是一个相对耗时耗力并且技术难度相对较低的一个过程，我们主要抽取了Haplotype Caller中计算ActiveRegions和多样性位点的计算逻辑。

val result =

fragments.keyBy(f => (f.getContig.getContigName, f.getFragmentNumber.toLong))

.join(myRecords.groupBy(r =>

(r.getContig.getContigName, (r.getEnd-1)/frgBcast.value)))

.leftOuterJoin(filtered.groupBy(f =>

(f.getContig.getContigName, (f.getStart-301)/frgBcast.value)))

.map(p =>

HaplotypeCaller.findVariants(p, List.empty, hdrBcast.value, frgBcast.value))

.flatMap(p => p.seq)

图 4.17 Spark版Haplotype Caller核心代码

最终，我们把Haplotype Caller中的逻辑进行了封装，在Spark中进行区间划分和数据分割后来进行调用。图4.17是Spark版本的Haplotype Caller数据分割方面的核心代码，通过把序列划分到不同的区间并在这些区间中调用我们封装好的计算逻辑来实现Haplotype Caller在Spark上的运行。

## 4.5 本章小结

本章主要是对GATK-Spark的设计与实现。我们首先对整体的平台架构和整个流程的一个概要设计进行描述，提出了针对不同步骤使用不同方法的策略。接下来对分别对GATK-Queue中存在的问题提出了具体的优化策略。针对大文件的切分合并提出了数据交换的优化，通过Spark中的RDD有效解决了这方面问题。对于Picard中存在的串行步骤，提出了对于预处理算法的并行优化，完全重写了Mark Duplicates和Sort部分，使其在Spark上能很好的运行。GATK-Queue中存在的Scatter-Gather部分效率不高的问题，我们通过把这些步骤改写成Spark的处理方式来对其进行并行优化，分别对其三个步骤进行了Spark版本的改写。最终我们完成了把GATK-Queue改写并优化成GATK-Spark的任务，实现了对GATK典型流程的整体优化。

# 性能评测及实验分析

通过对GATK-Queue运行方式的改写和优化，我们设计并实现了基于Spark版本的GATK数据分析流程：GATK-Spark。本章我们就对GATK-Spark进行详细的性能测试和分析，确认我们并行优化的效果，同时找出我们的优化存在的问题和不足。接下来，就开始我们的性能测试和分析。

## 5.1 实验配置

本文的实验平台采用曙光的一个小型集群，这个集群由33个节点组成，其中每台机器的详细配置可参考表5.1中描述。在这个集群中，我们选取了一个节点作为master节点，而其它节点作为slave节点，通过控制节点数量及计算资源来测试我们的GATK-Spark的性能。

表 5.1 单台机器的配置参数

|  |  |  |
| --- | --- | --- |
| Component | Details | |
| Processor | Intel Xeon E5-2680, Haswell micro-architecture | |
|  | Cores | 2\*12 @2.5GHz |
| Superscalar Width | 4 inst issue |
| Out-of-order Window | 192 |
| Load bandwidth  Store bandwidth | 64 Bytes/cycle  32 Bytes/cycle |
| L2 Unified Cache | 256K, 8-way |
| Fastest load-to-use  Bandwidth to L1 | 11 cycles  64 Bytes/cycle |
| LLC Cache | 30MB |
| L1 iTLB | 4K: 128,4-way  2M/4M: 8/thread |
| L1 DTLB | 4K: 64,4-way  2M/4M: 32,4-way  1G: 4,4-way |
| L2 Unified TLB | 4K+2M shared: 1024, 8-way |
| Memory | 128GB, DDR4 2133MHz | |
| Network | 1000Mb/s Ethernet | |

数据方面，我们使用了一套基于由1000 Genome Project Phase I得到的b37参考序列的数据，具体的实验数据及其参数在表5.2中有详细描述。

表 5.2 测试数据的相关参数

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Type | Dataset | Orgnism | Sequencer | Length(bp) | Input Size |
| DNA | SRR54516 | Homo sapiens | HiSeq 2000 | 101 bp | 152,905,146 reads |
| Reference | 1000 Genome  b37 | Homo sapiens | / | / | 3 GB |
| dbsnp | dbsnp\_138 | Homo sapiens | / | / | 10GB |
| indel | 1000G phase1 | Homo sapiens | / | / | 227MB |
| snp | 1000G phase1 | Homo sapiens | / | / | 6.9GB |
| indel | Mills&1000G  Gold standard | Homo sapiens | / | / | 83MB |

## 5.2 实验结果与分析

### 5.2.1 结果正确性测试

我们把SRR54516数据集使用GATK-Queue的运行方式和GATK-Spark的运行方式进行测试，得到图5.1所示的结果。GATK-Queue运行方式得到的多样性位点是289,849个，GATK-Spark运行方式得到的多样性位点是291,876个，其中有288,699个结果是相同的。测试结果显示，经过并行优化后的GATK-Spark方法基本保持了和原有方法基本上相同的结果。

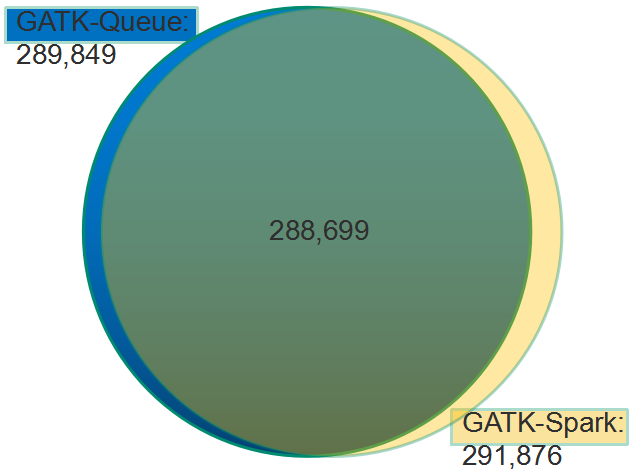


图 5.1 GATK-Spark和GATK-Queue结果正确性对比

另外，我们比较了HugeSeq、Churchill和GATK-Spark得到的多样性位点与GATK-Queue方法得到结果的差异。如表5.3所示，HugeSeq与GATK-Queue的多样性位点差异达到了20.2%，Churchill的结果和GATK-Queue相差17.3%，而GATK-Spark与GATK-Queue的差别只有1.5%。当然，HugeSeq和Churchill中对计算方法做了一定的修改，并不是完全对GATK-Queue的并行优化，因此这种差别是正常的，但是GATK-Spark和GATK-Queue存在的差别却是我们需要解释的。

表 5.3 不同方法得到的多样性位点与GATK-Queue的差别

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | GATK-Queue | HugeSeq | Churchill | GATK-Spark |
| Distinction | 0 | 20.2% | 17.3 | 1.5% |

通过我们对测试数据和源代码程序的分析，归结出GATK-Spark与GATK-Queue结果不完全相同的两个原因。第一个原因是串行算法与并行算法的不同，在GATK-Queue的源代码中，存在选取得分最高的数据情况，当得分相同时使用了随机数的方式来确定要选择的情况，而且这个随机数是一个固定的序列，对于串行算法来说，每次运行时确定的，但是在并行算法中，我们想要固定随机数序列的话就需要由master生成随机数来分发给顺序排列的数据，这种方式是比较浪费时间的，而且从原理上来说也是没有意义的，因此我们采用了在每个计算单元独立生成随机数的方式来解决。另外一个原因是区间强制切分的影响，虽然我们采用Overlap的方式来对错误切分区间的问题做了缓解，但是对于ActvieRegion的划分仍然有一定影响。从另外一个角度来讲，目前对于多样性发现还处在一个探索阶段，并不存在一定的标准，虽然GATK典型流程在这方面有一定的权威性，但也不一定就是准确的，所以存在1.5%的差异完全是可以接受的。

### 5.2.2 性能测试

确定了GATK-Spark的正确性后让我们来看一下其性能，我们在40 cpu情况下使用ganglia统计了GATK-Queue和GATK-Spark的cpu利用率，如图5.2所示，我们可以看到GATK-Queue中大部分时间cpu利用率都不高，只有在并行计算的部分才有比较高的cpu利用率，而GATK-Spark中大部分时间cpu利用率都非常高，只有在个别时间存在短暂的cpu利用率下降。

GATK-Queue的整体cpu利用率只有12.5%，而GATK-Spark的整体cpu利用率达到了81%。cpu利用率的巨大差距使得GATK-Spark的运行时间相较于GATK-Queue有很大的提升，接下来我们会测试在不同cpu个数情况下串行的GATK、GATK-Queue和GATK-Spark的运行时间情况，可以看到把GATK典型流程迁移到Spark上带来的性能上的提升。

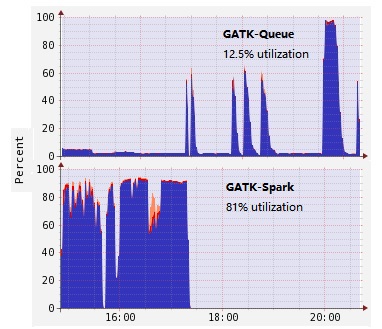


图 5.2 GATK-Spark和GATK-Queue的cpu利用率

我们在曙光的集群上对GATK-Spark和GATK-Queue以及串行GATK的运行方式进行了测试，得到图5.3所示的结果，其中蓝色线条代表GATK-Spark运行方式的结果，红色线条代表GATK-Queue运行方式的结果，绿色线条代表串行GATK的运行结果。

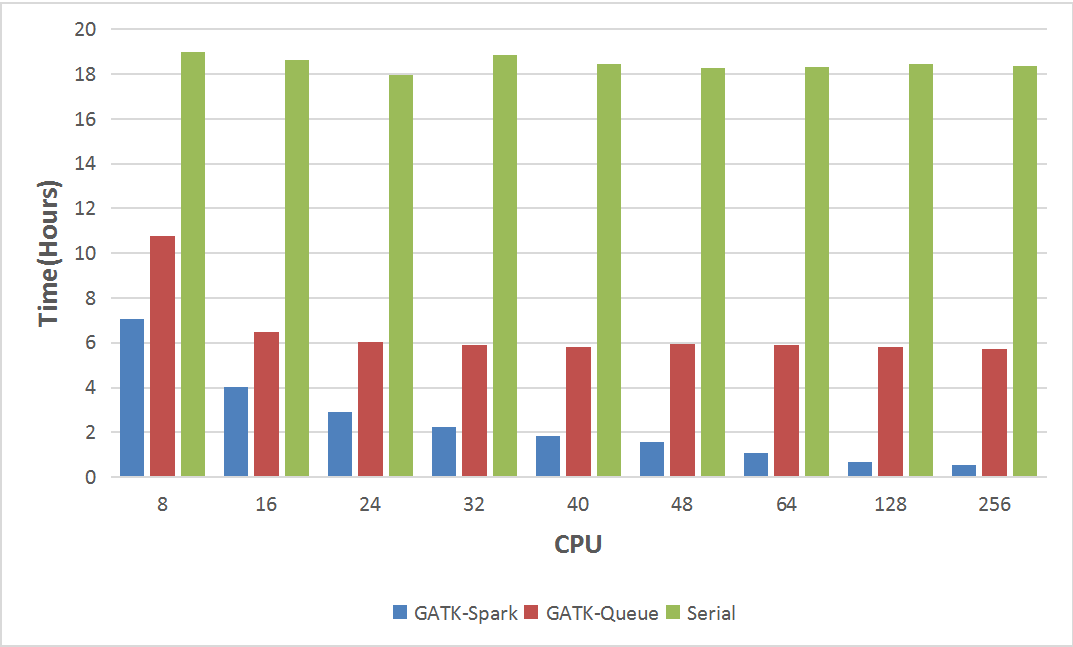


图 5.3 不同运行方式下流程的运行时间随cpu个数的变化

通过图5.3的实验结果可以看到，串行GATK由于算法是串行运行的，只能使用单个cpu的资源，即使在cpu资源增加时也不会有性能上的提升，运行时间基本保持不变。GATK-Queue运行方式在cpu个数比较少的时候时间提升较多，但是cpu个数到40以后时间上的提升基本不明显了，这与GATK官方声称的性能表现基本符合，也说明了GATK-Queue的扩展性不是很好，当然从其各个步骤来看并行的部分并不多，所以也能理解其扩展性差的原因。GATK-Spark运行方式的运行时间随着cpu个数的增加不断下降，在256cpu情况下运行时间更是达到了30分钟，相比于GATK-Queue运行方式6个小时的运行时间，性能提升了十几倍。

表 5.4 不同运行方式下流程的详细运行时间

|  |  |  |  |
| --- | --- | --- | --- |
| CPU | GATK-Spark(hour) | GATK-Queue(hour) | Serial(hour) |
| 16 | 3.995 | 6.45 | 18.63 |
| 32 | 2.206 | 5.89 | 18.82 |
| 48 | 1.558 | 5.93 | 18.26 |
| 64 | 1.034 | 5.88 | 18.28 |
| 128 | 0.667 | 5.78 | 18.44 |
| 256 | 0.501 | 5.72 | 18.34 |

GATK-Spark运行方式表现出了很好的扩展性，随着cpu个数的增加，性能得到的很大的提升，相对于GATK-Queue中串行步骤阻碍整个流程的性能提升，我们也希望看到GATK-Spark中各个步骤的并行效果。

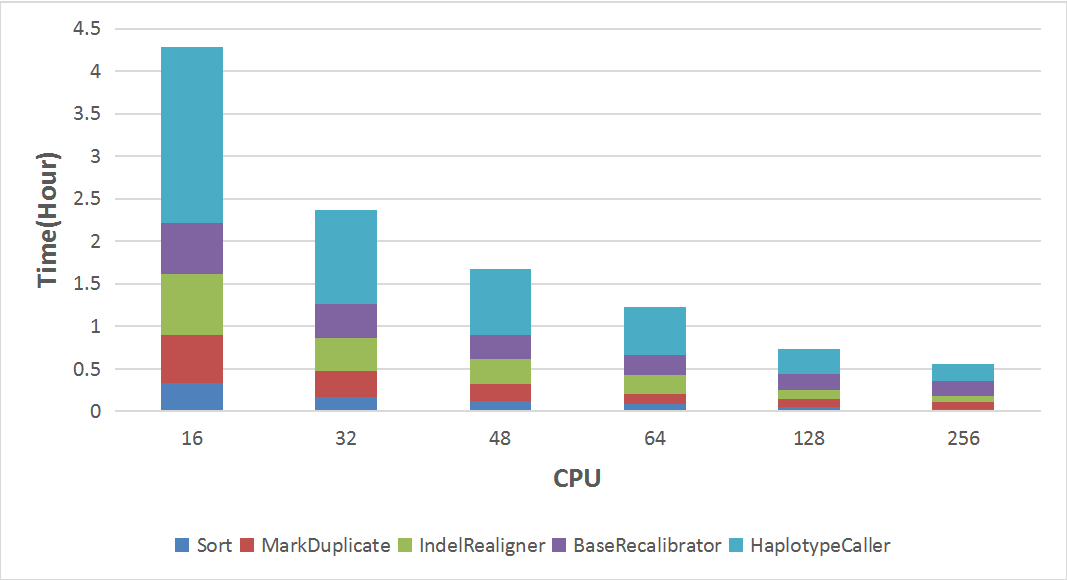


图 5.4 GATK-Spark各个步骤运行时间随cpu个数的变化

通过分析GATK-Spark中各个Stage的运行情况，我们粗略统计了各个步骤的运行时间情况，如图5.4所示。由于在Spark中，不同Stage之间不存在相关依赖时可以同时执行，所以对于各个步骤的统计时间存在一定的交叉，各个步骤的时间总和是大于真正运行时间的，但是这些交叉相对较少，各个步骤的运行时间虽然粗略，但还是能在一定程度上反映各个步骤的并行效果。通过对图5.4中各个步骤的运行时间的分析我们可以看到，随着cpu个数的增多，各个步骤的运行时间都有一定程度的下降，各个步骤的并行度相对于GATK-Queue提升非常大。但是，从另一个角度我们可以看到GATK-Spark的性能提升并不是随着cpu个数的增多线性提升的，这方面的原因还需要我们加以分析，接下来就让我们通过加速比的分析来找出GATK-Spark未达到线性加速比的原因。

### 5.2.3 加速比分析

通过性能测试中得到的运行时间统计，我们可以得到GATK-Spark和GATK-Queue运行方式的加速比，另外，参考Churchill论文中相关数据我们可以得到Churchill的加速比，经过分析统计，得到了图5.5所示的不同运行方式下的加速比。

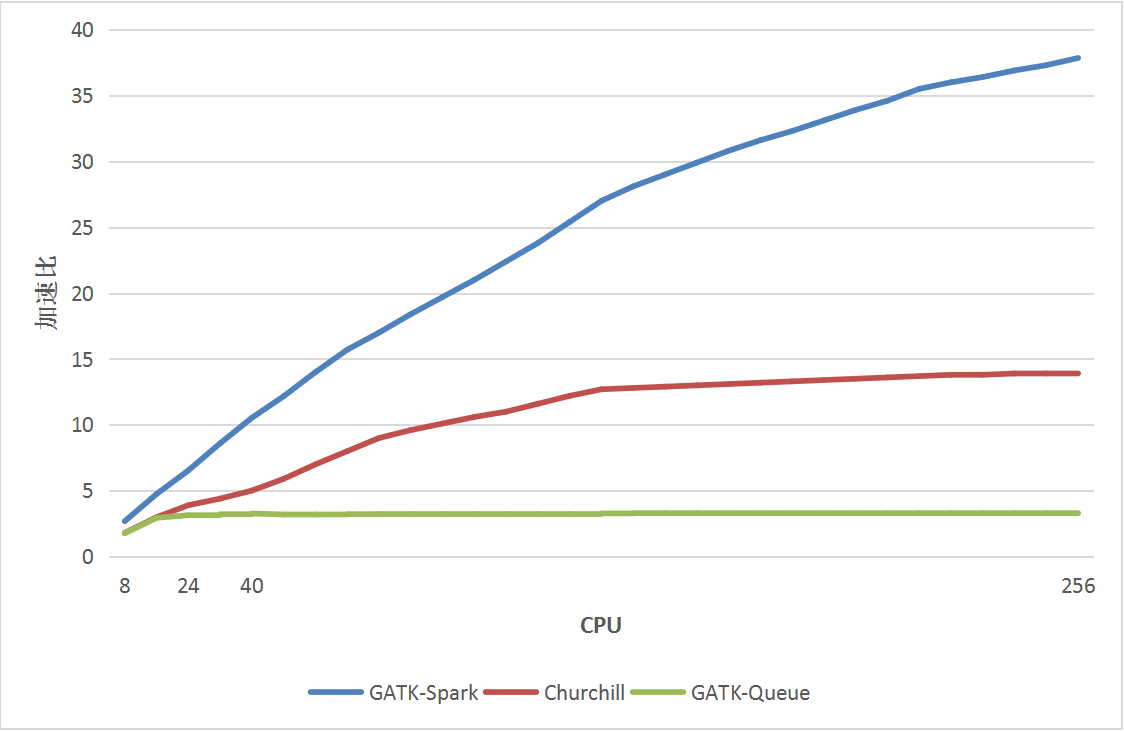


图 5.5 不同运行方式下的加速比

根据图5.5中的加速比我们可以看到，GATK-Spark运行方式的加速比相比于Churchill方法和GATK-Queue运行方式都提升很多，但随着cpu个数的增多，加速比的增长速度还是逐渐趋缓，虽然我们知道理想的加速比在并行加速方法中是很难达到的，但是还是应该分析造成加速比上升趋缓的原因。

我们使用一个Spark统计分析工具对运行的GATK-Spark进行了一个详细的分析，如图5.6所示，通过把GATK-Spark的运行时间分为了六个部分，分别是Scheduler delay、Task serialization、Network wait、Compute、Garbage collection和Output write wait。

这六个部分分别对应了不同的意义，其中Scheduler delay表示的是master节点分发任务的时间，随着节点个数的增多，任务的执行更加分散，master节点同时要分发的任务数量增多，势必会造成Scheduler delay时间比例的上升。Network wait代表网络等待时间所占的比例，随着cpu个数增加，节点个数增长，网络的使用也相应增加，网络延迟时间的比例也会相应提升。

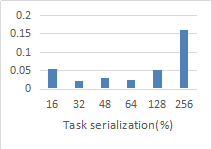
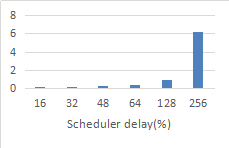
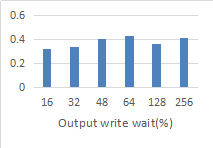
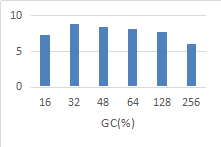
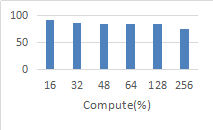
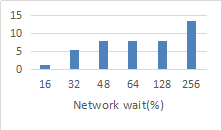


图 5.6 GATK-Spark各部分运行时间比例随CPU个数的变化

Task serialization是任务序列化的时间，根据图5.6中所示我们可以看到虽然随着cpu个数的增多Task serialization的时间比例相对上升，但是由于其所占比例相对太少，最多的时候也不到0.2%，所以这部分的时间比例基本可以忽略。另外，计算时间（Compute）、垃圾回收（GC）和输出时间（Output write wait）的比例随着cpu个数的变化基本上没有什么大的变化，计算时间的比例有一定程度的下降，当然这是在Scheduler delay和Network wait上升的情况下。

从各部分时间比例上来看，随着计算资源的增多，计算资源的相对分散造成了负载均衡和网络传输方面的时间比例的上升，这也就造成了整体任务量上的增长，使得我们并行加速的效率不能线性的增长。以上对于各个部分时间比例的分析只在一定程度上对我们的并行加速未达到线性加速比原因的一个简单分析，除了整体任务量的上升，对各个步骤详细分析才能找出其深层原因。

我们测试了GATK-Spark运行时不同接口对应的运行时间和并行度随着cpu个数的变化，如图5.7所示，不同步骤的时间比例和并行度都有很大的不同。接下来，我们把这些步骤分成三个类别具体分析，找出造成GATK-Spark未能达到线性加速比的原因。

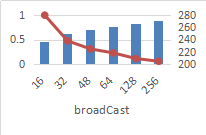
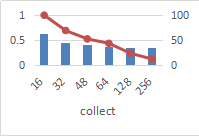
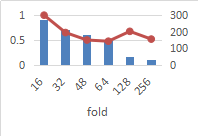
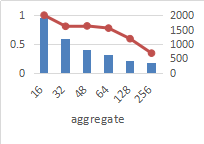
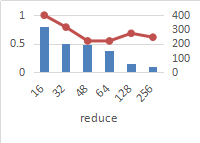
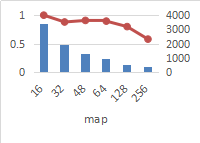
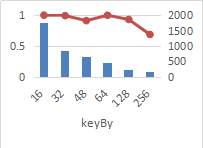
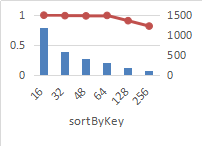
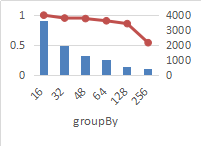




图 5.7 GATK-Spark各步骤运行时间和并行度随CPU个数的变化

根据对图5.7中各个步骤的分析，我们可以看到前四个步骤groupBy、sortByKey、keyBy和map的运行时间随着cpu个数的增加下降迅速，并行度基本能保持不变，只有在cpu个数达到256时并行度有一定程度的下降，在后续其他实验中我们可以知道，cpu个数到达一定程度后，不同cpu上的负载不够均衡的问题就会凸显出来，对并行度产生一定影响。

接下来，我们看reduce、aggregate和fold三个步骤。这三个步骤是比较类似的，都是对结果的合并操作，我们可以把它们类比成并行求和，在并行求和的算法中，我们需要不断对被求数进行求和，直到最终剩余一个数。这种方法的并行度并不能达到线性，而是只有n/log(n)的加速比。所以我们可以看到这三个步骤的并行度随着cpu个数的增加逐渐下降。

最后两个步骤collect和broadCast随着cpu个数的增加并行度迅速下降，并且运行时间基本变化不大。broadCast是一个广播的操作，由master把数据分发给所有计算节点，随着cpu个数和计算节点个数的增加，master广播的并行度不仅没有提升，而且其任务量还相对增加，所以并行度的下降也就不足为怪。collect操作按说应该是并行度很高的操作，但是由于在GATK中，collect的数据量较少，任务数量只有20几个，所以随着cpu个数的增加，其并行度并没有提升。

通过以上的分析，我们可以看到，随着cpu个数和计算节点个数的增加，整体任务量的上升和一些非线性并行的步骤的存在造成了GATK-Spark整体的加速比未能达到线性。但是也应该看到，我们的加速效果相对于GATK-Queue和Churchill有很大程度的提升，运行时间也更短。

### 5.2.4 其它测试

除了正确性和性能以及加速比的测试之外，我们在进行测试的过程中还涉及到一些其他方面的测试。

在进行性能测试的过程中，一开始我们是把机器的内存全部用上，然后逐渐增加cpu个数来进行测试。但是在实验的过程中我们发现，随着cpu个数的增加，平均到单个cpu上的内存量减少，造成了垃圾回收时间的急剧上升，甚至到后来增加cpu个数都不能减少整个流程的运行时间，所以我们就进行了图5.8所示的测试实验，通过固定节点个数和cpu总数，通过改变平均到单个cpu上内存数量来测试垃圾回收时间的变化。

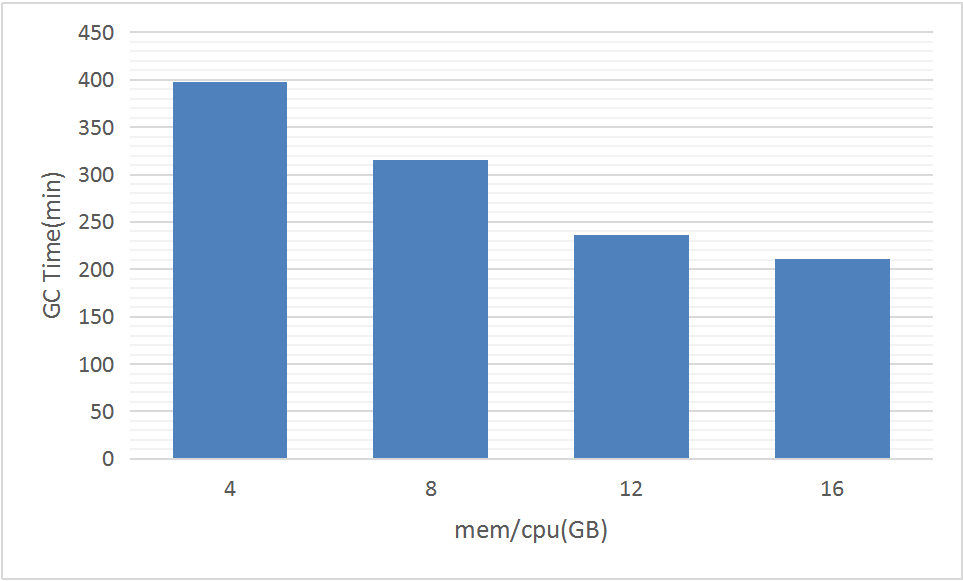


图 5.8 垃圾回收时间随着单cpu内存量的变化

经过实验测试和分析我们发现，当单个cpu对应的内存量在10GB以上时，垃圾回收时间在整个流程中的比例相对较低，而且增加内存量性能提升不明显。而在单个cpu对应的内存量在10GB以下时，垃圾回收时间在整个流程中的比例迅速上升，且对整个流程性能影响较大。所以在我们性能测试的实验中，我们把单个cpu对应的内存量固定在10GB，以减少垃圾回收对整个流程的影响。

在加速比分析一节，我们探讨了map步骤在256 cpu时并行度有下降，当时提出的原因是负载不够均衡。图5.9和图5.10中我们统计了map阶段在16 cpu和256 cpu时的执行过程，其中的每一条横线都是一个任务，横坐标是对应的时间，通过对比我们可以看到在256 cpu时负载不均衡的问题得到凸显，而在16 cpu时，由于map步骤的运行时间长，掩盖了这种负载不均衡的影响。

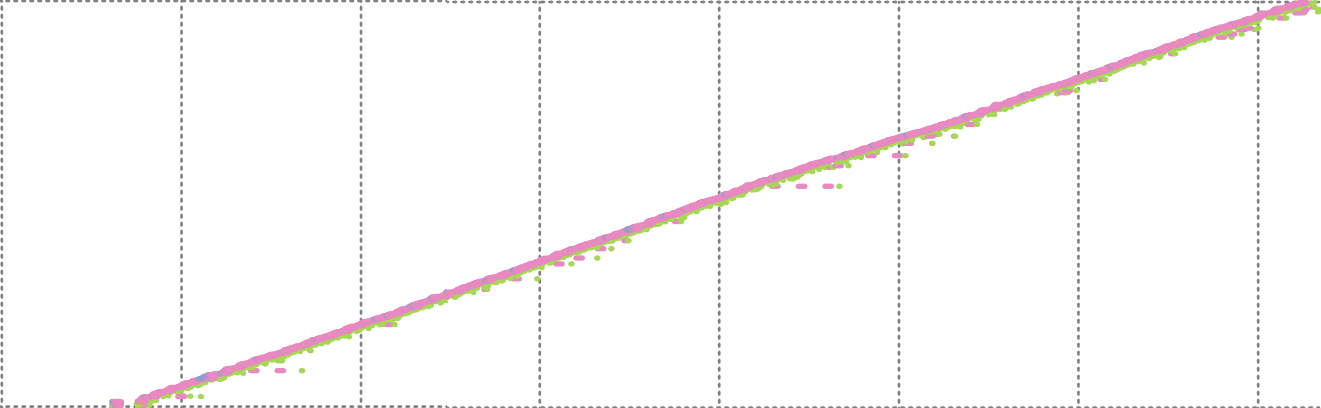


图 5.9 16 cpu下GATK-Spark中map执行图

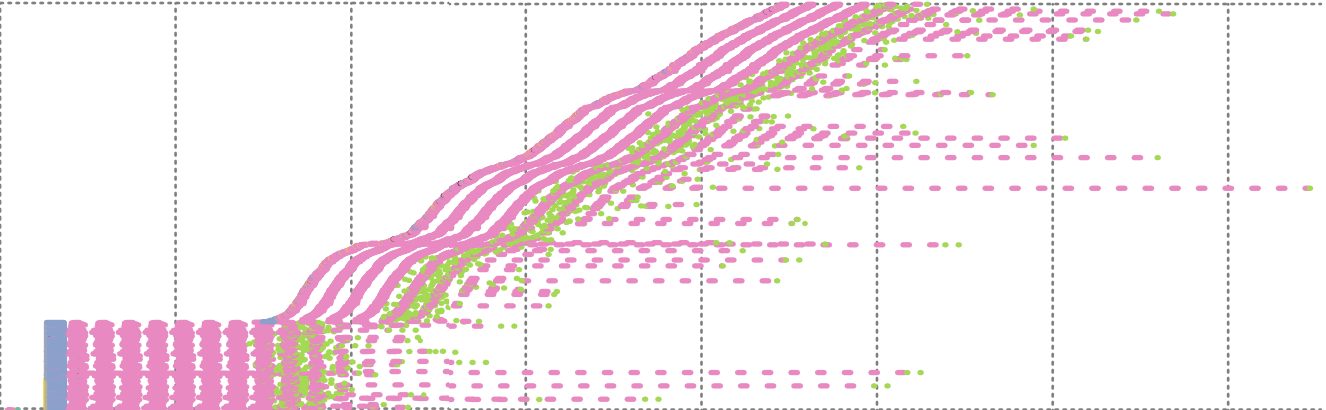


图 5.10 256 cpu下GATK-Spark中map执行图

另外，我们之前的实验都是在SRR54516数据上进行的，未考虑在数据量增大时GATK-Spark的性能是否还能够保持好的表现，所以我们增加了大数据量下的性能测试实验。通过表5.5我们可以看到，30X的数据相对于5X的数据也就是SRR54516数据在数据量上增加了6倍左右，在同等实验条件下，测试得到的运行时间也增长了6倍左右。那么我们也可以看到在数据量增长的情况下，GATK-Spark依旧能够保持其性能和扩展性等方面的优势。

表 5.5 不同数据量下GATK-Spark运行时间

|  |  |  |  |
| --- | --- | --- | --- |
| Data type | Data size | Cpu | Run time |
| 5 X | 50 GB | 256 | 0.501 h |
| 30 X | 305 GB | 256 | 2.962 h |

在整个测试工作中，我们使用了Spark的1.2.0版本，这是由于开发初期选择的比较稳定的版本，由于Spark的版本演进较快，所以我们没有不断更新版本，后续工作中我们可以考虑把整个流程迁移到最新的版本上。另外，Spark中的RDD本身具有容错的优势，当任务执行过程中出现一些Stage执行失败时，Spark中通过对其依赖RDD的分析，找出一个最优化的重新执行路径，使得其由于出错引起的任务重新提交能够最小化其影响。

## 5.3 本章小结

本章主要是对性能的评测和分析。首先介绍了实验的配置，接下来从三个方面进行了实验评测和分析，主要是正确性、性能和加速比方面，最后通过补充的其它测试实验，对性能评测和分析做了补充。整体来说，我们的实验结果良好，符合预期。

# 结束语

## 6.1 本文工作总结

随着测序技术的快速发展，测序数据分析流程的处理速度逐渐不能匹配数据量的发展，对测序数据分析流程的加速工作就势在必行。本文从二代测序数据分析流程入手，通过对二代测序数据分析流程相关知识的了解，选取使用广泛且权威性较高的GATK典型流程来作为我们并行优化的选择。

我们以GATK-Queue作为并行优化的基础，通过对流程的相关测试和具体步骤的算法详解，找出了阻碍流程并行加速的瓶颈所在，这三个瓶颈分别是大文件的切分合并、Picard中串行步骤和GATK中Scatter-Gather。基于性能和效率的考虑，在加上Spark在三个方面的优势：Spark中的RDD很好的解决了大文件切分合并的问题、加州大学伯克利分校开发的ADAM为在Spark上操作测序相关的数据格式提供了基础、Spark中提供了上百个API操作，为高效编写Spark程序提供了基础，我们提出了基于Spark平台来改写GATK典型流程。

由于GATK典型流程相对比较复杂且不同步骤之间差异较大，所以我们提出了针对不同步骤使用不同方式的Spark改写。对于一些较为简单的步骤和算法，我们在保证正确性的基础上完全重写成Spark的处理方式，而对于一些逻辑复杂且代码量大的步骤，我们提出了保留原有计算逻辑，改写数据切分部分的方法。最终，我们设计并实现了GATK-Spark，把GATK典型流程整体迁移到Spark上来。

最后，通过性能的测试和分析，我们在保证的结果正确性的基础上得到了性能极大的提升，而且扩展性方面也有突出的表现。另外，我们分析了GATK-Spark加速比未达到线性的原因，为以后想进一步提升GATK-Spark的性能提供基础。

## 6.2下一步工作

本文通过把一个二代测序数据分析流程GATK改写成Spark的处理方式，实现了GATK典型流程的并行优化。虽然，我们设计实现的GATK-Spark在性能和扩展性上都有很好的表现，但还有一些工作可供我们未来继续的。

首先，在正确性测试方面。我们主要是把GATK-Spark的结果和GATK-Queue进行比较，但是在测序数据分析中，是存在一些数据集的标准结果的，对于正确性方面的测试，我们不仅应该和原有程序进行比较，还应该使用数据集的标准结果来进行验证[]。

另外，在并行优化方面。由于一些步骤复杂性高，完全重写成Spark处理方式需要大量的人力物力，因此我们对这些步骤是保留大部分原有处理逻辑，只对数据切分部分进行改写。如果未来有可能，从根本上设计并实现契合Spark平台的算法和代码不仅可以提高性能、减少不必要操作，还能在正确性方面进行创新。

最后，在功能方面。测序数据量持续增长，那么未来想要找出某一个特定数据就会相对困难，Spark提供了Spark SQL的方式，未来我们可以把测序数据都存储成Spark中的方式，查找特定数据时只需要写一段SQL语句就可以很快的得到结果。

参考文献

[ 1 ] 解增言, 林俊华, 谭军,等. DNA测序技术的发展历史与最新进展[J]. 生物技术通报, 2010(8):64-70.

[ 2 ] Illumina Platform. <http://www.illumina.com/>

[ 3 ] Che D, Safran M, Peng Z. From Big Data to Big Data Mining: Challenges, Issues, and Opportunities[M]// Database Systems for Advanced Applications. Springer Berlin Heidelberg, 2013:1-15.

[ 4 ] Ghemawat B S, Gobioff H, Leung S T. The Google File System. SOSP[J]. 2010.

[ 5 ] Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters.[J]. In Proceedings of Operating Systems Design and Implementation (OSDI, 2004, 51(1):107-113.

[ 6 ] Chang F, Dean J, Ghemawat S, et al. Bigtable: A Distributed Storage System for Structured Data[J]. Acm Transactions on Computer Systems, 2008, 26(2):205--218.

[ 7 ] Zook J M, Chapman B, Wang J, et al. Integrating human sequence data sets provides a resource of benchmark SNP and indel genotype calls[J]. Nature Biotechnology, 2014, 32(3):246-51.

[ 8 ] Nielsen R, Paul J S, Albrechtsen A, et al. Genotype and SNP calling from next-generation sequencing data[J]. Nature Reviews Genetics, 2011, 12(6):443-451.

[ 9 ] Illumina Sequencing Data Sheet. Getting Started with RNA-Seq Data Analysis.

[ 10 ] Trapnell C, Roberts A, Goff L, et al. Differential gene and transcript expression analysis of RNA-seq experiments with TopHat and Cufflinks.[J]. Psychopharmacology, 2012, 7(3):562-78.

[ 11 ] 1000 Genomes. <http://www.1000genomes.org/data>

[ 12 ] Mardis E R. A decade's perspective on DNA sequencing technology.[J]. Nature, 2011, 470(7333):198-203.

[ 13 ] McKenna A; Hanna M; Banks E; Sivachenko A; Cibulskis K; Kernytsky A; Garimella K; Altshuler D; Gabriel S; Daly M; DePristo MA. The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data[J]. Genome Research, 2014, 20(9):1297-1303.

[ 14 ] Broad Institute GATK. <https://www.broadinstitute.org/gatk/>

[ 15 ] Garber M, Grabherr M G, Guttman M, et al. Computational methods for transcriptome annotation and quantification using RNA-seq[J]. Nature Methods, 2011, 8(6):469-77.

[ 16 ] Li H, Durbin R. Fast and accurate short read alignment with Burrows-Wheeler transform.[J]. Bioinformatics, 2009, 25(14):1754-1760.

[ 17 ] Kwok S, Higuchi R. Avoiding false positives with PCR.[J]. Nature, 1989, 339(6221):237-238.

[ 18 ] Mills R E, Luttig C T, Larkins C E, et al. An initial map of insertion and deletion (INDEL) variation in the human genome.[J]. Genome Research, 1994, 28(28):23-53.

[ 19 ] Nielsen R, Paul J S, Albrechtsen A, et al. Genotype and SNP calling from next-generation sequencing data[J]. Nature Reviews Genetics, 2011, 12(6):443-451.

[ 20 ] Klus P, Lam S, Lyberg D, et al. BarraCUDA - a fast short read sequence aligner using graphics processing units[J]. Bmc Research Notes, 2012, 5(1):1-7.

[ 21 ] M. Carneiro, “Accelerating variant calling,” Broad Institute, Tech. Rep., 2013.

[ 22 ] Niemenmaa M, Kallio A, Schumacher A, et al. Hadoop-BAM: directly manipulating next generation sequencing data in the cloud.[J]. Bioinformatics, 2012, 28(6):876-7.

[ 23 ] Massie, MattNothaft, FrankHartl, et al. ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing[J].

[ 24 ] Kelly B J, Fitch J R, Hu Y, et al. Churchill: an ultra-fast, deterministic, highly scalable and balanced parallelization strategy for the discovery of human genetic variation in clinical and population-scale genomics.[J]. Genome Biology, 2015, 16(1):6-6.

[ 25 ] Lam H Y, Pan C, Clark M J, et al. Detecting and annotating genetic variations using the HugeSeq pipeline.[J]. Nature Biotechnology, 2012, 30(3):226-9.

[ 26 ] Auwera G A, Carneiro M O, Hartl C, et al. From FastQ Data to High-Confidence Variant Calls: The Genome Analysis Toolkit Best Practices Pipeline[J]. Current protocols in bioinformatics / editoral board, Andreas D. Baxevanis. [et al.], 2013, 11(1110):11.10.1-11.10.33.

[ 27 ] Picard Tool. <http://broadinstitute.github.io/picard/>

[ 28 ] Garrison E, Marth G. Haplotype-based variant detection from short-read sequencing[J]. Quantitative Biology, 2012.

[ 29 ] Yoon B J. Hidden Markov Models and their Applications in Biological Sequence Analysis[J]. Current Genomics, 2009, 10(6):402-15.

[ 30 ] Apache Spark. <http://spark.apache.org/>

[ 31 ] Zaharia M A. An architecture for fast and general data processing on large clusters[J]. Dissertations & Theses - Gradworks, 2013.

[ 32 ] Apache Hadoop. <http://hadoop.apache.org/>

致谢

时光飞逝，岁月如梭！转眼间研究生生涯就要结束。在此研究课题即将完成之际，我怀着一颗虔诚、感恩的心对三年来帮助我、教育我、陪伴我的各位老师们、同学们以及家人们表示诚挚的感激和感谢！

回想三年前入学的那一刻，恍如昨日。中国科学院大学怀柔校区虽然比较偏远，还处在建设当中，但是有室友和同学们的陪伴，再加上有山有水的好风景，让我们学习中伴着快乐，成长中汲取着知识。感谢中国科学院大学的各位老师和同学，让我能在这一年的上课中收获知识、友情和快乐。

之后两年在实验室开始了课题的研究，从对生物信息中测序方面的知识一无所知到逐渐了解，再到修改其流程实现并行加速。感谢谭老师的谆谆教导和事无巨细的指导，是您让我明白在遇到难题时更要保持清醒的头脑，不能盲目的去试验，而要从一些可能的信息中去寻找问题的原因。还要感谢张老师和李旭，你们对于基因测序知识方面的深入理解为我能快速学习到一些相关生物学知识提供了很大的帮助，如果不是你们的帮助和指导，我可能还陷入在背景知识的泥淖中不能自拔。另外还要感谢王炳琛师弟，帮我解决了一部分抽取代码上的任务，让我能把更多的精力放在提高并行化的工作上。

除了繁忙的科研生活，身体锻炼也是必不可少的，感谢主任张佩珩老师和霍志刚老师为我们提供的篮球场地，虽稍显简陋，但中午饭后挥洒着汗水的乐趣能让我们仿佛置身豪华球馆，忘却不开心和不快乐。

感谢我们组的各位师兄师姐，张中海师兄、李强强师兄、骆裕龙师兄、赵喜全师兄、苑鲁峰师兄和张秀霞师姐，你们探索出的道路使我能避免走很多岔路。还要感谢刘闯、郭浩强、王元荣、刘军红和邵慧子，能一起来到高性能实验室是我们的缘分。当然还要感谢王炳琛、曾平和周可人三位师弟，你们积极勤奋的态度也激励着我。

最后，感谢我的家人和女朋友，愿明天会更好！

李红印

2016年5月

作者简介

姓名：李红印 性别：男 出生日期：1989.06.12 籍贯：河北邯郸

2013.9 – 2016.6 中国科学院计算技术研究所研究生

2009.9 – 2013.6 华南理工大学计算机科学与技术专业本科生

**【攻读硕士学位期间参加的科研项目】**

[1] 973课题“高通量计算系统的构建原理、支撑技术及云服务应用 课题2” (2011CB302502)，2011年1月-2015年8月

[2] 973课题“面向深度测序大数据量的计算模型与体系结构研究”(2012CB316502)，2012年1月-2016年12月

**【攻读博士学位期间的获奖情况】**

[1] 2015年获得中科院斯伦贝谢杯debug大赛“特等奖”

[2] 2015年获得中科院计算所“所长优秀奖”