doi:10.3772/j.issn.1002-0470.2015.12.001

KSI:面向 TB 级别的 DNA 序列匹配软件库①

赵喜全②[*,**] 李 旭[*] 吕慧伟[*,**] 谭光明[*]

([*]中国科学院计算技术研究所高性能计算机研究中心 北京 100190)

([**]中国科学院大学 北京 100049)

摘 要 为了满足对不同物种进行 DNA 序列分析的需求和适应 DNA 序列数据的快速增长,针对目前 DNA 序列分析软件大都各自实现一套序列存储和查询功能,工作重复且没有考虑并行性、扩展性和分布式系统或环境的缺陷,基于 DNA 序列分析的基本操作 k-mer 匹配,设计并实现了一个面向 TB 量级的 DNA 序列匹配软件库——k-mer 查找接口(KSI)。KSI 提供了一套分布式环境下的编程接口,并且针对生物计算领域的 DNA 序列匹配进行优化。实验显示,KSI 为 DNA 序列匹配提供了一个高效的解决方案。

关键词 生物信息学, k-mer 匹配, DNA 序列处理, 应用程序编程接口

0 引 言

DNA 测序技术给生物学带来了实质性的革命。1990 年人类基因组计划的启动是人类为了探索自身奥秘迈出的重要一步。2001 年第一份人类基因组草图[1]的发表,是人类基因组研究的一个里程碑。人类基因组图谱可使许多领域受益,可帮助我们认识特定病毒和疾病之间以及致癌基因和癌症之间的关系,促进法医学的发展,促进生物燃料等能源的开发,等等。尽管第一个基因组测序花费近一亿美金[2],但之后的测序成本迅速下降,使这项技术能够被更多人使用。下一代基因测序技术会使测序的吞吐率更高,有望进一步降低成本。

DNA 测序分为两种不同的方法:从头测序和重测序。前者不需要任何参考序列即可对某个物种进行测序,而后者则用一个关系紧密的物种的基因来指导测序的过程。从算法复杂度上看,从头测序是 NP 难题,而重测序则相对简单。对于认识自然界的生物多样性来说,从头测序是最关键的方法。目前从头测序主要有以下几种方法:贪婪测序算法、重叠布局共识(overlap layout consensus, OLC)测序算法和欧拉测序算法[3],这几种测序算法的最基本的操作都是两个 DNA 序列(k-mer)之间的匹配,即 k-mer 匹配。事实上,除了从头测序算法以外[4-7],基因标注(genome annotation)、相似性搜索、基因映射(genome mapping)和 k-mer 频谱分析等很多其他的序列分析算法的核心操作也都是 k-mer 匹配。但是上述应用往往是各自实现自己的 k-mer 匹配算法,这样不仅重复工作很多,而且没有考虑到并行性、扩展性和分布式系统的需求。例如,KMC 2[8]是一款优秀的 k-mer 匹配和统计的软件包,但是受限于单机的性能,它可以有效处理的 DNA 序列在 100GB 量级,然而,由于它不支持分布式环境,性能扩展性较差,无法应对 TB 甚至 PB 级别 DNA 序列查找的挑战。因此,基于消息传递接口(message passing interface, MPI)并行编程技术,我们设计并实现了一个面向 TB 级别的 DNA 序列匹配软件库,即 k-mer 查找接口(k-mer searching interface, KSI)。它抽象出一系列底层的 k-mer 匹配核心操作,向上提供标准

① 973 计划(2012CB316502, 2011CB302502)资助项目。

② 男,1984 年生,博士生;研究方向:并行计算,高性能计算;联系人,E-mail: zhaoxiquan@ncic.ac.cn

(收稿日期:2015-05-25)

的应用程序编程接口,并且后续可以针对不同的高性能计算硬件如现场可编程门阵列(field programming gate array, FPGA)、通用计算图形处理器(general-purpose computation on graphics processing units, GPGPU)或 Intel 集成众核架构(many integrated core, MIC)等进行优化。程序员在开发新的 DNA 序列分析程序时,只需直接使用 KSI 提供的标准编程接口就可以写出高效可移植的 k-mer 匹配算法,而不需要考虑底层的分布式数据存储、数据索引和并行查询等实现,这大大提高了程序员的开发效率。

1 KSI 数据模型

KSI 的核心数据模型是分布式的映射,提供键、值的集合。对于 k-mer 匹配来说,每个键是被查询的 k-mer,值则是原数据集中的序列片段集合。在这个映射关系中,每个键可以对应一个或者多个值。

如图 1 所示,在 KSI 中,DNA 片段被组织成记录(records),一个记录包含了以下几个域:DNA 序列本身、原 DNA 序列的个数以及该 DNA 序列的逆补序列个数。不同的记录被组织到一个或者多个表(table)中,每个表在 KSI 内部被排序和索引。不同的表被组织到一个或者多个数据库(database)中。在图 1 中,位于图右边的 DNA 序列集合被组织成 KSI 数据库中的记录,当用户查询(queries)以"AACC"为前缀的 k-mer 时,KSI 运行环境会向用户

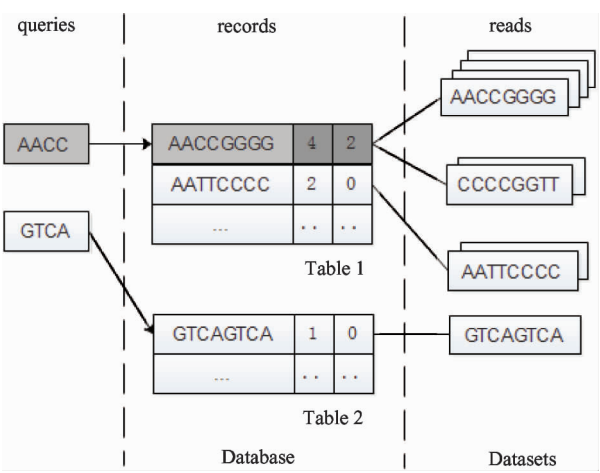

图 1 KSI 数据模型

返回 Table 1 中的记录{AACCGGGG, 4, 2},即数据库中符合该查询条件的 4 个"AACCGGGG"和 2 个"CCCCGGTT"。

KSI 在设计时充分考虑了在分布式环境下的扩展性,最多可以支持存储几百亿条 DNA 序列作为值的集合。同时 KSI 还提供了针对不同键长的查询功能。下面对 KSI 数据库中的记录、表和数据库分别作一下介绍。

1.1 记录

在测序过程中,为了容错的需要,测序仪器得到的 DNA 序列往往有多个拷贝。在 KSI 数据库中,并不需要存储这所有的拷贝,相同的 DNA 序列可以组织成同一个记录,并且用一个记数来代替。另外,由于 DNA 本身互补双螺旋结构的一个重要的特性是逆补序列的存在,在理想的测序情况下,在序列集合中的每个片段都会有一个对应的逆补序列存在。所以 KSI 数据库中的 DNA 序列被组织成记录的形式,每个记录是一个三元组:DNA 序列,原序列记数,逆补序列记数。记录是 KSI 数据管理中的最基本单位。

用户提供的 DNA 序列被逐个输入数据库时,存储一个序列 s 分两步完成:(1)在原序列所在的记录上将原序列记数加 1,如果不存在则新建该记录;(2)找到该序列的逆序序列 s_{rc} 所在记录并在将该记录的逆补序列记数加 1(同样,如果不存在则新建)。这么做的好处是,当用户需要查询以某个 k-mer(记为 q)为后缀的所有记录时,可以将该后缀查询转化为相应的以 q 的逆补序列 q_{rc} 为查询关键字的前缀查询。以图 1 为例,查找所有以"GGTT"为后缀的记录等价于查找所有以"AACC"为前缀的记录。因此,我们只需要对所有记录按照前序进行排序和索引,就可以对上层应用同时提供前缀查询和后缀查询的功能。

1.2 表

在实际 DNA 分析应用中,往往会需要将 DNA 序列数据集按照物种分类或者按照输入集合分类。KSI 将不同类型的记录组织到不同的表中,用户对某个记录的查询、删除或者合并的操作,可以只针对某个表进行,方便了用户更好地组织其数据。

1.3 数据库

KSI 还提供了数据库这个抽象数据结构来表示表的集合。用户可以将不同的表放到同一个数据库中,当用户查询某个数据库时,将会搜索该数据库中的所有表并返回查询结果。

至此,KSI 中的 DNA 序列被组织成不同的层次:记录(KSI _ record _ t)、表(KSI _ table _ t)和数据库(KSI _ db),用户可以按照需要在不同层次上对 DNA 序列进行操作。

2 应用程序编程接口设计与实现

如图 2 所示,KSI 软件库为基因从头测序、基因标注和基因映射等提供了 DNA 分析支持,包括对 DNA 序列的存储、索引和查询,其常用的应用程序编程接口(API)如表 1 所示。

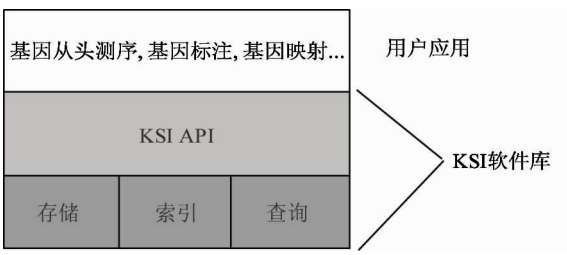

图 2 用于 DNA 分析的 KSI 软件库

表 1 KSI 主要的应用程序接口

API 功能类型	API 名称	功能描述
初始化函数	KSI _ db _ init	初始化数据库
	KSI _ table _ init	初始化数据表
添加 DNA 序列	KSI _ table _ add _ string	添加 DNA 序列到表中
	KSI _ db _ commit	提交更改到数据库
查询 k-mer	KSI _ query _ db _ start	开始查询
	KSI _ query _ db	查询 k-mer
	KSI _ query _ db _ finish	结束查询

KSI 的内部实现主要包括以下三个部分:分布式数据划分、本地数据存储和索引以及 KSI 查询,下面分别进行说明。

2.1 分布式数据划分

KSI 是针对多核集群设计的,所有数据都存储在分布式内存中。如果没有全局的数据划分规则,那么在查询某个 k-mer 时,需要将这个查询条件广播到所有的处理器,并等待所有的处理器返回查询结果,这么做是非常低效的,尤其是当处理器个数很大时,查询的速度会受限于速度最慢的处理器。相反,如果有全局的数据划分规则,当一个查询到来时,只需要根据这个规则就可以确定哪些处理器符合查询条件,这样就大大缩小了查询的范围,提高了查询效率。

KSI 内部使用的是基于分布式排序划分点的划分规则。在所有 DNA 序列都已经添加到数据库之后,对整个数据库作一次排序,同时记录将所有 DNA 序列 p 等分的位置的划分点,根据这些划分点来移动每个处理器上的数据到目的位置。使用排序后的划分点作为动态划分规则既可以支持对不同长度的 k-mer 进行查询,也可以使划分很均匀。动态划分规则增加了分布式排序的开销,但是对于特定的 DNA 序列数据集,数据划分时排序只需要进行一次,后面大部分时间是花在查询上,所以这样的开销是可以容忍的。

KSI 的数据划分则是以表为单位进行的。如图 3所示,划分向量(splitter vector)将 KSI 表划分成两个层次。KSI 表存储的第一层是用于划分处理器之间数据的划分向量(记为 splitter[ ]),对于 p 个处理器,划分向量的长度是 $p-1$,在每个处理器上有一份相同的拷贝。在提交更改到数据库函数中(KSI _ db _ commit),所有处理器上的 DNA 序列被排序,使得对于任意两个处理器 P_i 和 P_j,如果 $i<j$,那么 P_i 上所有的DNA序列的字典序都小于 P_j 上

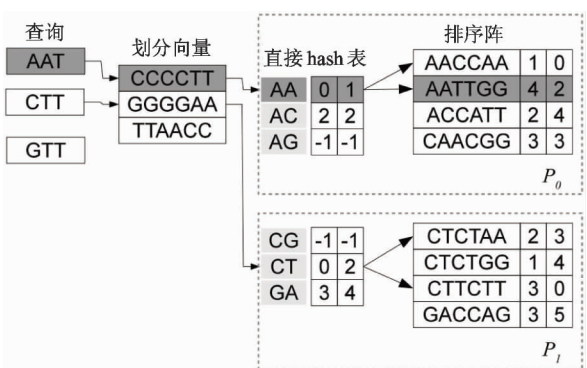

图 3 KSI 表的内部存储层次

DNA 序列。划分向量在排序完成时被赋值，splitter[i]被赋值为 P_i 上最大的一个 DNA 序列，每个处理器都保存一个相同的划分向量拷贝。

我们使用直方图排序(histogram sort)[9]来对 KSI 中的数据进行排序。如图 4 所示，直方图排序通过迭代查找 $p-1$ 个划分点来将整个数据集均匀划分成 p 份。算法从猜测 k 个划分点($k \geqslant p-1$)开始，每次猜测被称为一次探测(probe)，该猜测被广播给所有的处理器，然后所有的处理器试图从这 k 个候选划分点中选出 $p-1$ 个满足排序结束条件的划分点。排序的结束条件是划分后的 p 个区间大致包含了 n/p 个 DNA 序列。定义一次探测的不平衡率如下：令 $average = n/p$，被划分的 p 个候选区间中各自包含的 DNA 序列数记为 n_i，$0 \leqslant i < p$，不平衡率 $R_{\text{screw}} = \max\{|n_i - average| + 1, 0 \leqslant i < p\}$。如果不平衡率没有符合要求，则增加 k 的个数重新开始一次探测。只要迭代次数过多，直方图排序总能得到 R_{screw} 很接近 1 的探测，不过排序的时间也相应变长。在实际应用中，并不需要百分之百的完全平衡，因为系统本身在执行过程中也存在着其他的不平衡因素。在 KSI 中，排序的结束条件是 $R_{\text{screw}} \leqslant 1.1$。

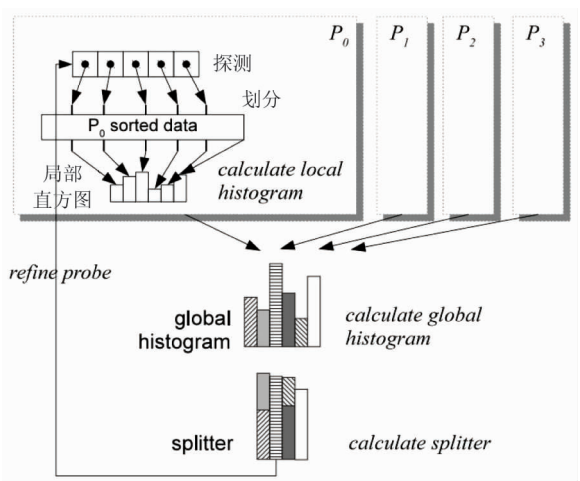

图 4 直方图排序示意图

在实际使用过程中，当处理器规模很大时，我们发现 Solomonik 和 Kale 的探测方法[9]比起 Kim[10]等人的有更好的收敛性。令 $R_{\text{ideal}} = 1.1$，$step = k$，在 KSI 内部的直方图排序的步骤如下：

(1)每个处理器将本地的数据排序，找到所有 DNA 序列中字典序最小的序列 $seq_{\min}$ 和最大的序列 $seq_{\max}$；每个处理器初始化首次探测，即等分区间 $[seq_{\min}, seq_{\max}]$ 的 k 个划分点，初始化 $R_{\text{screw}} = \infty$。

(2)每个处理器根据 k 个划分点统计被划分成的 $k+1$ 个区间中 DNA 序列的个数。

(3)统计全局的落在每个 $k+1$ 区间中 DNA 序列的个数，将 p 个处理器的结果合并。

(4)将这 $k+1$ 个区间合并成 p 个，使得每个区间的序列数 n_i，$0 \leqslant i < p$ 尽可能地接近 average。

(5)计算 $R_{\text{screw}} = \max\{|n_i - average| + 1, 0 \leqslant i < p\}$，如果 $R_{\text{screw}} \leqslant R_{\text{ideal}}$，转到(8)(探测结束)，否则转到(6)。

(6)标记满足 $|n_i - average| + 1 \leqslant R_{\text{ideal}}$ 的区间，同时统计总的满足条件的区间个数为 s，则总的不满足条件的区间个数 $u = p - s$。

(7)生成新的探测。令 $k = k + step$，旧的探测已经确定了 $p-1$ 个候选划分点(其中 s 已经满足最终条件)，在剩下的 u 个区间均匀挑选 $k-(p-1)$ 个候选划分点，转到(2)。

(8)根据探测得到的划分向量划分各个处理器的数据，通过消息传递接口(MPI)全交换通信交换数据。

(9)对全交换后得到的本地数据再次进行快速排序，至此所有处理器的数据都已经全局有序，排序结束。

2.2 数据本地索引和查询

KSI 表存储的第二层是本地存储的 DNA 序列。这一层的存储可以有多种选择，比如散列表、有序数组和前缀树等，在 KSI 中我们采用的是散列表和有序数组结合的方式，散列表对数据做部分索引，有序数组存储 KSI 的记录作为冲突链。在分布式排序结束后，单个处理器上的所有 DNA 序列已经是一个有序数组，我们要做的是对其前 L 个字符进行直接定址并记录在散列表中。以图 3 为例，分布式数据划分和排序结束后，P_0 上的 DNA 序列数组已经有序(记为 $s[\ ]$)，对前 $L=2$ 个字符进行统计，登记到直接定址的散列表中。比如 $s[0]$ 和 $s[1]$ 的位置被统计到散列表的第一项“AA”处，即以“AA”开头的

DNA 序列在 s 中开始的位置是 0,结束的位置是 1。类似地,$s[2]$ 被统计到“AC”下,$s[3]$ 被统计到“CA”下(中间的“AG”到“AT”的值将都是{-1,-1})。

在这一层的查找先是通过散列表找到查询结果所在有序数组中的位置范围,然后在该范围中搜索符合查询条件的结果。如图 3 所示,查找在 P_1 查找所有前缀为“CTT”的片段时,首先通过 hash _ table 找到以“CT”为前缀的所有片段所在的位置范围(从 array[0]到 array[2]),然后在上述范围中找到实际的匹配序列“CTTCTT”。

我们还对以上直接定址的方法做了进一步改进。所有处理器上的 DNA 序列都全局有序,处理器 P_i 上的 DNA 序列都在 splitter[$i-1$]和 splitter[i]之间。对于 p 个处理器,当 p 大于 5 时,肯定存在两个相邻的 splitter[ ]元素的第 1 位碱基字符相同(碱基只有 4 种)。改进后的直接定址方法取 splitter[$i-1$]和 splitter[i]的从第一个不相同的前缀字符开始往后数 L 个字符作为直接定址的对象。假设 splitter[$i-1$]和 splitter[i]最大的相同前缀个数为 d_i。当 p 增大时,d_i 也随之增大。d_i 的值会受到 DNA 序列分布模式的影响,假设所有 DNA 序列是完全均匀分布的情况下,所有的 d_i 都相同(记为 d),前 d 个碱基的排列数为 4^d,每个处理器的序列个数占总序列的比例 $\approx 1/4^d$,而根据排序划分的结果,这个比例是 $1/p$,所以 $1/4^d \approx 1/p$。$p = 1024$ 时,$d \approx \log_4(p) = 5$。

2.3 KSI 查询

图 5 给出了 KSI 查询的内部实现流程。其中以 KSI _开头的函数是提供给用户的接口,而以小写字母 ksi _开头的函数是 KSI 的内部函数实现。

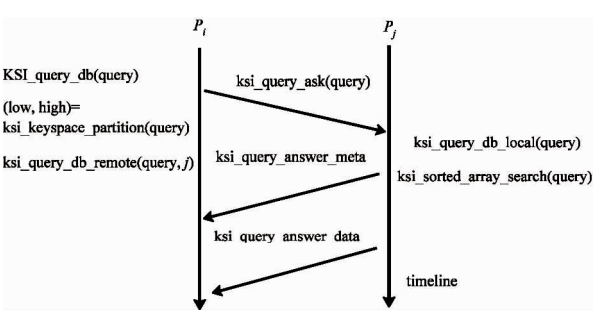

图 5 KSI 查询的内部实现

查询过程按照时间顺序分为以下几个步骤:

(1)用户代码在 P_i 通过 KSI _ query _ db 函数提交一个查询(query)。

(2)KSI 内部调用 ksi _ keyspace _ partition 函数在划分向量中查找满足查询条件的处理器号范围(low, high)。

(3)对于 low $< j <=$ high,如果 $j = i$,则调用 ksi _ query _ db _ local 查找本地有序数组返回符合查询条件的序列(图中未画出);如果 $j \neq i$,则调用 ksi _ query _ db _ remote 函数将 query 发送到远程处理器 P_j。

(4)通过 ksi _ query _ ask 函数发送 query 到 P_j。

(5) P_j 调用 ksi _ query _ db _ local 查找本地符合查询条件的序列。

(6) P_j 调用 ksi _ sorted _ array _ search 查找本地有序数组,返回符合查询条件的序列。

(7) P_j 调用 ksi _ query _ answer _ meta 将查询结果的元数据发给 P_i(主要是查询结果的数据长度);

(8)如果查询结果的数据长度不为 0,则 P_j 调用 ksi _ query _ answer _ data 将查询结果数据发给 P_i。

2.4 其它优化

2.4.1 DNA 序列压缩

一个 DNA 序列可以看做是在{A, C, G, T}有限集上的字符串。因为每个碱基的取值只有 4 种,所以每个字符可以用 2 位来表示,用 00, 01, 10, 11 分别表示 A, C, G, T。这样,原先需要 1 个字节存储的一个碱基现在只需要 2 位来存储,总共压缩了 1Byte/2bit = 4 倍。在 KSI 中,用户代码部分使用的是字符表示,而在 KSI 内部实现的字符串比较和排序使用的都是二进制的数组。

2.4.2 内存分配

当用户使用 KSI _ table _ add _ string 函数往 KSI 表中添加 DNA 序列时,所有的 DNA 序列被暂时缓存下来等待被重新排序和划分。对于每个处理器上亿的 DNA 序列,如果每次添加一个 DNA 序列都新申请一块内存,不但非常低效而且有可能导致内存碎片化。对此,我们的优化是每次分配存储 1K 个

DNA 序列的缓冲区,对内存的申请和重分配都是批量进行。

3 实验评测

3.1 实验环境和基准测试程序

实验的数据集是由一个随机字符生成函数人工生成的随机 DNA 序列。每个随机字符取 POSIX random 函数生成的随机数中间的 2 位数字,00、01、10、11 分别代表 A、C、G、T。生成 DNA 序列的长度可以由用户指定。测试时,包括输入 DNA 片段数据和被查询的 k-mer 都是由该随机字符生成函数生成,其长度分别记为 r 和 q,比如“$r100q20$”指的是数据库中插入的 DNA 片段长度为 100,而查询的 k-mer 长度是 20。

实验使用一个随机查询程序用于测试 KSI 的查询吞吐率。该程序先随机生成 n 个长度为 r 的 DNA 片段添加到 KSI 数据库中,然后对该数据库进行 m 个查询,每次查询是一个长度为 q 的随机 k-mer。查询吞吐率等于所有进程查询的 k-mer 总个数除以查询使用的总时间,单位是每秒查询次数。

3.2 单节点性能

本文针对 KSI 的查询吞吐率和多核扩展性等指标进行测试。实验平台配有 8 个 2.13 GHz 的 Intel Xeon E-7 8830 处理器,每个处理器有 8 个核,整个节点配有 1TB 内存。

图 6 给出了有序数组和散列表对 KSI 随机阻塞查询吞吐率的影响。每个进程的输入数据集是 500000 个长度为 100 的随机 DNA 序列,使用的进程数是 64 个。BS、HT1 和 HT2 分别代表了有序数组、有序数组结合散列表和有序数组结合改进直接定址的散列表。实验结果显示,HT1 和 HT2 的吞吐率比起 BS 分别提高了 7.1% 和 9.1%。

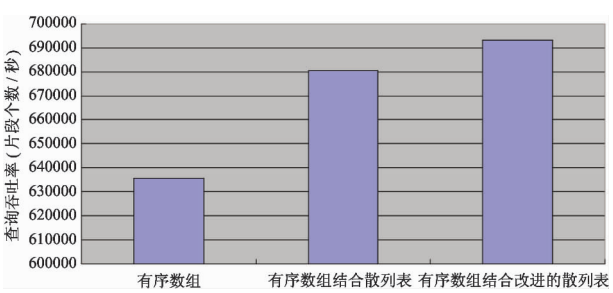

图 6 有序数组和散列表对 KSI 随机查询吞吐率的影响

图 7 是 KSI 随机查询的吞吐率曲线图,使用的是对数坐标。每个进程生成 500000 个长度为 100 的随机 DNA 片段添加到 KSI 数据库中,查询的 k-mer 长度分别为 20、40 和 100。

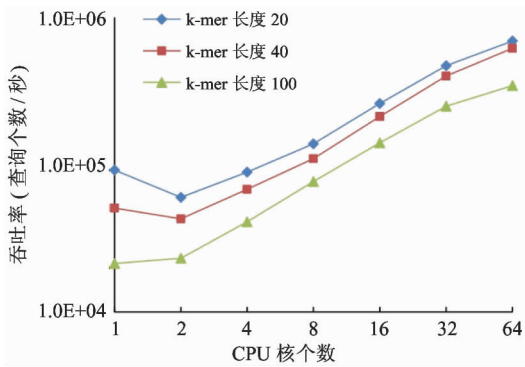

图 7 KSI 随机查询吞吐率

对于查询长度为 20 的情况,随机查询程序在单个进程时的吞吐率是每秒 9.22×10^4 次(查询所花的时间是 1.08 秒),是 2 个进程吞吐率的 1.53 倍(查询所花的时间是 3.34 秒),这是因为从单个进程查询切换到多个进程查询增加了图 5 中所示的进程间通信开销。随机查询测试程序中所有的 k-mer 都是随机生成的,所以发送到远程和本地的查询比例是 1:1。从上面的数据来看,使用 2 个进程时 3.34秒中的 $1.08/2 = 0.54$ 秒用于完成一半本地的查询,另外$3.34 - 0.54 = 2.80$秒用于完成另一半远程的查询。所以对于单节点内的情况,KSI 远程查询一次的时间是本地查询时间的 $2.80/0.54 = 5.19$ 倍。当进程数大于等于 2 时,随着进程数的增加,程序的吞吐率逐渐增加,在 8 个进程时的吞吐率为每秒 1.39×10^5,是单进程时的 1.51 倍,超过了单个进程的吞吐率,同时也是使用 2 个进程时的 2.32 倍;当增加到 64 个进程时,程序的吞吐率为每秒 6.93×10^5 次,是单进程时的 7.52 倍。

查询的 k-mer 长度为 40 和 100 时,随机查询程序的吞吐率比同样规模下查询长度为 20 时要差。为叙述方便,将查询长度为 20、40 和 100 这三种情形分别记为 q20、q40 和 q100。在单个进程时,q40 和 q100 的吞吐率相对 q20 分别下降了 44.9% 和 76.6%;在 64 进程时,q40 和 q100 的吞吐率则相对 q20 分别下降了 10.9% 和 34.5%。这是因为查询长度的增加使得 CPU 用于比对的计算量增大,从而导致查询性能下降。从图 7 同时可以看出,随着进程数的增加,序列长度对于查询吞吐率的影响在减少,这是因为通信比例的上升使得本地的查找开销相对下降。从扩展性上看,查询长度的变化并没有影响程序的扩展性,这主要是得益于每个查询请求都是理想并行的。

表 2 比较了数据库片段长度对吞吐率的影响,同样,每个进程生成 500000 个随机 DNA 片段添加到 KSI 数据库中,片段的长度分别是 100 和 200。从表中可以看出,两者吞吐率的差距小于 1%,相比图 7 可以得出,相对于 DNA 片段长度,随机查询函数的吞吐率更多受查询序列长度的影响。

表 2　DNA 片段长度对吞吐率的影响

系统规模(核数)	片段长度 100	片段长度 200
1	92156.0	92155.3
2	59946.6	60584.8
4	81419.4	81460.1
8	139211.0	140297.5
16	257253.9	255994.9
32	442064.2	442662.5
64	692566.1	694364.8

3.3　多节点扩展性

扩展性决定了 KSI 处理数据集的大小和速度,我们的设计目标是能够处理 TB 级的 DNA 序列的存储和查询。

实验平台是由 32 个计算节点组成的集群系统,每个节点配有 64G 内存和双路的 Intel Xeon E5-2670 CPU。每个 CPU 有 8 个核,每个节点最多运行 16 个线程。

我们对 KSI 在集群节点间的扩展性进行了测试,仍然使用单节点性能测试中的随机查询程序。随机查询程序生成 1.6×10^{10} 个长度为 100 的随机 DNA 片段(约 1.45TB)添加到 KSI 数据库中,查询的 k-mer 长度为 20。我们增加处理器核的规模,从 16 个核增加到 512 个核,测试结果如图 8 所示。在 16 核时,随机阻塞查询吞吐率为每秒 6.97×10^5 次,在 512 核时吞吐率为 2.24×10^7,增加了约 32 倍,达到了理想的扩展性。究其原因主要有两点,首先是因为随机阻塞查询函数是理想并行的,每个查询互相之间没有依赖性;其次,由于采用了阻塞查询的设计,所以程序的通信并没有使得网络带宽达到饱和。

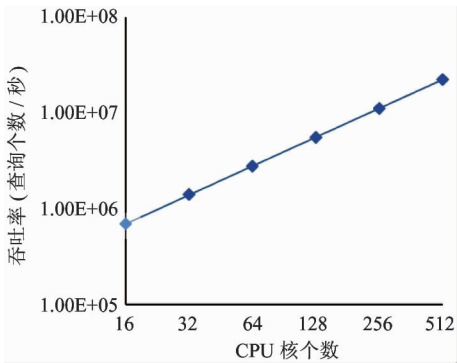

图 8　KSI 随机查询扩展性测试

4　结　论

本文设计并实现了一个基于消息传递接口(MPI)的 DNA 序列匹配软件库 KSI。KSI 为 DNA 序列提供了最基本的数据库功能:提供记录、表和数据库的数据抽象层次;提供并实现最基本的数据库存储和查询的操作接口。通过我们在单节点和分布式内存环境下的实验,证明 KSI 为 DNA 序列分析算法提供了一个高效可扩展的底层软件库。

基于 KSI 的应用程序可以不用考虑底层的数据管理而得到很好的性能以及扩展性,包括 DNA 序列的排序、划分、索引以及查询等操作,都可以通过直接调用 KSI 应用程序编程接口实现。KSI 作为一个分布式的解决方案,大大提高了 DNA 序列分析工具的性能和扩展性,对于元基因组学等需要进行生物大数据处理的研究领域有着重要意义。

参考文献

[1] Lander E S, Linton L M, Birren B, et al. Initial sequencing and analysis of the human genome. *Nature*, 2001, 409(6822): 860-921

[2] Wetterstrand K A. DNA sequencing costs: data from the NHGRI Genome Sequencing Program (GSP). http://www.genome.gov/sequencingcosts: National Human Genome Research Institute, 2014

[3] Pop M. Genome assembly reborn: recent computational challenges. *Briefings in bioinformatics*, 2009, 10(4): 354-366

[4] Boisvert S, Laviolette F, Corbeil J. Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies. *Journal of Computational Biology*, 2010, 17(11): 1519-1533

[5] MacCallum I, Przybylski D, Gnerre S, et al. ALLPATHS 2: small genomes assembled accurately and with high continuity from short paired reads. *Genome Biology*, 2009, 10(10): R103

[6] Simpson J T, Wong K, Jackman S D, et al. ABySS: a parallel assembler for short read sequence data. *Genome research*, 2009, 19(6): 1117-1123

[7] Zerbino D R, Birney E. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, 2008, 18(5): 821-829

[8] Deorowicz S, Kokot M, Grabowski S, et al. KMC 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, 2015, 31(10): 1569-1576

[9] Solomonik E, Kale L V. Highly scalable parallel sorting. In: Proceedings of the IEEE International Symposium on Parallel & Distributed Processing. IEEE, Atlanta, USA, 2010. 1 - 12

[10] Kim C, Park J, Satish N, et al. CloudRAMSort: fast and efficient large-scale distributed RAM sort on shared-nothing cluster. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, Scottsdale, USA. ACM, 2012. 841-850

KSI: a DNA sequence matching library for terabyte scale bio-data

Zhao Xiquan[*,**], Li Xu[*], Lv Huiwei[*,**], Tan Guangming[*]

([*] High Performance Computing Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([**] University of Chinese Academy of Sciences, Beijing 100049)

Abstract

It was paid attention that current mainstream softwares for DNA sequence analysis perform much repetitive work because they mostly implement a set of functions for sequence storage and query for their own use, and their design ignores the requirements of parallelism, scalability and distributed environment, while the volume of DNA data is increasing rapidly. To meet the needs for analysis of different species' DNA sequences, and adapt to DNA data's rapid increase, a DNA sequence matching library for terabyte scale bio-data, called the k-mer searching interface (KSI), was designed and implemented based on k-mer matching, the basic operation for DNA sequence processing. KSI provides a set of application programming interfaces (APIs) under distributed computing environments, and optimizes the DNA sequence matching in the biological computing field. The experimental results show that KSI is an efficient and scalable solution for big bio-data processing.

Key words: bioinformatics, k-mer searching, DNA sequence processing, application programming interface