目 录

摘要

在人类基因组计划完成之后，获得了大量的生物数据。要想从这些生物数据中获取对人类有用的信息就必需对这些数据进行分析和处理。PrefixSpan 是一个效率非常高的序列模式挖掘算法，但是原始的 PrefixSpan 算法并不适合对单条 DNA 序列中频繁子序列的挖掘，本论文中将给出一种改进后的 PrefixSpan 算法，它可以很容易地应用于单条 DNA 序列中的频繁子序列的挖掘。在 PrefixSpan 算法中随着输入序列的规模逐渐增大，算法的运行时间也越来越长，甚至达到让人无法忍受的程度，所以必需提高 PrefixSpan 算法的运行效率。可以从三个方面提高，第一改进 PrefixSpan 算法，第二提高运行的机器的配置，第三使用并行平台。提高机器的配置需要很大的投入，所以对于一般用户不切实际，如果将 PrefixSpan 算法运行在并行的平台上就可以利用廉价的 PC，很大程度上降低投入。

本论文的工作主要在上面指出的第一点和第三点上，详细工作如下：

(1) 改进 Prefixpan 算法使其更有效地适用于对单条 DNA 序列中频繁子序列的挖掘，并提高 PrefixSpan 算法在 DNA 序列中挖掘的效率。

(2) 提出一种将 BLAST 与 PrefixSpan 算法融合的方法。

(3) 提出一种将 PrefixSpan 算法并行的方法。

(4) 在 MPI 和 HADOOP 并行平台上对 PrefixSpan 算法进行并行。

(5) 比较在 MPI、HADOOP、IBM X5 3850 服务器等平台上 PrefixSpan 算法的运行效率，分析各平台的相似点和不同点。

关键字：PrefixSpan；HADOOP；MPI；改进；BLAST

1 前言

1.1 研究背景

在人类基因组计划完成之后，RNA、蛋白质以及 DNA 的数据量空前增长（马 燕和范植华，2005）。要想从这些生物数据中获取对人类有用的信息就必需对这些数据进行分析和处理。序列模式挖掘算法即从序列数据中查找多次出现的子序列，它常被用于 DNA 序列等方面的分析、疾病治疗、消费者消费行为、自然灾害、科学实验（Jian Pei et al.， 2001）。PrefixSpan 算法作为序列模式挖掘算法中的一种，它在序列模式挖掘中表现出高效性。但是原始的 PrefixSpan 算法并不适合对单条 DNA 序列中频繁子序列的挖掘，并且随着输入序列的规模逐渐增大，算法的运行时间也越来越长，甚至达到让人无法忍受的程度，所以必需提高 PrefixSpan 算法的运行效率。

当前数据的规模不断增长，数据处理难度就不断增加。使用传统的处理方法已经不再能处理当前的海量数据。但随着高性能计算技术在国内外受到高度重视，我们便可以将大数据与高性能计算相结合。现在高性能计算已经在科学研究等多个方面的应用上取得了成功。目前国内外在高性能计算机系统中使用最广泛的并行编程平台为 MPI 和 HADOOP。

1.2 主要的研究内容

PrefixSpan 算法是一个高效的序列模式挖掘算法，它应用分治的策略不断地减小投影数据库的规模。虽然它具有高效性，但是随着生物序列数量的增加，运行的时间也不断地增加，所以必需提高 Prefixpsan 算法的运行效率，有三种方法可以用来提高 PrefixSpan 算法的运行效率。第一种方法为：改进 PrefixSpan 算法，提高 Prefxispan 算法的效率，并使其更适合于 DNA 中序列模式的挖掘。第二提高运行的机器的配置，第三使用并行平台。本论文主要的研究内容在第一点和第三点上。具体工作如下：

(1) PrefixSpan 算法运行的时间主要花费在生成投影数据库上，本论文根据 DNA 序列的特性对 PrefixSpan 算法作出改进，减少 PrefixSpan 算法在投影数据生成上花费的时间，所以提高 PrefixSpan 算法在 DNA 序列中序列模式挖掘的效率。

(2) PrefixSpan 算法是通过精确匹配来查找输入序列中重复出现的子序列，所以最终得到的结果都是完全匹配的序列模式。本论文提出一种将 BLAST 算法思想融入 PrefixSpan 算法的方法，用于查找相似的序列模式。

(3) PrefixSpan 具有良好的并行性，所以本论文的另一个工作就是提出一种 PrefixSpan 算法并行的方法，将 PrefixSpan 算法在 MPI 和 Hadoop 平台上实现并行。

(4)因为 MPI 和 Hadoop 都有自己的优点与缺点,所以本文通过在 MPI 和 Hadoop 上运行 PrefixSpan 算法比较它们的相同点和不同点。

(5)随着大数据时代的到来，各种数据迅速增长。海量的数据规模已经超出了以前的 IT 架构的计算能力，所以一种新的概念被提出——云计算。云计算作是并行处理和网格计算和分布式计算的进一步发展（张建勋等， 2010），虚拟化是云计算和重要特征之一。所以本次的 MPI 和 Hadoop 实验平台搭建在一台 IBM X5 3850 服务器上。在 IBM X5 3850 上使用虚拟化平台 Vmware vSphere ，并在这台机器上搭建 10 台虚拟机作为集群。

1.3 章节安排

本论文章节安排：

第一部分介绍研究背景和主要研究内容

第二部分对 PrefixSpan 作一个详细介绍

第三部分介绍对 PrefixSpan 算法的改进

第四部分介绍将改进的 PrefixSpan 算法与 BLAST 融合的方法与实现

第五部分介绍改进的 PrefixSpan 算法在 MPI 和 Hadoop 上的实现方法

第六部分对实验结果进行分析

第七部分总结研究的成果和讨论未来的工作

2 PrefixSpan 算法介绍

2.1 PrefixSpan 算法的由来

PrefixSpan 算法是一个序列模式挖掘算法。序列模式挖掘最初是被 Ramakrishnan Srikant 和 Rakesh Agrawal 提出，主要用来分析超市的购物数据（陈卓等，2008）。序列模式挖掘就是：给定一组序列，每个序列由若干个项集组成，每个项集由若干个项，用户指定一个最小支持度，序列模式挖掘就是找出所有频繁出现的子序列，这个子序列在整个序列出现的次数不能小于支持度。基本的序列模式挖掘算法有 Apriori 算法、类 Aprior 算法(AprioriAll、AprioriSome 等)、GSP 算法、PrefixSpan 算法、SPADE 算法、MEMISP 算法、SPIRIT 算法。PrefixSpan 是 Pei 在 2001 年提出，它的性能优于 AprioriAll 算法和 GSP 算法（Jian Pei et al.， 2001）。

2.2 PrefixSpan 中的基本概念

定义 1：前缀，给定两个序列 a=<a1,a2,a3,a4……an>, b=<b1, b2, b3, b4,……, bm >。b 可以称为 a 的前缀仅当：①ai=bi（i<m） ②bm⊆ am ③bm 中所有的项(am-bm)中所有项的前面这几个条件都成立（Jian Pei et al.， 2001）。

比如：<a1>,<a1,a2>都为<a1,a2,a3,a4 …… an>的前缀

定义 2：投影，给定两序列 a=<a1,a2,a3,a4 …… an>, b=<b1,b2,b3,b4,……,bm>，其中 b 为 a 的子序列。要 P 称为 b 关于 a 的投影当且仅当：①b 是 p 的前缀，②不存在 p 的超级序列（p⊆ Q 但是 P≠Q）Q，使 Q 是 a 的子序列并且 b 也为 Q 的前缀（Jian Pei et al.， 2001）。

比如：< a2,a3,a4 …… an>为<a2>在 a 上的投影, < a3,a4 …… an>为<a3>上的投影。

定义 3：后缀，P=< a1,a2,a3,a4 …… an>为 a 中以 b=< b1,b2,b3,b4,……,bm>为前缀的投影，r=<bm',a_{m+1},……an>其中 bm'=am-bm（Jian Pei et al.， 2001）。

比如：< a2,a3,a4 …… an>为<a2>在 a 上的投影，<a2>的后缀为< a3,a4 …… an>

定义 4：投影数据库，设 a 为序列数据库 s 中的一个序列，a 在 s 中的投影数据库为 s 为所有以 a 为前缀的后缀之集（武珍珍， 2010）。

定义 4：项集，一个项的非空集合可以表示为(x1,x2,x3……xn)，其中 xi 为一个项（Jian Pei et al.， 2001）。

定义 5：序列，项集的集合，可以表示成<a1,a2,a3,……,an>其中 ai 为一个项集（Jian Pei et al.， 2001）。

定义 6：子序列，设 a=<a1,a2,a3,...,an>,b=<b1,b2,b3,...,bm>若存在整数 k=e1<e2 < ... em<=n 使得序列 b1⊆ a_{e1},b2⊆ a_{e2},....,bm⊆ a_{em} ，称 b 是 a 的子序列，又称 a 包含 b,词为 b⊆a（Jian Pei et al.， 2001）。

定义 7：序列长度，项的个数（Jian Pei et al.， 2001）。

定义 8：支持数，序列数据库中包含序列 S 的个数据（Jian Pei et al.， 2001）。

定义 9：支持度，预先设定的一个阈值（Jian Pei et al.， 2001）。

定义 10：序列模式，如果存在一个序列 a，它在序列数据库中出现的次数不小于支持度，则称它为一个序列模式（Jian Pei et al.， 2001）。

2.3 PrefixSpan 的执行过程

2.3.1 执行步骤

输入：Sdatabase（序列数据库）用户设定的最小支持度 n

输出：满足条件的序列模式

使用方法：调用 PrefixSpan(Sdatabase,a,len)，其中 PrefixSpan(Sdatabase,a,len)为 PrefixSpan 运行函数，用于处理 PrefixSpan 算法的执行，Sdatabase 为用户输入的序列，a 之前计算出的一个序列模式，初始时 a 为空，len 为 a 的长度。

调用子程序：PrefixSpan($Sdatabase|_a$,a,L)，其中 $Sdatabase|_a$ 为以 a 为前缀的投影

数据库，a 为一个序列模式，L 为序列模式 a 的长度。

Prefixsapn(Sdatabase|$_a$,a,L)执行过程：

遍历一遍 Sdatabase|$_a$，找到一组频繁项 b，如果 b 可以添加到 a 的最后一个项集之中则加入此项集之中或者可以作为一个新的项集添加到 a 的末尾处则作为一个新的项集加入 a 的末尾处，这样就组成了一个新的序列模式。

(1) 对于每一组频繁项 b，把它添加到 a 的未尾处组成一个新的序列模式 a_1 并且输出这个序列模式。

(2) 对于新的序列模式 a_1，再次调用 PrefixSpan 函数，Prefixsapn(Sdatabase|$_{a1}$,a_1,L)

(3) 运行以上步骤直到不存在满足最小支持度 n 的序列模式出现。

2.3.2 一个 PrefixSpan 执行实例

输入：设置支持度为 3，序列数据库为 Sdatabase (表 1)。

表 1. 序列数据库 Sdatabase

Table1. Sequence Database Sdatabase

编号	序列
1	<agctg>
2	<agcgt>
3	<gcgat>
4	<cacag>

第一步：遍历序列数据库 Sdatabase，找出长度所有长度为 1 的序列模式。结果为<a：4>，<g：4>，<c：4>，<t：3>

第二步：生成在第一步中所得的序列模式的投影数据库，如表 2 所示。

表 2. 长度为 1 的序列模式的投影数据库

Table2. Projected Databases

<频繁序列 >	投影数据库	支持度
<a>	<gctg><gcgt><t><cag>	4
<g>	<ctg>< cgt>< cgat>	4
<c>	<tg><gt><gat><acag>	4
<t>	<g>	3

第三步：对<a>的投影数据库调用 PrefixSpan(Sdatabase|$_a$，<a>，1)，在<a>的投影数据库中找出的频繁模式为<c：3><g：3><t：3>，生成它们的投影数据库如表 3。

表 3. <a>的序列模式的投影数据库

Table3. the Projected Databases of <a>

<频繁序列 >	投影数据库	支持度
<c>	<tg><gt><ag>	3
<g >	<ctg><cgt>	3
<t>	<g>	3

第四步：对<ac>生成的投影数据库调用 PrefixSpan(Sdatabase|$_{<ac>}$，<ac>，2)，在投影数据库中找出频繁模式为<g：3>，生成它们的投影数据库如表 4。

表 4. <ac>的序列模式的投影数据库

Table4. the Projected Databases of <ac>

<频繁序列 >	投影数据库	支持度
<g>	<t>	3

第五部：因为<acg>的投影数据库中的序列条数小于支持度，所以递归调用的此分支运行结束得到结果<acg>，如表 5 所示。

表 5. 此递归运行的结果

Table5. the result of this recursion

<频繁序列 >	支持度
<acg>	3

第六步：利用同样的递归方法可以找出其它频繁序列的序列模式。

3 PrefixSpan 算法的改进

3.1 原始 PrefixSpan 算法在 DNA 序列模式挖掘应用中的缺陷

(1) 原始的 PrefixSpan 算法是在一个包含多个序列的序列数据库中查找重复出现的序列模式，而且在同一个序列中重复出现的子序列在此序列出现的次数也计算为一。此并不适用在单条序列中查找重复出现的子串，并且人类 DNA 一共有 24 条，每一条都有超过千万的碱基对，我们经常要在单条 DNA 中查找重复出现的子序列。所以原始的 PrefixSpan 算法并不适合查找单条 DNA 序列中的序列模式。常采用的方法就是将一条长度为 L 的 DNA 序列分成长度为 m 的子序列集合，再把这 L/m 条子序列作为输入的序列数据库，查找出现的序列模式，但是这必定会导致部分信息丢失。

(2) PrefixSpan 是一个高效的序列模式挖掘算法，该算法执行时主要的时间花费在投影数据库的产生上，这也成为了这个算法的瓶颈。

(3) PrefixSpan 算法在执行中可能会出现对一些子串进行重复挖掘，比如对一个序列进行挖掘，已经得到了结果<aac>它的支持数为 3，<ac>也是所给的序列中的一个子序列，它出现的次数也为 3 所以它完全包含于<aac>之，但因为算法是递归调用，就可能会出现得到结果<aac>之后再次挖掘<ac>，这样做的就是一个无用功。如果能改进此缺点就可以大大地提高算法的性能。

(4) PrefixSpan 算法是通过精确匹配查找出序列模式，但有可能存一条生物序列因为变异等原因造成某个位置上的碱基发生了变异，所以在精确匹配中这条序列就有可能被忽略。

3.2 对 PrefixSpan 算法的改进

3.2.1 将 PrefixSpan 算法应用于单条 DNA 序列中的序列模式挖掘

为了使用 PrefixSpan 算法在挖掘单条 DNA 序列中的序列模式时不丢失信息，必需更改 PrefixSpan 的执行流程，使其适用于单条 DNA 序列。本文提供的方法如下。

将输入的一条 DNA 序列作为输入的序列数据库。遍历整个序列数据库记录每个长度为 1 的项出现的次数（支持数），保留支持数不小于支持度的项，并根据 PrefixSpan 原始算法的步骤生成这些项的投影数据库。根据生成的投影数据库不断扩展频繁序列的长度直到支持数小于支持度，当两个相同但出现位置不同的子序列有相交时则他们的只算出现一次（也就是支持数记为 1）。

实例如下：

输入：一个序列数据库 S=<ctaaaaacaacgctaacc>，设置支持度为 3。

(1) 遍历整个序列，取出所有长度为 1 且出现次数大于等于支持度的项并生成它们的投影数据库，结果如表 6。

表 6. 长度为 1 的项的投影数据库

Table6. the Projected Databases of items which length is 1

长度为 1 的项	投影数据库	支持数
a	<a:3><a:4><a:5><a:6><a:7><a:9><a:10><a:15><a:16>	9
c	<c:1><c:8><c:11><c:13><c:17><c:18>	5

注：其中<char ： add>中 char 表示项的值，add 表示 char 在序列数据库的位置，比如 S 中第一个 c 出现的位置为 1 所以它表示为<c:1>。

(2) 对 a 项扩展，结果如表 7。

(3) 对步骤(2)得到的结果继续扩展。取出<aa>，对它继续扩展，结果如表 8。

PrefixSpan 算法使用的时候可以使用递归的方法调用，不断向下扩展直到不再满足支持度。上面的结果得到以 a 开始扩展的一个分支的结果<aac>，出现次数为 3。

通过上面的方法使用 PrefixSpan 算法将不会发生信息丢失的情况，它能将一个单序列中的所有频繁模式挖掘出来。

表 7. a 扩展后的投影数据库

Table7. the Projected Databases of <a>

<频繁序列 >	投影数据库	支持数
<aa>	<aa:4><aa:5><aa:6><aa:7><aa:10><aa:16>	4
<ac>	<c:8><c:11><c:17>	3

注：由于<aa:4><aa:5>有重叠部分所以出现次数记录为 1 但同时保留两个项集出现的位置，同理<aa:6><aa:7>出现次数记录为 1，所以<aa>的支持度记为 4。

表 8. aa 扩展后的投影数据库

Table8. the Projected Databases of <aa>

<频繁序列 >	投影数据库	支持数
<aac>	<aac:8><aac:11><aac:17>	3

3.2.2 对投影数据生成改进

本文根据DNA序列的特点对PrefixSpan算法生成投影数据库的方法进行了改进，这很大程度上提高了 PrefixSpan 算法的性能，当然这改进不仅适用于对 DNA 序列的频繁子序列的挖掘也适用于普通的只含项的序列的频繁子序列的挖掘。

在一个 DNA 序列中，单个类型的碱基出现的次数肯定很多，比如一条人类的 dnaY，它拥有 25653566 个碱基对，并且 DNA 序列中只有四种碱基，如果按平均值来算，那碱基 a 出现的次数达到 7667625 次。如果用户要查看长度至少为 100 支持度为 100 的频繁序列，初始时如果以 PrefixSpan 算法的原始流程执行，那对于碱基 a 它的投影数据库将拥有 7667625 条，这不仅占用非常大的内存而且运行速度非常慢，对于每次增加 1 个碱基对的长度，那 aa 出现的次数也达到了 1823333，虽然数据规模下降了三倍多，但是这还是一个非常大的数子，aaa 出现的次数为 637764。我们可以从上面的数据规模发现一个规律——随着子序列的长度增加，它出现的次数也逐渐降低。在 DNA 序列模式挖掘中用户一般会指定的个挖掘的最小长度（其它数据的模式挖掘也可能会指定最小长度），也就是说虽然 a 出现了七百多万次，但是单个 a 并不能当作一个序列模式输出。如果用户指定输出的序列模式最小长度为 MIN_LEN，我们是否可以设定一个长度 NODE_LEN（1<=NODE_LEN<=MIN_LEN），把 NODE_LEN 当作一个序列块，长度为 NODE_LEN 的序列块出现的次数肯定比长度为 1 的单个碱基出现的次数少了很多，这也大大地减小了投影数据库的规模。当然如果直接把一个长度为 M 的序列划分为 M/NODE_LEN 份，再把每一份当作一个

整体这必定会导致大量信息的丢失，这样做是不明智的，本文采用的方法的是从序列的头部开始向后遍历，将遍历的到的项和后 NODE_LEN-1 位当作一个块，并且将此块出现的次数加 1，当块出现的次数等于支持度 N 时就将此块保存。在向前遍历时每次向前推进一位。

本论文不仅在初始分块的时候将输入的序列数据库划分为多个块，并且在递归调用 PrefixSpan()时如果当前的子序列长度还没有达到指定的最小序列长度，就可以每次增加 NODE_LEN 的长度，因为子序列的长度没有达到要求的最小值并不会输出。

通过上面提出的对 PrefixSpan 算法的改进将对算法的性能大大地提升。假定给一个输入序列<ctaaaaac>设定最小长度为 MIN_LEN，NODE_LEN 为 4 并且 MIN_LEN >= NODE_LEN，对它的划分如图 1 所示。

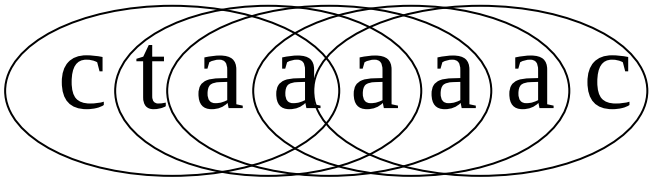

图 1 分块方法

Fig.1 Partitioning method

每个圆圈都表示一个分块。分块的流程如图 2 所示。下面举一个实例来详细说明分块的方法：

输入：一个序列数据库 S=<ctaaaaactaa>，设置支持度为 2，最小长度为 4，NODE_LEN 为 2。

(1) 将 S 划分成长度为 2 的块。

表 9. S 划分后的块

Table9. The Block After Dividing S

块	出现位置
ct	2
ta	3
aa	4
aa	5
aa	6
aa	7
ac	8
ct	9
ta	10
aa	11

(2) 取出其中支持数不小于支持度的串，生成投影数据库如表 10。

(3) 对<ct>继续生成投影数据库，因为<ct>的长度为 2 而设定的最小长度为 4，4 与 2 的差为 2，所以每次向前推进 2 位，如表 11 所示。

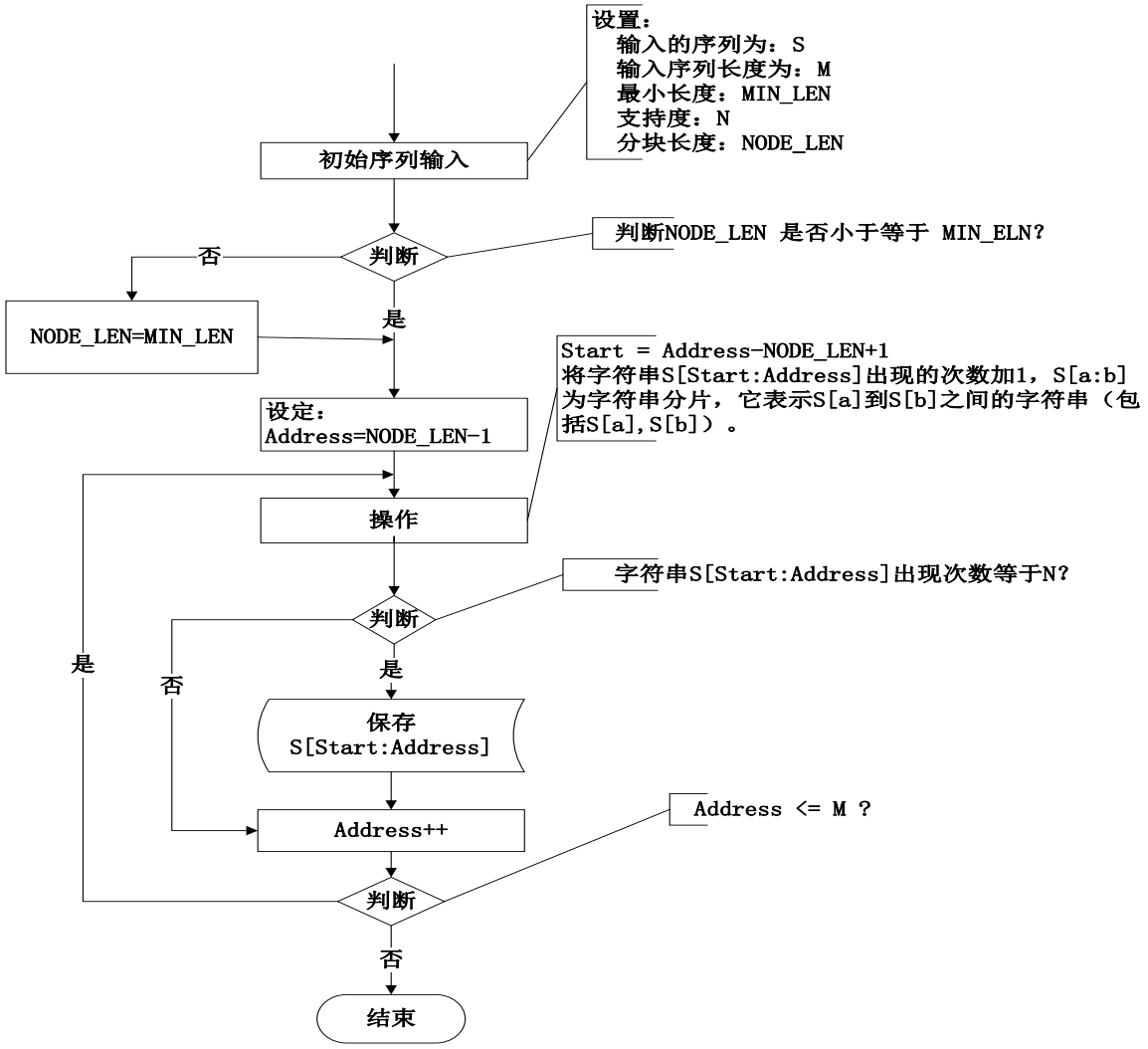

图 2 初始化的执行流程

Fig.2 The Execution Flow Of init

表 10. 投影数据库

Table 10. the Projected Databases

<频繁序列 >	投影数据库	支持数
<ct>	<ct:2><ct:9>	2
<ta>	<ta:3><ta:10>	2
<aa>	<aa:4><aa:5><aa:6><aa:7><aa:11>	3

表 11. 投影数据库

Table11. the Projected Databases

<频繁序列 >	投影数据库	支持数
<ctaa>	<ctaa:4><ctaa:11>	2

(4) 通过上面的划分后，因为设置的最小长度为4，所以再每次加2个长度的项，如果设定最小长度为6，可以每次增加3个长度的项。最终得到一个结果<ctaa>支持度为2。

从上面的结果可以看出，通过划分片段可以大大地姜减小投影数据的规模，而在 PrefixSpan 算法执行过程中，时间主要花费在投影数据库的产生中。所以此改进可以提高该算法的效率，本文将在实验数据分析这一节用实验数据证明此算法的可行性和高效性。

3.2.3 将序列项转变为 4 进制

在 DNA 序列中有一个重要的特性，DNA 的碱基只有四种。我们可以很好地利用此性质来提高 PrefixSpan 算法的效率，本文提供的方法就是将碱基转变为四进制，本文采用转换表来转换，转换表如表 12 所示。

表 12. 碱基转换表

Table12. translation table

碱基名	转换后的数字
a	0
c	1
g	2
t	3

比如将表 9 的序列转变为四进制后，如果如表 13 所示。

表 13. 四进制表

Table13. quanternary table

块	出现位置
28	3
48	4
0	5
0	6
0	7
1	8
4	9
16	10
1	11
6	12

对于给定的一个长度为 L 的序列 S，将其划分为长度为 NODE_LEN 的块，并将块转换为四进制，这个步骤的时间复杂度为 O(n)，也就是只需要对这个序列遍历一遍，流程图如图 3 所示。

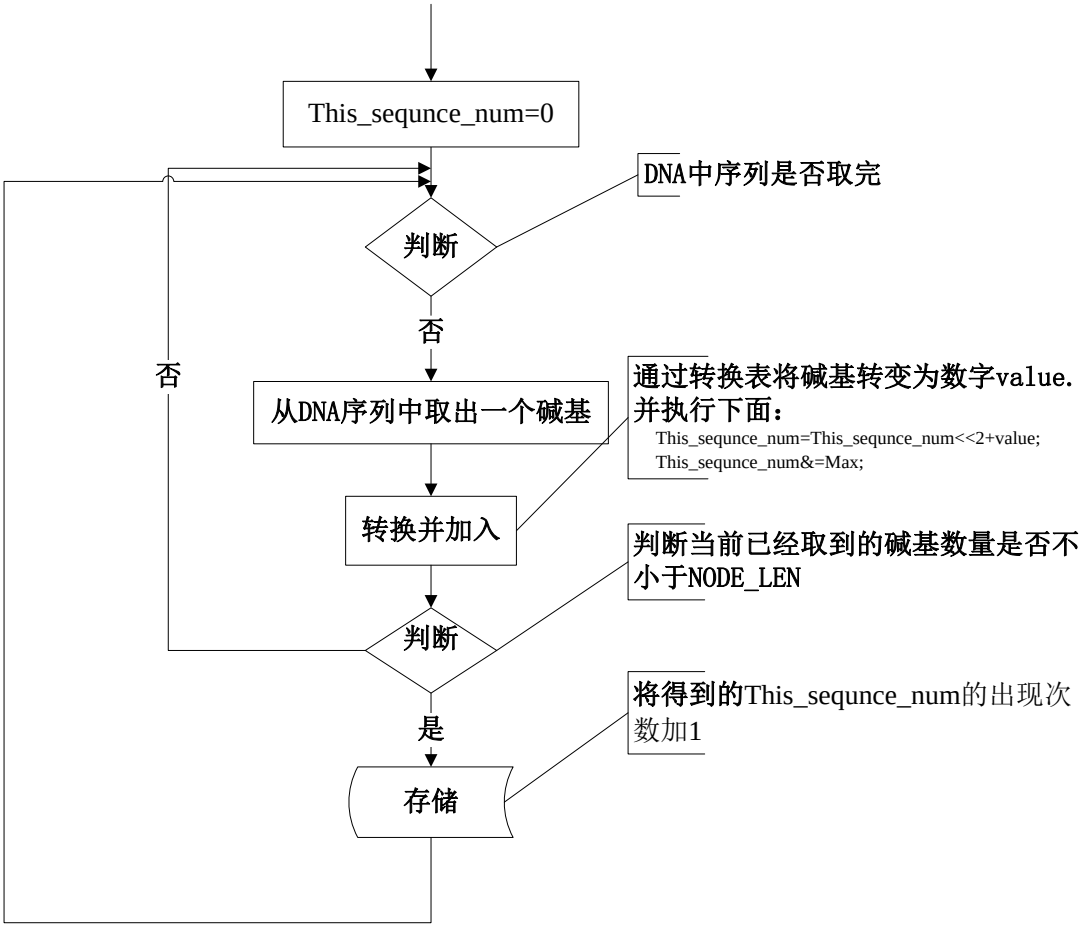

图 3　init_node 的执行流程

Fig.3　The Execution Flow Of init_node

上面的流程也为本文程序中 init_node 函数的实现，通过将碱基转换为四进制，生成一个 hash 表，方便子串的搜索和比较，应用四进制形式比较两个字符串是否相等时时间复杂度为 O(1)，如果不将序列转换为数字形式，两字符串比较的时间复杂度至少为 O(n)。通过将字符串转换为四进制的形式还可以方便于 hash 定位，在 PrefixSpan 执行时要统计一个子序列出现的次数，所以就要将子序列保存于数组或者结构体中，当要查找一个子串保存位置时如果使用字符串的比较，则比较时间复杂度平均为 O(n)，如果使用数字的表示形式时间复杂度为 O(1)，所以这一改进将对 PrefixSpan 算法的性能有一定程度的提升。在实验数据分析的章节，本文将用实验数据证明此方法的可行性和高效性。

3.3 算法实验数据分析

3.3.1 实验数据与环境

实验数据：本实验使用人数的 dnaY 作为输入序列，本文使用的 dnaY 序列一共有 25653566 个碱基对。

实验目的：找出这条 dnaY 上长度大于等于 MIN_LEN 并且支持数大于等于 N 的子序列。即设定最小长度为 MIN_LEN 支持度为 N。

实验环境：Centos　CPU*2　8G 内存

3.3.2 PrefixSpan 算法在 DNA 序列中应用的有效

本文使用 3.2.1 介绍的方法，将 PrefixSpan 算法应用于 DNA 序列的序列模式挖掘，设定支持度 10，最小长度 100。得到的部分结果如下：

accattgtggaagtcagtgtggcgattcctcagggatctagaactagaaataccatttgacccagccatcccattactgggtatatacccaaaggactat : 10

aaaaaaaccctacttctttctgtcatttactgtgaggcattactgaatctgggtgtattcatgtatgctgctacctgtatgttttcaaacaataagaatt : 11

这是实验得到的 7468（其中包含重复的结果）条结果中的 2 条。上面的结果表示形式可以表示为<string ： num>，其中 string 表示得到的频繁序列，num 表示在输入的 dnaY 中出现的次数，上面的两个子序列出现的次数分别为 10 和 11。

实验结果的正确性：本文将 dnaY 通过编辑器 UltraEdit 打开，通过 UltraEdit 统计得到结果中的子序列在 dnaY 中出现的次数，所得结果与本文实验结果相同（图 4），这可以证明实验结果正确。

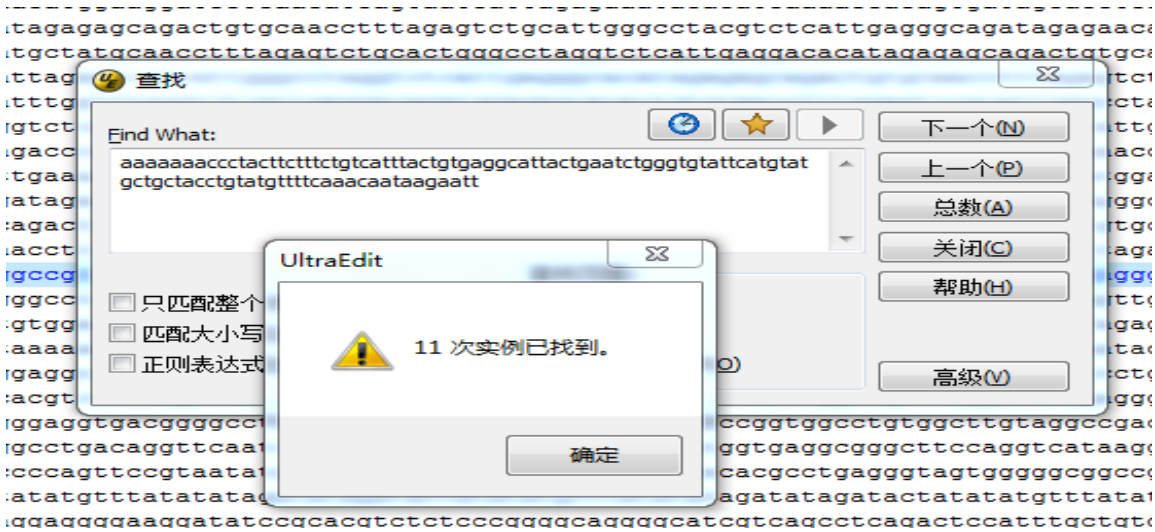

图 4　利用 UltraEdit 的统计结果

Fig.4　the results of UltraEdit

3.3.3 PrefixSpan 算法投影数据库改进的高效性

当子序列长度小于设定的最小长度时可以将长度为 NODE_LEN 的序列当作一个块，将一个长度为 LEN 的块划分为 LEN-NODE_LEN+1 个块，并且如果当前子序列长度小于设定的最小长度时生成投影数据库时每次向前推进 NODE_LEN。

本文设定支持度为 10，最小长度为 100，分块长度为 NODE_LEN，在 dnaY 中挖掘频繁序列，执行 PrefixSpan 算法，得到的结果和没有使用分块时结果相同，所以这可以证明分块后算法的正确性，由于当子序列长度大于设定的最小长度后每次向前进的长度为 1，所以这样可以保证算法不会丢失信息，但是这也有一个缺点：算法的性能与设定的最小长度相关。本文将 NODE_LEN 的长度设定为 1、3、5、7、8、9、10、11、12、13 得到不同的运行时间，如图 5 所示。

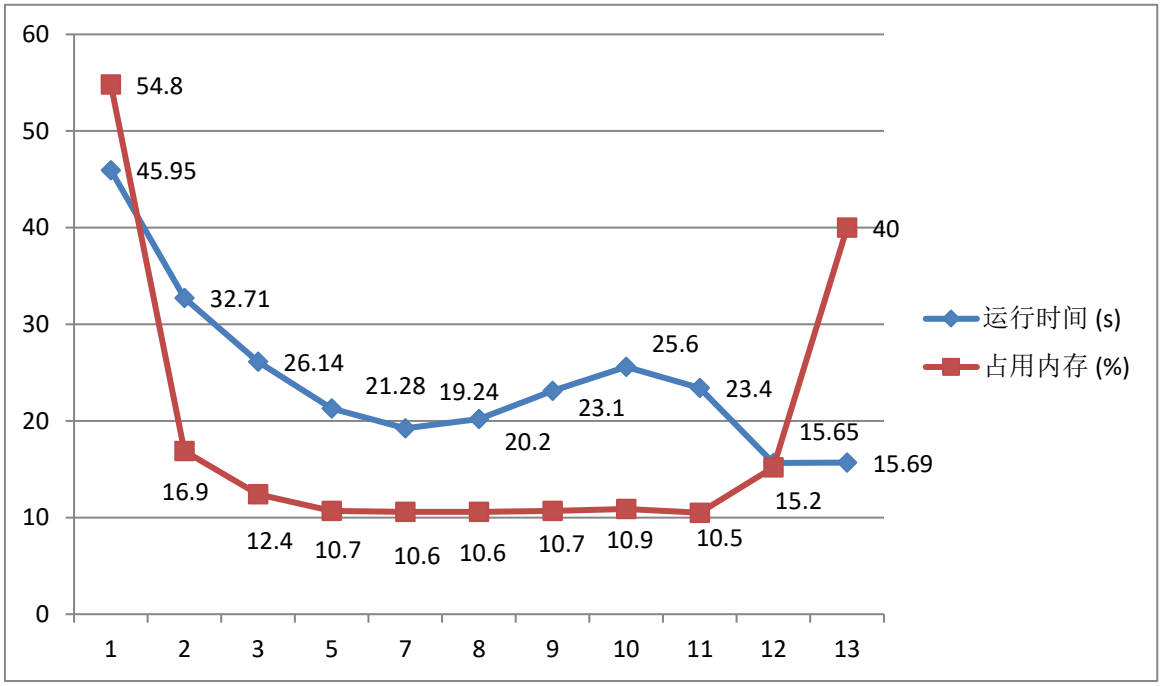

图 5　总体运行时间

Fig.5　Overall Running Time

从图 5 和图 6 可以明确地得到当产生投影数据库时向前推进的长度 NODE_LEN 变化时 PrefixSpan 算法的执行时间也会产生变化。从表中可以得到当 NODE_LEN 设置为 1 时 PrefixSpan 算法的执行效率最低。从图 6 可以看出当 NODE_LEN 在 1 到 7 时随着 NODE_LEN 的减增大执行 PrefixSpan 算法时投影数据花费的时间与 NODE_LEN 的大小成反比，当 NODE_LEN 的大小在 7 到 10 时生成投影数据库所花费的时间变化较小，但当 NODE_LEN 大于 10 时，随着 NODE_LEN 的增大，生成投影数据库所花费的时间也变小。

从图 5 我们可以得到当 NODE_LEN 设置为 1 时和当 NODE_LEN 设置为 13 时，

PrefixSpan 所占用内存突然变大，这是因为当 NODE_LEN 设置为 1 时生成的投影数据中将含有非常多的子序列，所以占用内存会很多，当 NODE_LEN 设置为 13 或者更大时虽然投影数据库的规模会变小，但是算法初始执行时会进行分块这将可能产生 4**NODE_LEN（表示 4 的 NODE_LEN 次方）的一个 hash 表（此 hash 表本文已前面介绍过），当 NODE_LEN 设置为 13 时 hash 表长度为 67108864，在本文程序中 hash 表的一个节点大小为 56 字节，经过计算 hash 表占用的内存为 3584MB，所以内存占用过大，而当将 NODE_LEN 设置为 10 时，占用的内存只有 56MB，所以本文的程序中 NODE_LEN 取的值为 10。

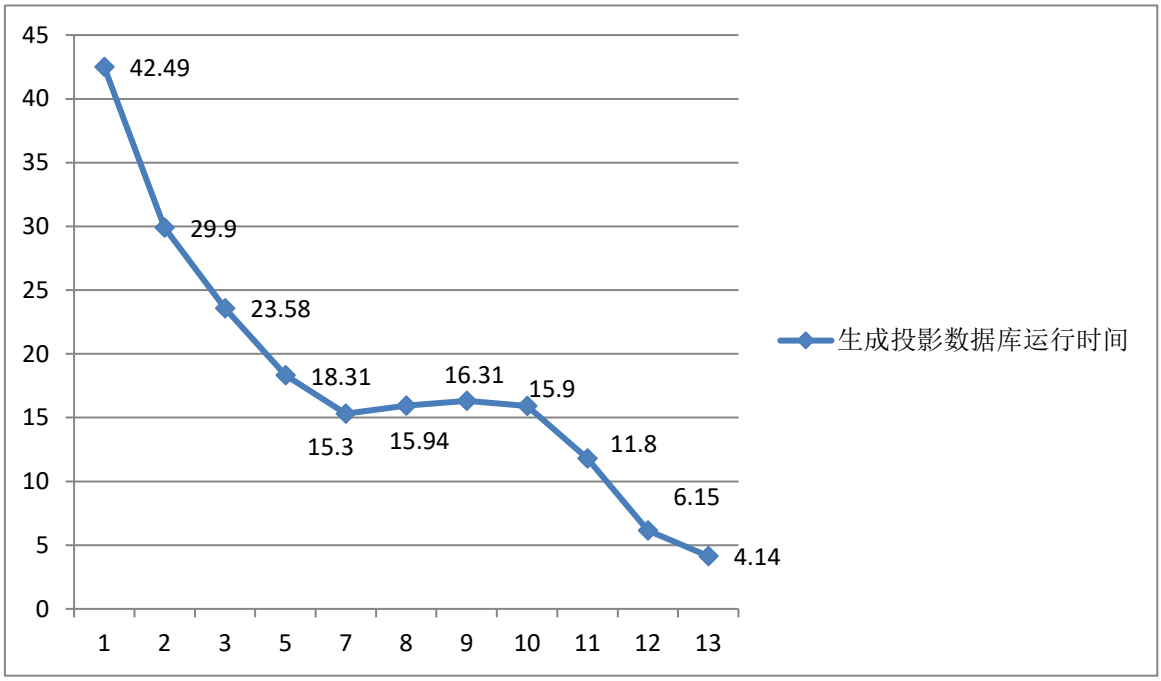

图 6　生成投影数据库运行时间

Fig.6　Create Projected Databases Running Time

经过图 5 和图 6 的对比我们可以发现当对 Prefixsapan 进行分块后，PrefixSpan 算法整体的性能提高了 2 倍左右，当 NODE_LEN 设置为 13 时生成投影数据库所花费的时间降为原来的十分之一，并且内存占用量也降低，综上所述可得：通过对 PrefixSpan 分块后算法的效率明显提高。

3.3.4 对碱基采用四进制编码后算法性能显著提升

对碱基采用四进制编码后，可以提高 PrefixSpan 算法中字符串比较速度和搜索速度。为了证明采用四进制后提高了算法的高效性，本文采用了两搜索和比较的方法，第一种方法为采用四叉树搜索，第二种方法采用将碱基转换为四进制形式并使用 hash 的方法进行搜索。这里用的实验环境和数据和上一节使用的相同，结果如图 7 与图 8 所示。

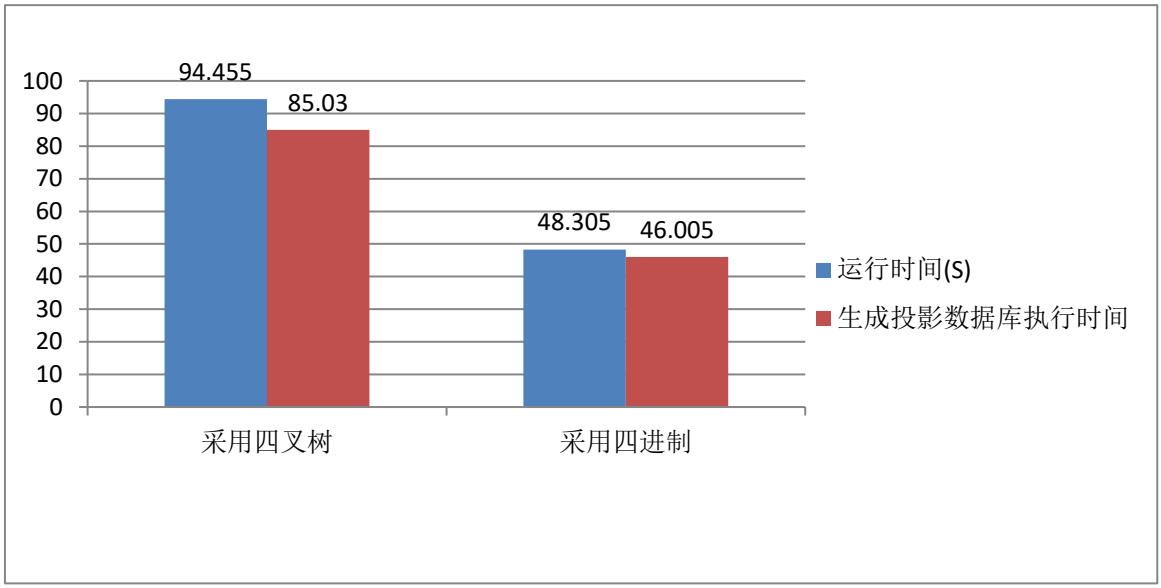

图 7　总体运行时间

Fig.7　Overall Running Time

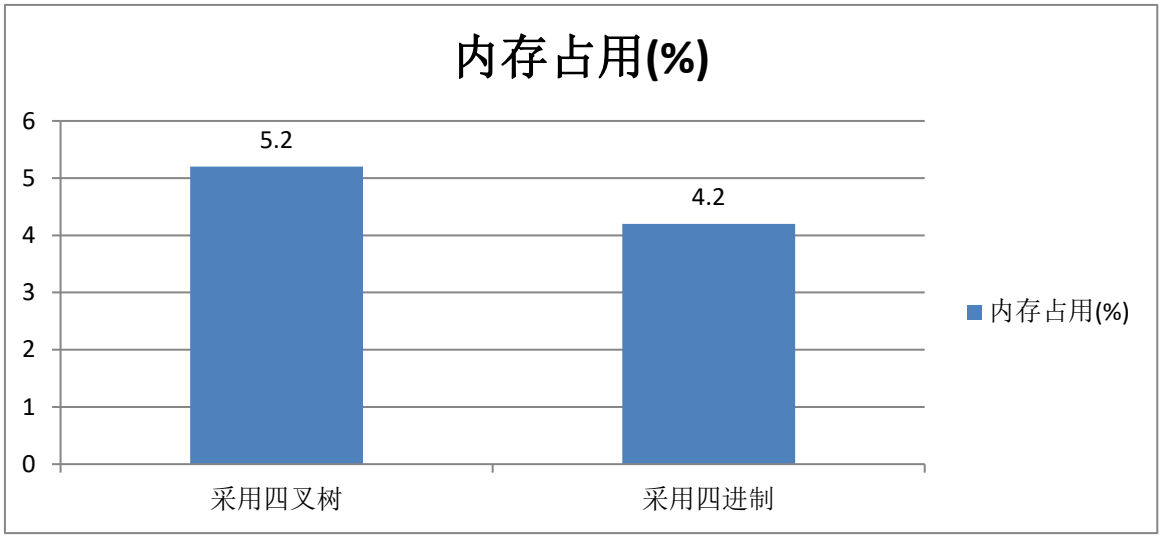

图 8　运行内存占用

Fig.8　Memory usage

这两个对比程序都是对 dnaY 中序列模式的挖掘，本文设定的最小长度为 100，支持度为 10，NODE_LEN 长度为 10。对比程除了搜索方法和字符串比较方法不相同，其它都相同，所以两程序满足对比的要求，从图 7 可以看出采用四进进制形式进行搜索和比较的程序运行时间为采用四叉树方法的 1/2，并且当采用四进制形式时程序所占用的内存也变小。综上所述：当采用将碱基转换为四进制形式，再利用 hash 的方法，能够大大提高 PrefixSpan 算法的性能，但此程序采用四进制是因为 dna 序列的特性（只有四种不同的碱基），所以四进制的方法并不能应用于各类序列的挖掘分析工作中，只有当序列中项的种类数量较小才能使用。

4 将在 PrefixSpan 算法与 BLAST 算法结合

4.1 结合目的

图 9 给出的一个用改进后的 PrefixSpan 算法挖掘 dnaY 的频繁模式的一条结果，从结果中可看出 PrefixSpan 算法可以很好地挖掘单条 DNA 序列上的频繁模式。此条结果表示 aaaaaaaccctacttctttctgtcatttactgtgaggcattactgaatctgggtgtattcatgtatgctgctacct gtatgttttcaaacaataagaatt 在 dnaY 中出现了 11 次，PrefixSpan 算法是通过精确匹配查找输入序列中出现的频繁模式，但是在生物序列中可能存在多条序列本属于同源，但是由于变异导致这些序列中存在个别碱基不相同，这种序列不能通过本论文前面使用的方法找出。所以本文提出一种将用于相似性比较的 BLAST 算法的思想引入 PrefixSpan 算法的方法。通过将 BLAST 算法思想与 PrefixSpan 算法思想结合就可以不忽略那些存在变异的序列。

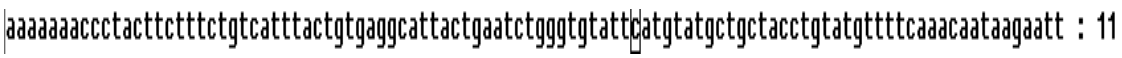

图 9　一条实验结果

Fig.9　one experimental results

BLAST 算法采用打分矩阵，通过计算最终得分来评估两条序列的相似程度。但是 BLAST 是两两序列之间相互比较相似性，所以如果只用 BLAST 算法来查找一条 DNA 序列中的频繁模式效率将非常低下。

4.2 实验的假设

首先提出本文在实现时所作出的假设：

(1) 多条长度为 LEN 的序列如果它们之间相似则假设它们之间存在百分之 M 的区域完全相同（M 可以根据比较的相似程度来设定）。

(2) 在一定的相似比例下，如果序列 a,b,c 中 a 与 b 相似且 a 与 c 相似则 b 与 c 相似。

上面的假设可能对于某些序列不成立，这就可能造成信息的丢失，但是这此序列所占比例非常少，所以本实验中将忽略丢失信息。

4.3 结合的方法

输入一条 DNA 序列要求挖掘最小长度为 MIN_LEN，支持为 MIN_NUM 的频繁序列。我们假设这条 DNA 序列中存在一组不小于 MIN_NUM 条的子序列它们之间相似并且每条长度大于等于 MIN_LEN，这一组子序列就是我们要挖掘的序列模式之一。根据假设 1 我们认为它们之间存在长度为 MIN_LEN*M%的完全匹配部分，我

们可以使用本论文前面所提出的改进后的 PrefixSpan 算法查找长度为 MIN_LEN*M% 并且满足支持度的完全匹配的子序列，然后再在这些完全匹配的序列的基础上利用 BLAST 算法的相似度比较的思想向左右两个方向扩展。向左右扩展时本文还是采用 PrefixSpan 生成投影数据库减小数据规模的方法，具体扩展步骤如下：

向右扩展

开始：假设本投影数据库中共有 n 条子序列，第一条的编号为 f，i 的值为 1，min_num 值为 1（表示相似条数）。

(1) 取出编号为 f 的子序列前 NODE_LEN 个碱基组成 e1。
(2) 如果所有子序列已比较完所有序列转向 7，否则取出第 f+i 条序列的前 NODE_LEN 个碱基组成 ei。
(3) 比较 e1 与 ei 的相似性（当前分数 score 与最高分数 hsp 不仅与局部相关还与整体相关）。如果相似 i 值加 1 出现次数 min_num 值加 1，进入步骤 1。
(4) 当移动位数小于设定的最大次数时将 e1 向右移一位转向 3
(5) 当移动位数小于设定的最大次数时将 ei 向右移一位转向 3，当已经到达尾部转向 6。
(6) i 值加 1 并转向 1
(7) 如果 min_num 不小于支持度则将扩展中与子序列 f 相似的串取出，生成它们的投影数据库，对这些相似的串的投影数据库重复上面的步骤直到不再有满足的序列为止，否则将投影数据库中的序列 f 和与序列 f 相似的串去除，将剩下的子序列重新编号，第一条子序列编号为 f。返回开始。

向左扩展与向右扩展相似。上面的算法中通过引入 PrefixSpan 算法生成投影数据库的思想可以减少序列之间的比较次数，提高算法的效率。

4.4 一个实例

设定：最小长度 min_len 10，最大移动次数 2，支持度 2，NODE_LEN 5。完全匹配比例 50%（这里为了方便举例这里设定完全匹配率为 50%，但在实际使用中应该取较小的值），所以完全匹配长度为 5。

输入序列：tatgcaggttccgacgt tctgcagcttccactag

(1) 通过 PrefixSpan 算法找出长度取出长度为 5 的完全匹配如下。

表 14. 完全匹配

Table14. complete matching

子串	投影数据库
<tgcag>	< tgcag:7> < tgcag:19>

(2) 向右扩展：

a 将< tgcag:7>编号为 f，< tgcag:19>编号为 f+1。

b 取 e1 为 gttcc，e2 为 cttcc

c 比较 e1 与 e2 的相似性，根据 BLAST 规则 e1 与 e2 相似（此处不介绍 BLAST 算法，在文献 5 中有详细介绍），将出现次数 min_num 加 1。进入步骤 1。

d 因为投影数据库中只有两条子串，所以在步骤 2 中转入步骤 7。Min_num 的值为 2，所以为 e1 与 e2 生成投影数据库，如下表。

表 15. 投影数据库

Table15. the Projected Databases

子串	投影数据库
<gttcc><cttcc>	< gttcc:12> < cttcc:24>

e 进入步骤 1 和 2，取 e1 为<gacgt>，e2 为<actag>。

f 步骤 3 比较 e1 与 e2 的相似性，可得 e1 与 e2 不相似。所以进入步骤 4，取 e1 为<acgtt>。进入步骤 3 比较 e1 与 e2，不相似所以进入步骤 4。

g 取 e1 为<cgttc>，进入步骤 3 与 e2 比较,结果为不相似，所以进入步骤 4，因为比较次数已为 2 所以进入步骤 5。

h 因为 e2 已到尾部转向 6，因为投影数据库中只有两条序列，所以已经比较完成，最终从 2 转向 7。运行结束。

向右扩展得到的投影数据库为:

表 16. 投影数据库

Table16. the Projected Databases

子串	投影数据库
<tgcaggttcc >	<tgcaggttcc:12 >
< tgcagcttcc>	< tgcagcttcc:24>

(3) 向左扩展

a 因为 e1 向左扩展只有 2 位长度所以 e1 为<ta>，e2 为<tc>

b 比较 e1 与 e2 相似度，结果相似，进入步骤 1，因为投影数据库中只有两条子序列，所以从步骤 2 转向 7，因为支持数不小于支持度，所以<ta>与<tc>向左的投影数据库运行上面步骤，但是这两子序列的投影数据库中只有 1 条序列所以运行结束。

4.5 实验结果与分析

下表为从 dnaY 中挖掘最小长度为 200，支持度为 10 的序列模式中一个结果的一部分。

表 17.　实验结果

Table 17.　experimental result

编号	序列
1	tctgcagcttcattcttgaagtgagcaagaccatgaacctgccagaa
2	actggagcttcattcttgaagtgagcaagaccatgaacctgcccg
3	Tctgcagcttcattcttgaagtgagcaagaccatgaacct__ccca
4	tgtgcagcttcattcttgaagtgagcaagaccatgaacctgct

从上面的实验结果可以看出，在将 BLAST 算法融入 PrefixSpan 算法后挖掘出的实验结果可以容忍同源序列的小部分变异和缺失。

将 PrefixSpan 投影数据库生成的方法加入 BLAST 后提高了算法的执行速度（如图 10 所示，此结果为挖掘 dnaY 序列中的频繁模式，设置支持度为 10，最小长度为 X 轴的值），没有加入 PrefixSpan 思想时执行方法为：用 PrefixSpan 算法找出长度为 MIN_LEN*M%的完全匹配，再在这些完全匹配的基础之上两两比较，比较时取出两条完全匹配的子序列，向左右比较直到不再相似，再取下一条进行比较。

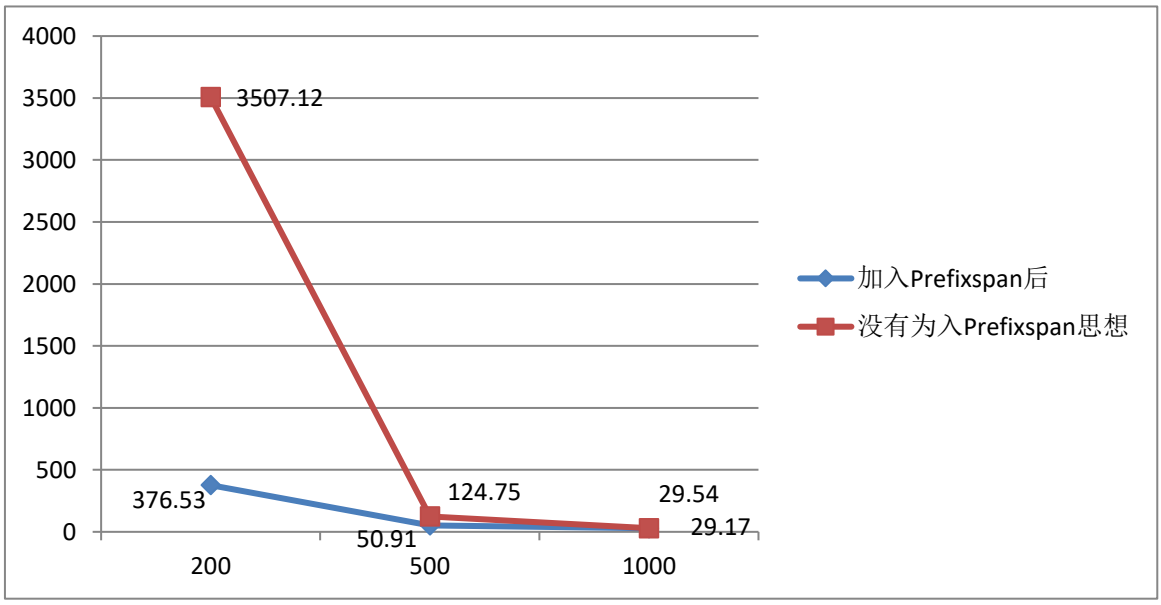

图 10　执行时间

Fig.10 execution time

5 改进的 PrefixSpan 算法在 MPI 和 Hadoop 上的实现

改进后的 PrefixSpan 算法的性能有很大的提升，但是随着输入序列的规模增加，运行的时间还是越来越长。如在单机上输入序列 dnaY 设置最小长度为 100 支持度为 10，算法运行的时间为二十秒左右，但是如果将 dnaY 变为 dan1 运行的时间将达到三百多秒，这运行时间有时是让人无法忍受的。在这一章本文将给出一种

PrefixSpan 算法并行的方法，并且同时在 MPI 和 Hadoop 平台上实现并行，两种平台都配置在一个集群上，这个集群由 10 台虚拟机构成，这 10 台虚拟机通过 Vmware vSphere 的虚拟化技术装在一台 IBM X5 3850 服务器上，同时把另外一台万全 R520 服务器作为 Vmware vCenter Server 管理端，通过将集群配置到虚拟机上不仅可以便于管理，而且可以将程序在并行平台上运行的时间与在单台服务器上运行时间比较，来分析并行平台的利与弊。

5.1 集群与服务器的配置

表 18. 服务器配置

Table18. The server's configuration

服务器名称	cpu	内存
万全 R520	2*4*2	16G
IBM X5 3850	2*6*2	64G

因为 IBM X5 3850 有 12 个 cpu 核心，64G 内存，所以在这台服务器上配了 10 台虚拟机，虚拟机的配置如下：

表 19. 虚拟机配置

Table19. VM's configuration

主机名	vcpu 数量	内存大小	主机 ip
centos_master	2	8G	192.168.0.254
centos1	1	6G	192.168.0.241
centos2	1	6G	192.168.0.251
centos3	1	6G	192.168.0.252
centos4	1	6G	192.168.0.253
centos5	1	6G	192.168.0.247
centos6	1	6G	192.168.0.249
centos7	1	6G	192.168.0.250
centos8	1	6G	192.168.0.248
centos9	1	6G	192.168.0.242

5.2 MPI 与 Hadoop 的配置与编程模型

MPI 与 Hadoop 都是现在比较流行的并行技术，本文使用 MPI 的版本为 mpich2-1.4.1，本文使用的 Hadoop 版本为 hadoop-2.2.0。在下面本文将简要地介绍一下 MPI 和 Hadoop 的编程模型。MPI 和 Hadoop 的编程模型相似（如图 11 和图 12），但是它们也有不同点，比如 MPI 不会自动地 reduce，所以需要自己去从各个机器上将得到的结果复制到本地， Hadoop 会将输出结果保存到 HDFS 中，我们只需要从 hadoop 文件系统将结果提取到本地。另外一个为同点就是在 Hadoop 中它自动地将输入文件分成块输出到不同的 map 中不受用户控制，而在 MPI 中则需要自己写分配的方法用户可以自由控制。

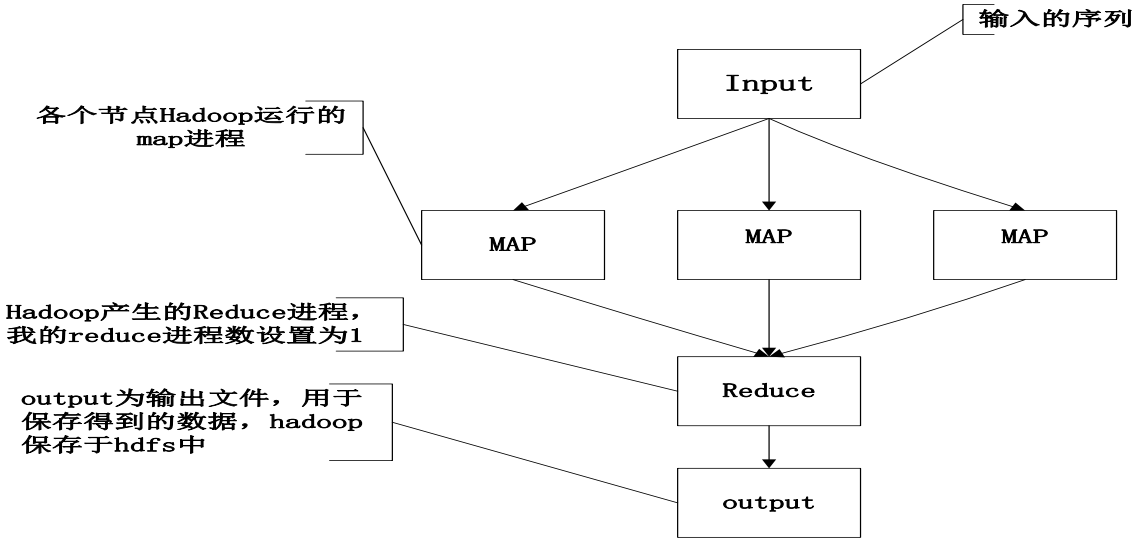

图 11　Hadoop 模型

Fig.11　The Mode of Hadoop

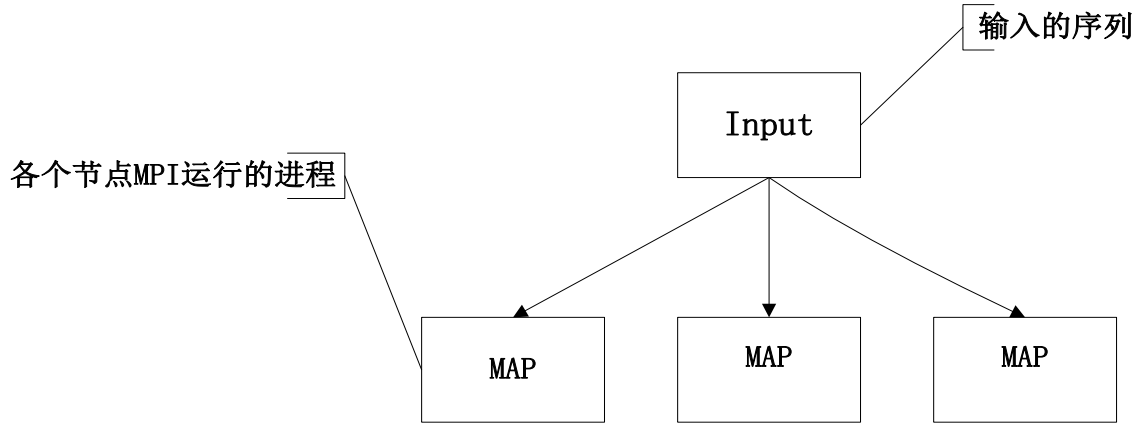

图 12　MPI 模型

Fig.12　The Mode of MPI

5.3 PrefixSpan 算法在 Hadoop 和 MPI 平台中的实现

5.3.1 MPI 中 PrefixSpan 算法的实现

MPI 相当一个 C 语言的扩展库，在编写程序时只需要将 MPI 的头文件”mpi.h”引入程序则可以调用 mpi 的各种扩展程序，mpi 程序运行时要调用 MPI_init()初始化。PrefixSpan 算法在 MPI 上实现方法如下：

(1) dfile 函数，在本地读取输入的要被挖掘的 dna 序列。

(2) 运行 init_node 函数，初始化 dna 序列，初始化 dna 序列所做的工作为：

a 建立一个长度为 4**NODE_LEN 长度的 hash 表 nodes[4**NODE_LEN]。

b 遍历整个 dna 序列（遍历时向前推进的速度为 1），将最新得到的长度为 NODE_LEN 个碱基转变为四进制，假设得到的值为 this_sequnce_num，则将 nodes[this_sequnce_num]中用于保存 this_sequnce_num 出现次数的值加 1。

c 遍历整个 nodes，取出出现次数满足支持度的所有值，并将结果分为 n 份（n 为参与运行的节点数）每个节点只保存自己将要运行的那一份，并将他们保存于链表 cutednode 中。

(3) 运行主函数 mainrun，mainrun 的主要功能如下：

a 从链表 cutednode 中取出 ONECE_S_NUM 个子序列，作为 get_shadow_data 的输入，get_shadow_data 函数的功能为遍历原始的 dna 序列，生成这 ONECE_S_NUM 个子序列的投影数据库，并分别保存于 nodeaddress 中。

b 分别对每个子序列的投影数据库调用 get_node 函数进行处理，get_node 的功能为取出投影数据库中每个序列的前 NODE_LEN 个序列，记录它们出现的次数，保存满足支持度的序列，生成它们投影数据库直到不存在满足条件的序列存在。

c 重复 ab 直到子序列运行完。

下面使用流程图的形式介绍本文的程序运行流程。程序的整体流程图 13 所示。Mainrun 的流程图如图 14 所示。

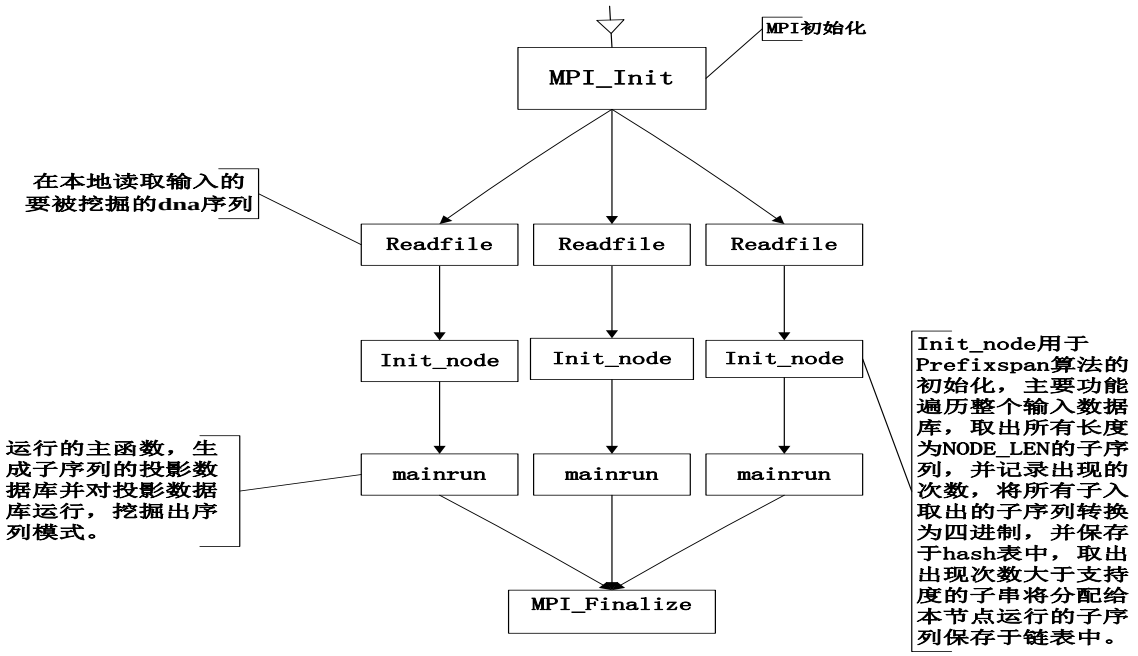

图 13　PrefixSpan 算法在 MPI 中的执行流程

Fig.13　The Execution Flow Of MPI

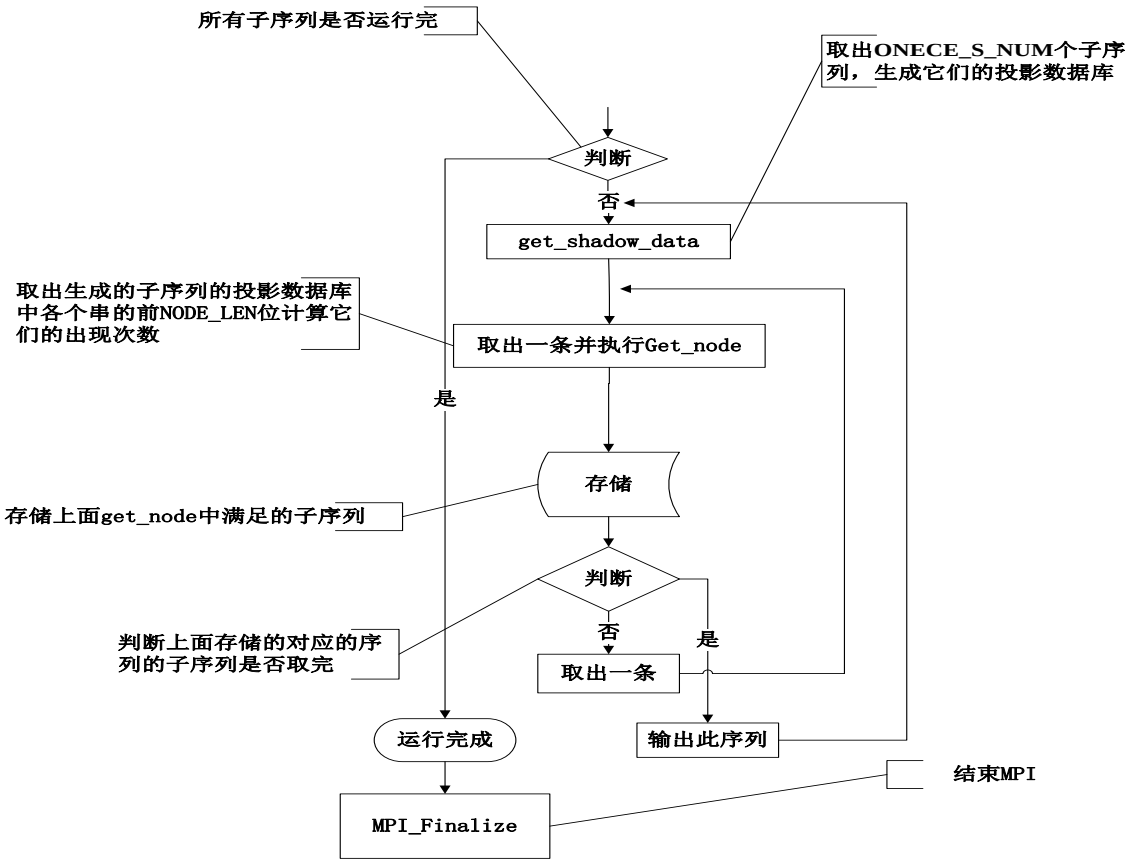

图 14　mainrun 的执行流程

Fig.14　The Execution Flow Of mainrun

5.3.2 Hadoop 中 PrefixSpan 算法的实现

Hadoop 与 MPI 的编程相似，只是将 MPI 中 init_node 中的部分功能写入到了一个新的程序 init 中，用于初始化输入序列，将所有长度为 NODE_LEN 且满足支持度的序列取出保存于文件 init_file 中，然后将 init_file 作为 hadoop 的输入文件，将 init_file 中的值发送到各个 map 进程。Hadoop 整体的执行流程如图 15 所示。

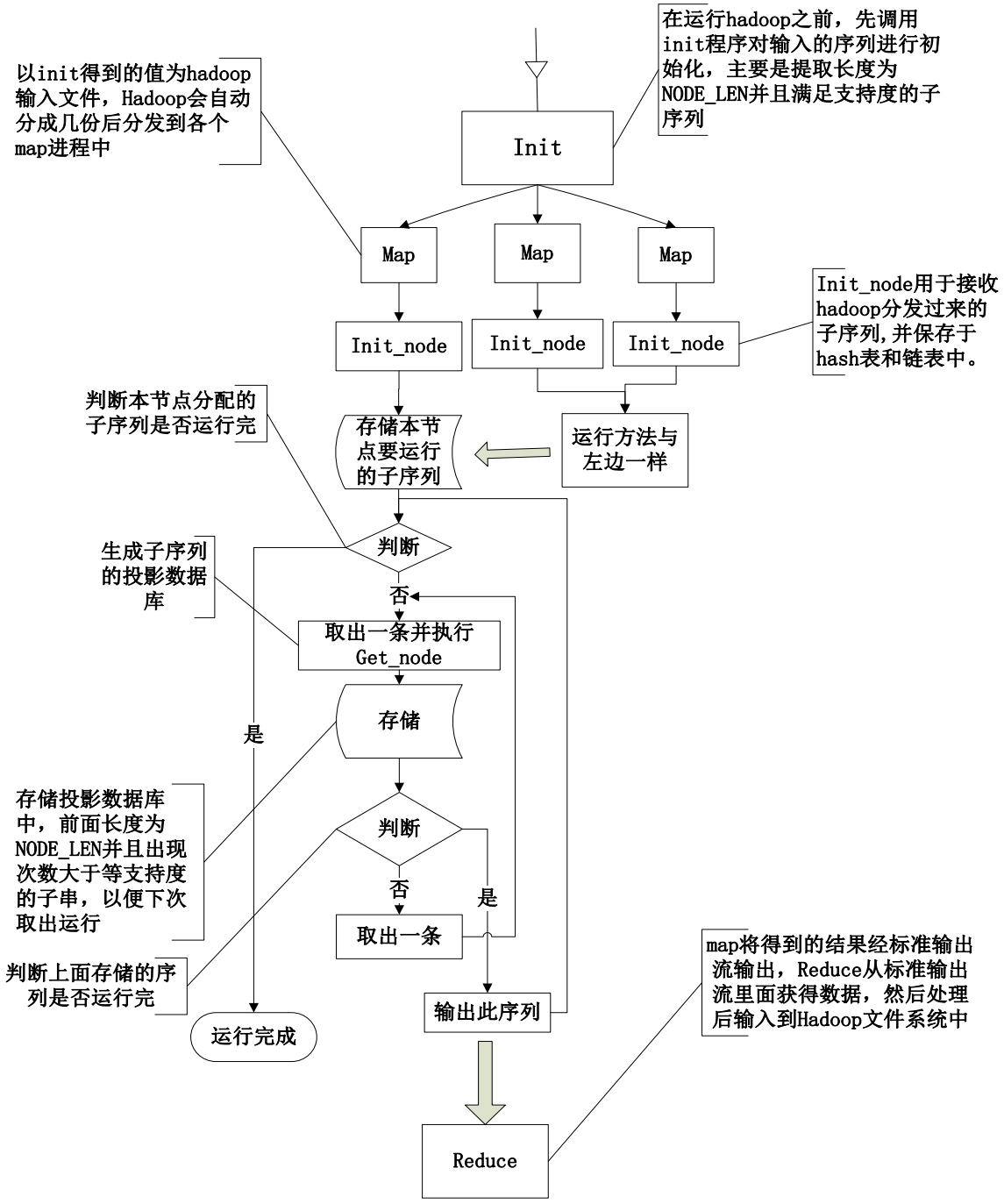

图 15　PrefixSpan 算法在 hadoop 中的执行流程

Fig.15　The Execution Flow Of hadoop

6 实验结果及分析

实验结果出自三个平台上的运行，第一：MPI 并行平台。第二：hadoop 并行平台。第三：服务器上的一台虚拟机，它被分配的资源与所在的主机相同，当此台机器运行时其它虚拟机停用，保证将服务器的资源全部分配给它。这一小节本文将呈现将 PrefixSpan 算法在不同平台上挖掘不 dna1 序列中频繁模式所花费的时间。并对结果进行分析。

各个平台运行挖掘 dna1 的时间如图 16 所示，加速比如图 17 所示。

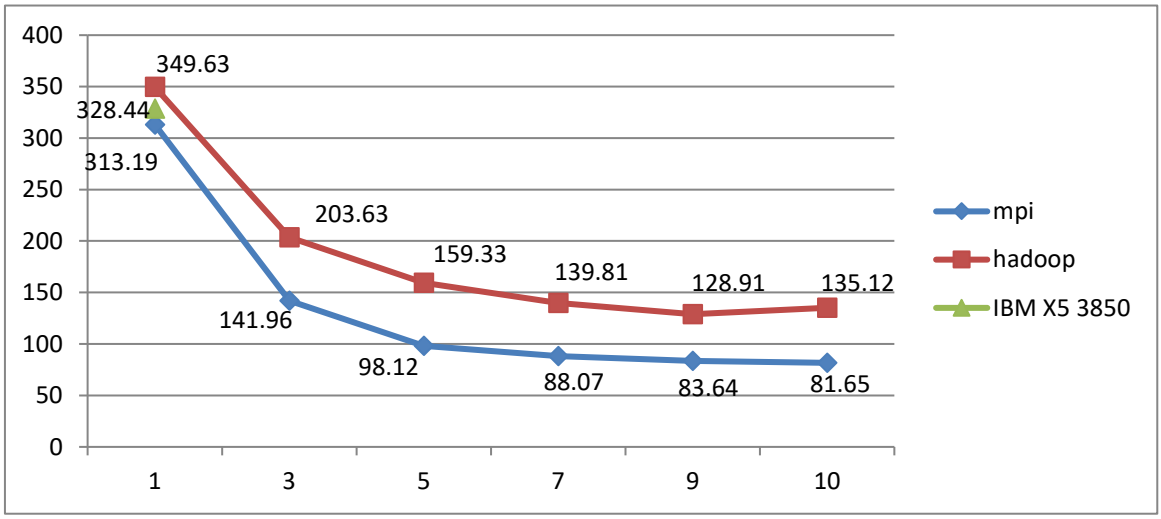

图 16　运行时间

Fig.16　Running Time

注：X 轴为并行的节点数，Y 轴为运行时间单位 S

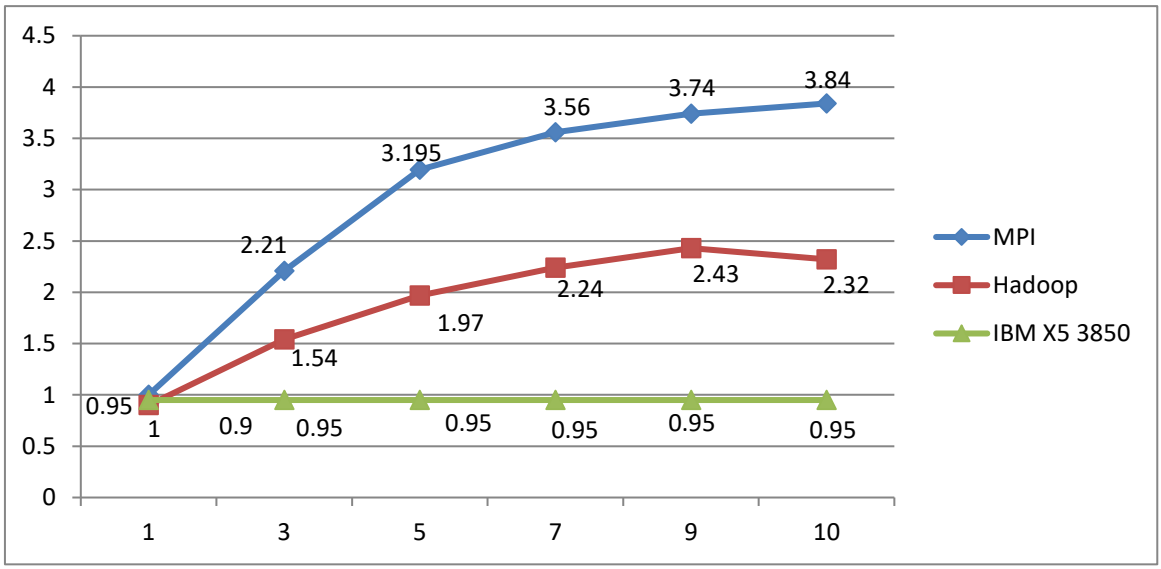

图 17　整体加速比

Fig.17　speedup

Hadoop 和 MPI 的程序模型都是将输入的数据分为多个部分，每个节点执行一部份，程序运行完成之后再将得到的结果输出到主节点。从图 16 可以看出 MPI 平台中随着节点数的增加程序的运行时间逐渐缩短，Hadoop 的节点数在 9 以内时也随节点数的增加运行时间缩短，但是当从 9 个节点增加到 10 个节点时运行时间反而增加了，这是因为集群的规模为 10 台机器，其中只有 9 台 daanode，所以运行时最多只有 9 台能同时间产生 map 进程，设置为节点数为 10 时必将导致一个 map 程序要等待其它 map 运行完成之后才能运行，所以导致运行时间增加。

当节点数增加时程序运行的时间逐渐缩短，理想中的情况应该是当节点数从 1 增加到 10 时运行的时间也应该由三百多秒钟减少到三十多秒，但是各个平台中效率最高的 MPI 平台，当节点数增加到 10 时运行时间还是八十多秒，下面本文将对此原因进行分析。第一由于在为各个节点分配任务时输入数据可能会分配不公平，导致各个节点运行时间不同，从而影响整个程序的性能，如图 18 所示在 hadoop 运行同一个程序时，各个 map 的结运行时间相差很大，但是由于 MPI 能自由控制输入的自由分配，所以可以自己设定分配法则，但分配公平，所以各节点运行时间相差不大。

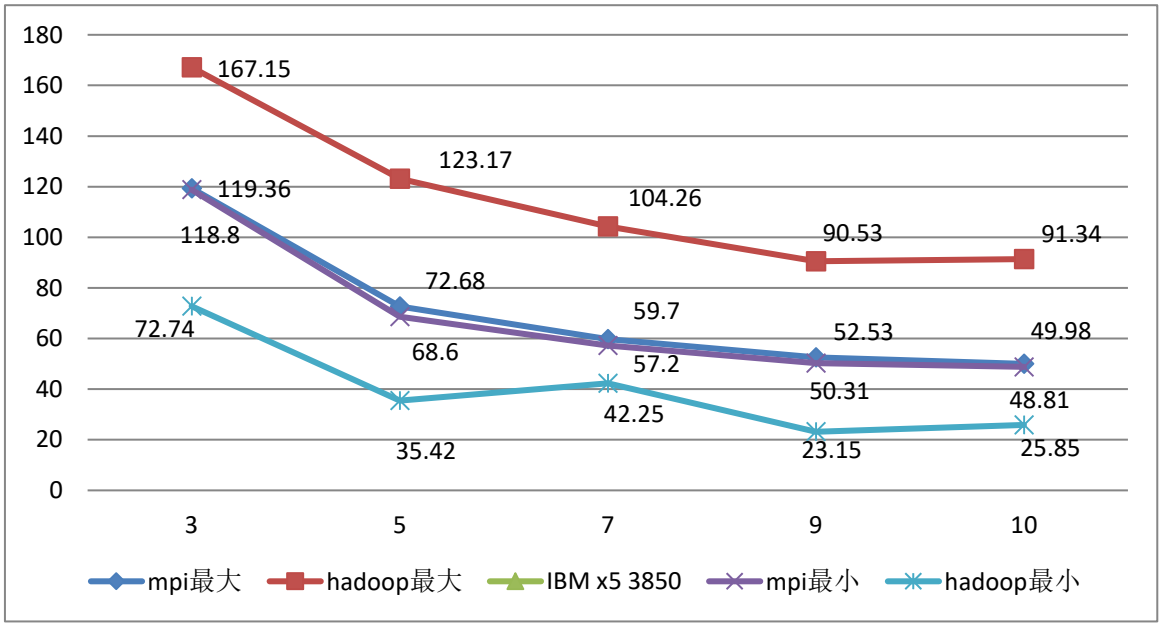

图 18　mainrun 运行时间

Fig.18　Running Time of mainrun

第二，由于本文的程序中要对输入的 DNA 序列进行预处理，并且每个节点都要处理一次，所以这部分不能并行，所以这部分不能并行，所以这部分会导致程序的整体性能下降。图 19 和图 20 为程序中的 get_shadow_data 和 get_node 函数运行时间和加速比，由于这两个程序处于可并行的部分，所以他们的加速比接近 10。

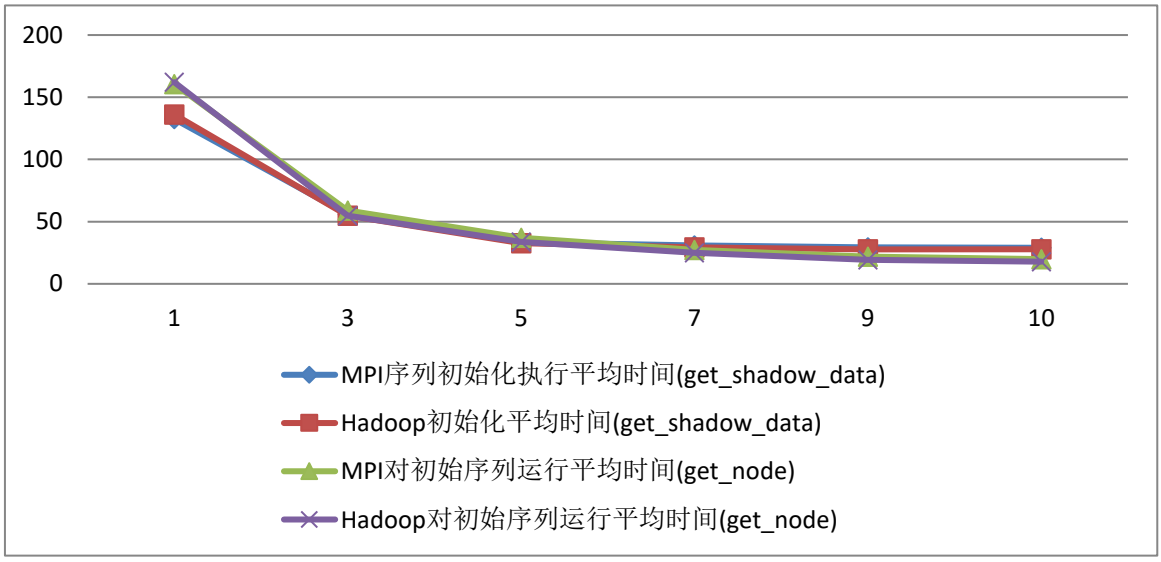

图 19　get_shadow_data 和 get_node 的运行时间

Fig.19　Running Time of mainrun get_shadow_data and get_node

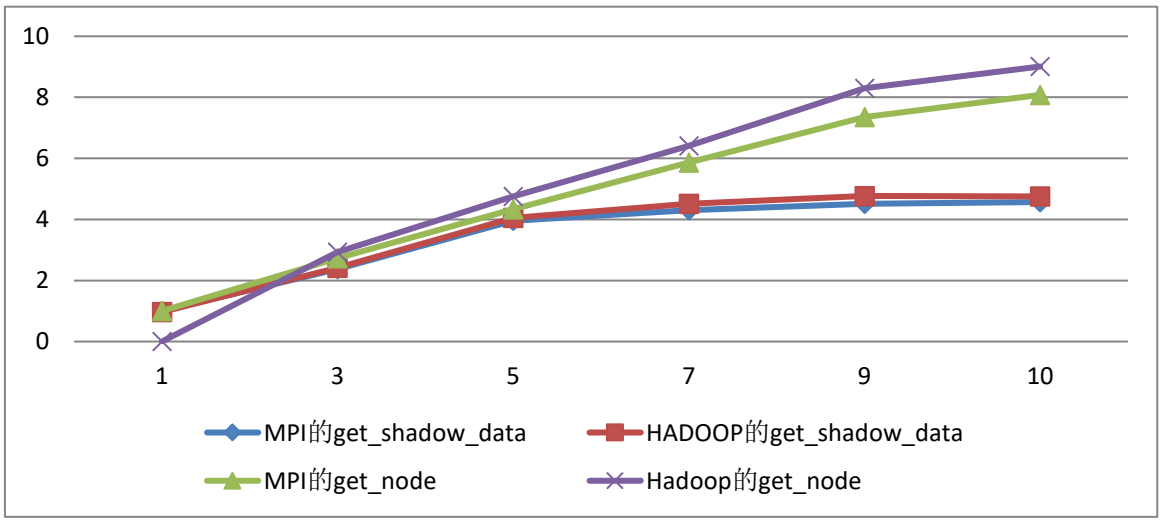

图 20　get_shadow_data 和 get_node 的加速比

Fig.20　the speedup of get_shadow_data and get_node

图 16 中的结果是将 PrefixSpan 算法应用于 dna1（碱基个数为 229446296）的序列模式挖掘所得到的程序整体运行时间，从图 16 可以看出 PrefixSpan 算法在 MPI 平台上运行的效率最高，当 PrefixSpan 算法运行在一个节点上时，效率最高的也是 MPI，其次是服务器单机运行，hadoop 的运行效率最低。但是当 MPI 只在一个节点上运行时，它只在 master 上运行，本文为它分配的 cpu 只有两个，但是服务器上单机运行时分配的 CPU 有 12 个，且内存为 64G，但是服务器上运行的效率却没有 MPI 一个节点运行的效率快，这说明当 PrefixSpan 运行在服务器上时，资源利用率非常低，这是也当前为什么虚拟化比较流行的原因之一。另外可以看出 hadoop 的效率比 MPI 和服务器上运行的效率低很多，其中一个原因就是 hadoop 系统运行时它本身占

用了很大的时间，比如将输出结果写入 hadoop 文件系统中，运行时 hadoop 从 hadoop 文件系统中读取信息，这些操作之中包含了大量的网络传输，网络传输的效率非常低，所以导致 hadoop 效率下降。

7 总结研究的成果和讨论未来的工作

7.1 研究的成果

由第 3 章我们知道 PrefixSpan 算法并不适合于 dan 序列中的频繁模式的挖掘或者可以说成将原始的 PrefixSpan 算法应用于 dna 中的频繁子序列的挖掘有严重的缺陷，所以本文根据 dna 序列的特点，改进 PrefixSpan 算法，使其能挖掘单条 dna 序列中的频繁子序列。另外 PrefixSpan 算法在执行中大量时间都花费在了投影数据库的产生上，所以这对 PrefixSpan 算法的效率影响较大，本论文的另一个成果就是将引入分块的策略，将输入的序列分块，这种策略可以减少投影数据库的规模，这大大提高了 PrefixSpan 算法的性能。此外本文根据 dna 序列的特性提出了四进制的方法，将碱基转换为四进制形式，再通过 hash 表的方法，加快了字符串间的比较和查找速度，这将 PrefixSpan 算法的性能提升一倍左右。

提出一种将 BLAST 思想与 PrefixSpan 融合的方法，可以在挖掘 DNA 序列时容忍同源子序列存在小部分变异和缺失。

虽然对算法的改进可以增加算法的效率，但是增加的效率可能是有限的。随着高性能支持的发展，高性能计算已成为现在的潮流，使用并行技术对算法的效率提升可能是无限的（只要集群够多），所以本论文的另外一处成果就是提出了一种 PrefixSpan 算法的并行方法，并且在 MPI 平台和 Hadoop 平台上实现了并行，并对实验结果进行了分析。

7.2 未来的工作

这篇论文中对 PrefixSpan 算法的提升主要是围绕 dna 序列中频繁模式的挖掘来展开的，所以本论文中提出的对 PrefixSpan 算法改进的方法仅限于少数的几种序列类型，所以本文的一个工作就是考虑如何将此论文中提出的改进方法扩展到其它的数据类型的序列模式挖掘中，比如引入分块的策略和四进制编码的方法。另外一点就是在本文提供的并行方法中还存在不能并行的模块，这些模块也成为了整个程序的一个累赘，它将整个程序的运行时间拉长，所以本文未来的另一个工作就是进一步改进并行算法。本文提出的将 PrefixSpan 算法与 BLAST 思想相融合的方法可能会丢失部分输出结果，另外一点就是将 BLAST 和 PrefixSpan 算法相融合后 PrefixSpan 算法的性能会下降，所以本文提出的融合方法有待改进。

参考文献

[1] 陈卓，杨炳儒，宋威等．序列模式挖掘综述[J]．计算机应用研究，2008, 25:1960-1967

[2] 郭本俊，王晨，陈高云．基于 MPI 的云计算模型[J]．计算机工程，2009，35：84-86

[3] 刘立军，崔 杰，梅红岩．GSP 与 PrefixSpan 算法的比较与分析[J]．辽宁工学院学报，2006，26：300-302

[4] 刘 栋，尉永清，薛文娟．基于 Map Reduce 的序列模式挖掘算法[J]．计算机工程，2012，38：43-44

[5] 李红燕．基于 BLAST 算法的序列分析软件开发[D]．湖南：中南大学图书馆，2009

[6] 马燕，范植华．基于神经网络的基因分类器[J]．计算机工程与设计，2005，26：308-311

[7] 孟小峰， 李勇，祝建华．社会计算：大数据时代的机遇与挑战[J]．计算机研究与发展，2013，50：2483-2491

[8] 魏顺平．学习分析技术：挖掘大数据时代下教育数据的价值[J]．现代教育技术，2013，23：5-11

[9] 武珍珍．改进的 PrefixSpan 算法在生物序列模式挖掘中的应用[D]．广州：中山大学图书馆，2010

[10] 吴楠，胡学钢．基于 PrefixSpan 序列模式挖掘的一种改进算法[J]．电脑知识与技术，2007，10：479 -480

[11] 于啸， 孟繁疆， Yu Xiao 等．数据挖掘技术在生物信息学中的应用[J]．农机化研究，2009，31：186-188

[12] 张建勋，古志民，郑超．云计算研究进展综述[J]．计算机应用研究，2010，27：429-433

[13] 周斌，吴泉源．序列模式挖掘的一种渐进算法[J]．计算机学报，1999，22：882-887

[14] 张静， ZHANG Jing．数据挖掘在生物信息中应用的现状及展望[J]．电脑知识与技术，2008，2：816-817

[15] 张巍，刘峰，滕少华．改进的 PrefixSpan 算法及其在序列模式挖掘中的应用[J]．广东工业大学学报，2013，30：49-54

[16] Chia-Ying Hsieh. “An Efficient Sequential Pattern Mining Algorithm Based on the 2-Sequence Matrix, [A]” IEEE International Conference on Data Mining Workshops, 2008.

[17] Dhany S. “Sequential Pattern Mining using PrefixSpan with Pseudoprojection and

Separator[A]” International Symposium on Information Technology, IEEE conf , 2008.

[18] J. Pan, P. Wang, W. Wang, B. Shi, and G. Yang, "Efficient Algorithms for Mining Maximal Frequent Concatenate Sequences in Biological Datasets[A]" In proceedings of the 5th International Conference on Computer and Information Technology(CIT,2005), Shanghai, China, vol. 1. pp. 98-104, Sept. 2005.

[19] Konstantin Shvachko, Hairong Kuang, Sanjay Radi，et al . The Hadoop Distributed File System[A] , In Proc. of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) , 2010

[20] Kang Tae-Ho, Yoo Jae-Soo, Mining Maximal Frequent Contiguous Sequences in Biological Data Sequences[J]. Korea Information Processing Society . 2008, pp.155-162

[21] Liu Pei-yu, Gong Wei, Jia Xian, An improved PrefixSpan algorithm research for sequential pattern mining[J]. In Proc. of the International Symposium on IT in Medicine and Education (ITME),2011

[22] Pei Jian, Han Jiawei, Behzad M ortazavi-Asl, Helen Pinto. PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth[A]. In proc. of the IEEE International Conference on Data Engineering, Germany, 2001

[23] Ronald C Taylor. An overview of the Hadoop/MapReduce/HBaseframework and its current applications in bioinformatics[J]. BMC Bioinformatics 2010:1-6

[24] Tae Ho Kang, Jae Soo Yoo, Hak Yong Kim, "Mining Frequent Contiguous Sequence in Biological Sequences[A]", In Proc. of the Proceedings of the 7th IEEE International Conference on Bioinformatics and Bioengineering, 2007

[25] Yong-qing Wei, Dong Liu , Lin-shan Duan. Distributed PrefixSpan algorithm based on MapReduce[J]. In Proc. of the International Symposium on Information Technology in Medicine and Education (ITME), 2012