

采用 BWT 的多核并行的子串匹配算法

王佳英,王 斌,李晓华,杨晓春
(东北大学 计算机科学与工程学院,辽宁 沈阳 110819)

摘 要 :针对 P-BWT 精确匹配算法存在只支持短串查询并且只能工作在单处理器上的问题,提出了一个多核并行的支持任意查询长度的精确查询算法.改进了 P-BWT 索引上的查询过程,当一个查询串跨越了多个数据分片时,首先在其匹配的最后一个分片上查询,然后依次在前面分片上进行验证.进一步提出了一个多核并行查询算法来减少搜索和验证过程的迭代次数.实验结果表明,所述算法可以高效并行地完成子串匹配任务.

关 键 词 : BWT ;全文索引 ;精确匹配 ;并行 ;多核

中图分类号 : TP 311.13 文献标志码 : A 文章编号 : 1005-3026(2016)05-0624-05

Multi-core Parallel Substring Matching Algorithm Using BWT

WANG Jia-ying , WANG Bin , LI Xiao-hua , YANG Xiao-chun
(School of Computer Science & Engineering , Northeastern University , Shenyang 110819 , China. Corresponding author : WANG Bin , E-mail : binwang@mail.neu.edu.cn)

Abstract : In order to solve the problem that P-BWT (Burrows-Wheeler transform) could only support short queries , and work on a uniprocessor , a multi-core parallel exact matching algorithm was proposed which any query length could be supposed. Firstly , the search process on P-BWT index was modified. When a query spans multiple data fragments , it first searches on the last segment , then verifies on the other segments. Further , a parallel algorithm was proposed to reduce the iterations in the search and verify process. Finally , the experimental study show that using the proposed algorithm , the substring matching task could be accomplished efficiently in parallel manner.

Key words : BWT(Burrows-Wheeler transform) ; full text index ; exact matching ; parallel ; multi-core

近些年来,网络数据、记录数据以及生物数据等文本数据增长迅速^[1-2].在文本上进行精确的子串查询是工业界和学术界中的一个常见应用,同时也是子串近似匹配的一个基础操作^[3-6].BWT(burrows - wheeler transform)是对一个文本数据的等长转换,转换之后的数据便于压缩.该方法广泛地应用于无损压缩领域,例如开源压缩工具 bzip. BWT 还可以做为一种全文索引来支持快速的子串查询^[7],该方法被称为 BWT 反向搜索方法.利用 BWT 索引进行子串查询的好处是可以同时支持数据的压缩存储和快速查询.

尽管 BWT 索引可以支持数据压缩,但对于大规模的文本数据构建索引仍需要消耗大量的内存空间^[8].当内存不够时,一些方法选择利用外存来帮助构建索引^[9-10].然而外存方法要比内存方法效率低几千乃至几万倍.通过对数据集进行划分可以有效地解决空间问题且同样可以支持精确子串查询^[11].但该方法有以下不足:首先,原有方法只支持短串查询,即查询串长度要小于等于一个数据分片的大小.而对于许多应用场景,在构建索引前,用户并不能确定查询串的长度.其次,现在多核架构已经成为了主流的计算机架构.多

核处理器可以提高计算效率,减少计算能耗.结合多核计算资源可以提高子串匹配速度.而原有方法只能在单处理器上工作.本文提出了基于分割的 BWT 索引的支持任意查询串长度的查询方法,提出了在多核架构下并行的基于 BWT 索引的精确查询方法.

1 背景知识与问题定义

1.1 BWT 和 P-BWT 索引

考虑字符串 $S[1..n]$,其中 n 是字符串的长度.每个 $S[i]$ 属于一个有限的字符集 Z ,其中位置 i 满足 $1 \leq i \leq n$ 并且 $|Z| = \sigma$.

字符串 $S[i..j]$ 是一个 S 中从位置 i 开始到位置 j 结束的一个子串.当 $i > j$ 时, $S[i..j]$ 代表一

个空串.如果 $i = 1$,字符串 $S[1..j]$ 被称为 S 的一个前缀;如果 $j = n$,字符串 $S[i..n]$ 被称为 S 的一个后缀.

为了方便子串比较,一个特殊字符 $\$$ 被添加到字符串 S 后面,其中字符 $\$$ 小于所有字符集 Z 中的字符.本文用字符串 T 表示对字符串 $S \$$ 进行转换后的结果.它同样是一个 $n + 1$ 长的字符串.字符串 T 和 $S \$$ 的区别只是其中字符的顺序不同.

图 1 左部分展示了对字符串 $S = \text{mississippi}$ 进行 BWT 转换的过程.转换后的结果为 $T = \text{ipssm \$ pissii}$.BWT 转换的结果串 T 和后缀数组 SA 之间的关系如公式 (1) 所示.

$$\pi[i] = \begin{cases} S[SA[i]-1], & \text{if } SA[i] \neq 1; \\ \$, & \text{if } SA[i] = 1. \end{cases} \quad (1)$$

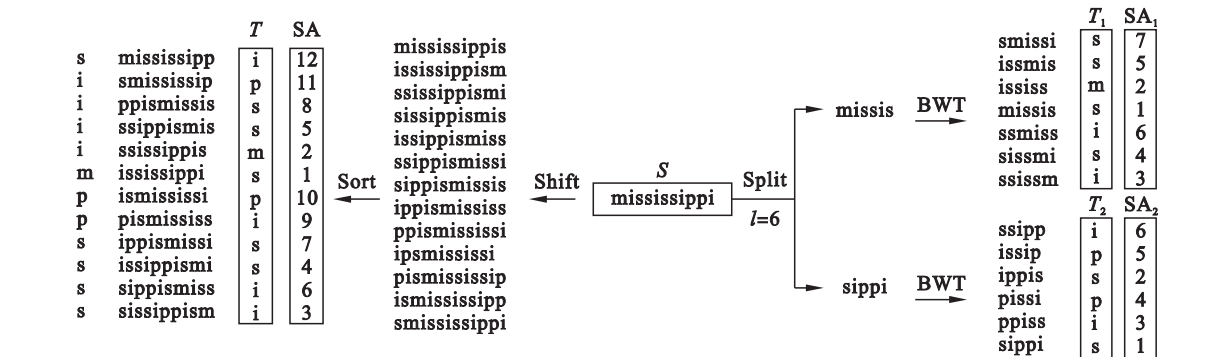


图 1 BWT 和 P-BWT 转换
Fig. 1 BWT and P-BWT transformation

图 1 右部分展示了分割的 BWT 索引(P-BWT)的转换过程.首先将原字符串 S 分割为长度为 l 的数据片段.由于该索引将原串分割为长度为 l 的数据片段,每次只需要将一个数据片段载入内存,进行 BWT 转换,因此构建该索引的空间复杂度为 $l \times (l \log l + l \log \sigma)$.

1.2 问题定义

给定一个字符串 S 和一个查询串 P ,子串精确匹配问题是要找到字符串 S 上所有精确匹配的子串 $S[i..j]$,其中 $1 \leq i \leq j \leq n$.图 2 展示了一个子串精确匹配的例子.



图 2 字符串精确匹配问题

Fig. 2 String exact matching problem

2 子串精确匹配算法

首先讨论基于 P-BWT 的精确长串匹配算

法.然后基于 P-BWT 索引,提出子串精确匹配算法.

2.1 长串精确匹配的算法

假设查询串 P 的长度为 m .P-BWT 的分片长度为 l .文献 [11]讨论了短串精确匹配的方法,即当 $m \leq l$ 时的情况.本节讨论长串的精确匹配方法,即当 $m > l$ 时的情况.图 3 展示了一个长查询串精确匹配的情况.

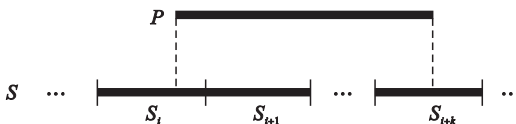


图 3 长串精确匹配情况

Fig. 3 A long substring matching case

当一个查询串的匹配子串跨越了多个数据片段 $S_i, S_{i+1}, \dots, S_{i+k}$ 时,本文方法的思路如下.首先在其匹配的最后一个片段 S_{i+k} 上进行查询,找到与查询串 P 的一个后缀 $P[j_{k+1}..m]$ 相匹配的字符串 S_{i+k} 的前缀,然后在 S_{i+k-1} 片段上进行反向扩展验证,判断是否该查询串子串 $P[j_{k-l+1}..j_k]$ 与整个

分片 S_{i+k-1} 相匹配. 如果该匹配没有结束, 则在之后的分片 $S_i \sim S_{i+k-2}$ 上进行同样的操作来验证该查询串 P 的其他子串. 整个过程在验证到 P 的开始位置结束.

2.2 基于 P-BWT 的子串精确匹配算法

综合上述匹配场景分析, 基于 P-BWT 的子串精确匹配算法分为两个步骤: 步骤 1 是反向搜索每个分片上的查询串的候选者; 步骤 2 是反向验证后一个数据分片搜索的待验证候选者. 算法 1 给出了具体的算法执行过程.

算法 1 ExactMatch(I_S, P)
输入: I_S 为字符串 S 的 P-BWT 索引, P 为查询串.
输出: A 为 P 在 S 上的所有精确匹配子串.
1) 初始化结果集合 $A = \emptyset$,
2) For $i \leftarrow \lceil \frac{n}{l} \rceil$ to 1 // 从后向前循环分片
 计算步骤 1 BackwardSearch,
 3) 在数据分片 I_S^i 上反向搜索查询串 P 的候选者集合 C ,
 4) For 每个候选者 $c \in C$
 5) If $c = P$ // 验证候选者
 6) $A \leftarrow c$
 7) Else If c 是 S_i 的前缀
 8) 将该候选者 c 记录在 I_S^{i-1} 的待验证列表中;

 计算步骤 2 BackwardVerify,
 9) If $i < n$
 10) While I_S^{i+1} 分块待验证列表 L_c 非空
 11) 取出一个候选者反向验证.

步骤 1 算法反向搜索每个数据分块 I_S^i 上该查询串 P 的后缀串, 其中 $1 \leq i \leq n$, 见算法 1 中行 3). 当以下两种情况满足时, 算法将该匹配串看作是 P 的候选者: ① 当该匹配串结束于查询串的第一个字符; ② 当该后缀结束于当前数据分片开始位置. 找到每个分块上的候选者之后, 算法将对每个候选者其进行验证并将满足条件的候选者记录在结果集中, 见算法 1 中行 6). 如果该数据分片无法完成全部验证, 即该匹配跨越了两个或多个数据分片, 它将该候选者记录在其前一个数据分片 I_S^{i-1} 的待验证列表中, 见算法 1 中行 8).

步骤 2 算法反向验证候选者. 该过程检查验证列表中的每个候选者, 见算法 1 中行 10) 和 11). 验证过程中会有 3 种情况发生: ① 当该候选者等于查询串, 算法将保存这个候选者到结果集中; ② 当该候选者验证出错, 算法将直接丢弃该候

选者; ③ 当该分片仍然无法完成验证时, 即该分片跨越了多个数据分片, 算法将保存该候选者到其前一个数据分片 I_S^{i-1} 的待验证列表中.

该算法的时间复杂度为 $O(o + \frac{nm}{l})$, 其中 o

为该子串在数据串 S 上出现的次数, n 为该数据串长度, m 为查询串长度, l 为数据分片大小.

3 支持多核架构的精确匹配算法

3.1 多核架构下的匹配策略

本节讨论如何在多核架构下更高效地完成匹配任务. 根据基于 P-BWT 索引的精确匹配算法, 本文将并行任务分为两种: 一种是反向搜索, 一种是反向验证. 其中对于一个数据分片来说, 反向搜索只需执行一次, 而反向验证需要迭代多次完成. 每次验证一组候选者.

图 4 展示了 P-BWT 索引在多核环境下的索引架构. 处理器上的每个核可以选择进行两种操作: 反向搜索和反向验证. 当一个处理器核完成对当前数据块的计算, 它将由外存换入新数据块到内存. 并在新的内存块上重复上述过程.

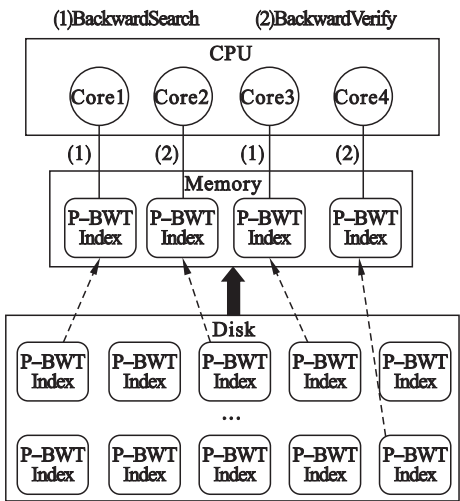


图 4 多核 P-BWT 索引框架
Fig. 4 Multi-core P-BWT index framework

由于候选者只有在反向搜索后才能产生, 因此一个简单的策略是将两个操作的执行分成两个阶段: 第一阶段, 各节点只处理反向搜索; 第二阶段节点只处理反向验证.

两阶段策略存在一些问题和不足. 首先, 反向验证只能在反向搜索之后开始, 影响整体并行性; 其次, 对于同一个数据, 迭代地进行反向验证操作可能需要对该数据块换入多次, 将会影响系统的执行效率. 最后, 如果一个数据块后面的数据块完

成了所有计算工作,那么对于当前数据块就不会有新的候选者.因此一个好的验证策略应该减少等待时间,减少迭代次数,并且最右侧数据块优先.

3.2 多核架构下的基于 P-BWT 的匹配算法

为了充分利用多核资源,减少迭代次数,一个好的执行顺序至关重要.根据 3.1 节并行策略的分析,本节给出了一个基于权重模型的多核并行匹配算法.

首先系统将每个数据块和每种操作绑定作为键值 $\langle I_s^i, \text{opt} \rangle$,也称为一个操作块,其中 I_s^i 是一个数据块的索引,而 opt 是反向搜索和反向验证两种操作中的一种.而它们的权重是对该数据块上该操作计算量的一个估计.对于反向搜索操作来说,该权重为在该数据块上预估的匹配数;而对于反向验证操作来说,该权重为该数据块的候选者数量.权重相同时编号大的数据块优先.

为了实现右侧数据块优先,在系统上维护一个令牌,获得令牌的数据块表示它的右侧数据块已经完成所有计算任务.令牌从数据块最后的数据块开始向前传递.获得令牌的数据块优先进行计算.

算法 2 给出了多核架构下的基于 P-BWT 的匹配算法.首先系统初始化权重列表,此时所有数据块的权重相同.然后将令牌放到最后一个数据块并更新其权重(见算法 2 中行 3).然后并行地在每个处理器核上选择一个权重最高的操作块 I_s^i, opt ,在数据块 I_s^i 上执行相应操作 opt (见算法 2 中行 5)和 6).在做完反向搜索后如果该分片有待验证候选者则应立即进行检查验证,从而减少换入次数(见算法 2 行 8).如果该数据块获得了令牌,则将令牌传递给他前面的数据块.在该轮操作结束后更新该操作块权重(见算法 2 行 13).

算法 2 ParallelMatch(I_s, P)

输入: I_s 为字符串 S 的 P-BWT 索引, P 为查询串.

输出: A 为 P 在 S 上的所有精确匹配子串.

- 1) 初始化结果集合 $A = \emptyset$,
- 2) 初始化权重列表,
- 3) $\text{Token} = \lceil \frac{n}{l} \rceil$ 并更新对应操作块权重,
- 4) 在多核并行执行以下操作,
- 5) 选择一个 $\text{score}[I_s^i, \text{opt}]$ 最高的操作块,
- 6) 在 I_s^i 上执行操作 opt ,
- 7) If $\text{opt} = \text{BackwardSearch}$

- 8) 在 I_s^i 上执行操作 BackwardVerify,
- 9) If $i = \text{Token}$
- 10) $\text{Token} = \text{Token} - 1$,
- 11) If $\text{Token} = 0$
- 12) 结束过程,
- 13) 更新权重块 I_s^i, opt .

4 实验与分析

硬件环境: Intel Core 4 核处理器,主频 2.93 GHz,内存 8 GB;软件环境: Ubuntu 操作系统, C++ version 4.45. 本文在两个真实的数据集上评价所提出的方法.

1) 英文数据集: 英文数据是从 Gutenberg 项目中获取的,包含了 1 GB 的英文文本,字符集大小为 236 B.

2) DNA 数据集: DNA 序列是从 UCSC golden path 项目中获取的,包含了 1 GB 的基因序列,字符集大小为 5 B.

实验随机选取 100 个字符集中的子串作为查询串,并计算平均的查询时间.本文用经典的 BWT 索引方法作为基准方法,对采用不同大小分片的索引构建内存开销和子串查询性能进行了对比.

图 5 展示了在英文数据集上的实验结果.其中图 5a 比较了在英文数据集构建 BWT 索引和 P-BWT 索引的内存开销. P-BWT 的内存开销远小于 BWT 的内存开销.随着数据分片大小的增加, P-BWT 的内存开销上升.图 5b 比较了在所构建索引上的查询时间.随着数据分片大小的增加,查询时间先减少后增加,在分片大小为 16 MB 时达到最小.利用多核计算可以进一步提高查询速度.平均提高速率接近最优的 4 倍.图 5c 比较了在不同长度查询串上的查询时间.查询时间首先随着查询串长度增加而减少,在超过 10^3 之后随长度增加而增加.该结果符合 2.2 节对子串查询的时间复杂度分析.

图 6 展示了在 DNA 数据集上的实验结果.其中图 6a 比较了在 DNA 数据集构建 BWT 索引和 P-BWT 索引的内存开销.和英文数据集类似,构建索引的开销也随着分片大小的增加而增加.图 6b 比较了不同长度的数据分片下各算法的查询时间.查询时间的最优值同样出现在分片大小为 16 MB 时.在 DNA 上的查询时间要比在英文数据集上的时间长,这是因为 DNA 字符集较小,因此子串的重复率高.大量的时间消耗在定位每

个匹配的位置上. 图 6c 比较了在查询串长度改变的情况下查询时间的变化情况. 查询时间同样先减小后增加, 查询时间的最小值出现在查询长度为 10^4 时.

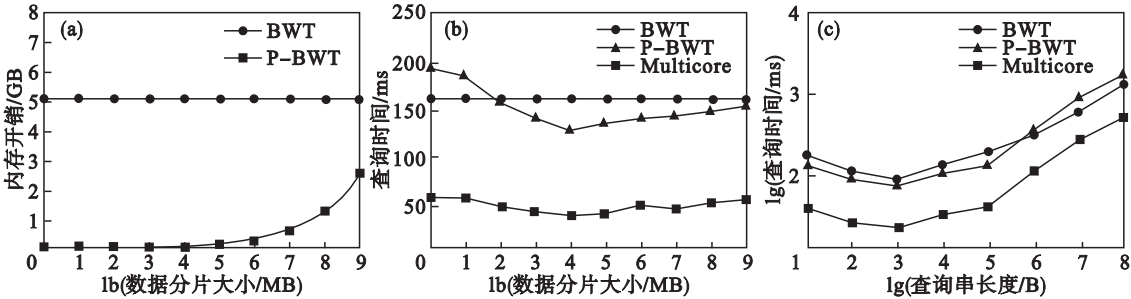


图 5 英文数据集实验结果
Fig. 5 Experiment results of English dataset

(a)—不同数据分片大小下内存开销 ; (b)—不同数据分片大小下查询时间 ; (c)—不同长度的查询串的查询时间.

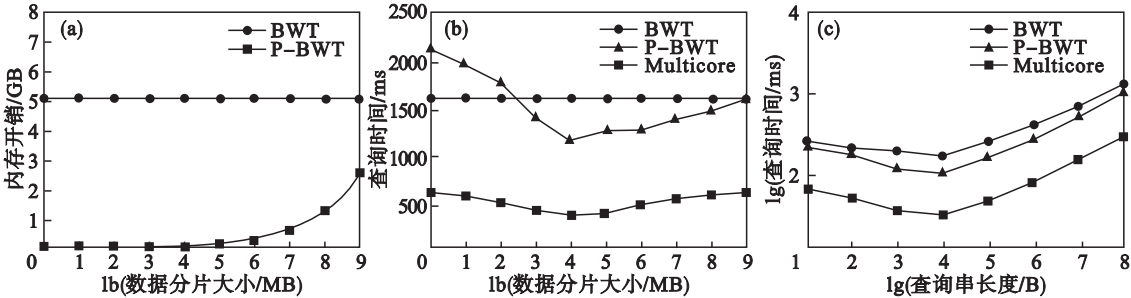


图 6 DNA 数据集实验结果
Fig. 6 Experiment results of DNA dataset

(a)—不同数据分片大小下内存开销 ; (b)—不同数据分片大小下查询时间 ; (c)—不同长度的查询串的查询时间.

5 结 论

1) 本文提出了在有内存限制环境下利用 P - BWT索引进行长串匹配的方法. 该方法可以支持任意长度的精确子串查询.

2) 分析了多核环境下的并行匹配策略 , 并在此基础上 , 提出了一个多核并行子串精确匹配算法. 该算法既保持了原方法的空间开销小的优势 , 又发挥了多核优势 , 提高了原方法的查询效率.

3) 在真实数据集上对所提出的算法进行了实验测试 , 验证了本文提出的算法在空间和时间上的高效性.

参考文献 :

[1] Navarro G ,Mäkinen V. Compressed full-text indexes[J]. *ACM Computing Surveys (CSUR)* 2007 ,39(1) 2 - 50.
[2] Yang X ,Wang B ,Li C ,et al. Efficient direct search on compressed genomic data[C]//*IEEE 29th International Conference on Data Engineering (ICDE)*. Brisbane ,2013 : 961 - 972.
[3] Li H ,Durbin R. Fast and accurate short read alignment with Burrows-Wheeler transform[J]. *Bioinformatics* ,2009 ,25 (14) :1754 - 1760.

[4] Li R ,Yu C ,Li Y ,et al. SOAP2 :an improved ultrafast tool for short read alignment[J]. *Bioinformatics* ,2009 ,25(15) : 1966 - 1967.
[5] Wandelt S ,Deng D ,Gerdjikov S ,et al. State-of-the-art in string similarity search and join[J]. *ACM SIGMOD Record* , 2014 ,43(1) 64 - 76.
[6] Jiang Y ,Li G ,Feng J ,et al. String similarity joins :an experimental evaluation[J]. *Proceedings of the VLDB Endowment* 2014 ,7(8) 625 - 636.
[7] Ferragina P ,Manzini G. Opportunistic data structures with applications[C]//*Proceedings 41st Annual Symposium on Foundations of Computer Science*. Redondo ,2000 :390 - 398.
[8] Manzini G ,Ferragina P. Engineering a lightweight suffix array construction algorithm[J]. *Algorithmica* 2004 ,40(1) : 33 - 50.
[9] Crauser A ,Ferragina P. A theoretical and experimental study on the construction of suffix arrays in external memory[J]. *Algorithmica* 2002 ,32(1) 1 - 35.
[10] Dementiev R ,Kärkkäinen J ,Mehnert J ,et al. Better external memory suffix array construction [J]. *Journal of Experimental Algorithmics* 2008 ,12(1) 3 - 27.
[11] Wang J ,Yang X ,Wang B ,et al. Memory-aware BWT by segmenting sequences to support subsequence search[C]// *Asia-Pacific Web Conference*. Kunming 2012 73 - 84.