分类号 TP311.1
UDC 004.6
学校代码 10590
密 级 公开

深圳大学
SHENZHEN UNIVERSITY

硕士学位论文

并行度量空间相似性索引MVP-tree的设计和实现

申请人姓名 雷富立
申请人学号 2120230503
专业名称 软件工程
导师姓名及职称 毛睿 副教授

二〇一五年 五 月 九 日

原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律后果由本人承担。

论文作者签名：[signature] 日期：2015 年 5 月 9 日

学位论文版权使用授权书

本学位论文作者完全了解深圳大学关于收集、保存、使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属深圳大学。学校有权保留学位论文并向国家主管部门或其他机构送交论文的电子版和纸质版，允许论文被查阅和借阅。本人授权深圳大学可以将学位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、汇编学位论文。

（涉密学位论文在解密后适用本授权书）

论文作者签名：[signature] 导师签名：[signature]

日期：2015 年 5 月 9 日 日期：2015 年 5 月 10 日

分类号 TP311.1　　学校代码 10590
UDC 004.6　　密　　级 公开

深圳大学硕士学位论文

并行度量空间相似性索引 MVP-tree 的设计和实现

学位申请人姓名　雷 富 立
专　业　名　称　软件工程
学院（系、所）　计算机与软件学院
指导教师姓名　毛 睿

摘 要

相似性查询广泛存在于多媒体和生物数据管理分析中。当面对不同领域数据多样性问题时，传统做法为每种数据类型量身定做专门的索引系统，性能较好但是开发维护的成本高，适用范围窄。度量空间索引把数据抽象成度量空间的点，利用用户定义距离函数的三角不等性实现数据排除。它不要求数据具有坐标，距离函数也不限于欧氏距离，具备高度的通用性，可以利用度量空间索引技术开发一个统一的维护成本低且使用范围广的索引系统来处理多种类型的数据。但是，度量空间索引系统追求通用性而无法利用领域信息，导致其搜索性能逊色于传统的专用系统。针对这一问题，本论文采用并行的方法提高目前最流行的度量空间索引之一 MVP-tree 的性能，主要包括以下几方面的工作：

首先，设计和实现了一种并行化构建 MVP-tree 索引的方法，在多核计算机系统上为索引分支的构建过程分配单独的构建线程，加快索引的构建速度；

其次，设计和实现了多种 MVP-tree 索引多线程并行查询算法。为 MVP-tree 设计了多种查询线程创建和分配的方式，多种缓存机制和多种并行化方式。通过实验分析比较在不同参数设定前提下多种并行查询算法对响应时间和吞吐率的改善效果，得出与各并行查询算法相对应的优化参数搭配方式；

第三，实现了基于 MVP-tree 的并行分布式局部索引架构，实现了数据分块并分散存储以及索引构建和查询的多机并行化，并与单个计算节点上多线程并行方法相结合进一步提高 MVP-tree 索引的性能；

最后，根据以上工作实现了开源软件包 UMAD 的相似性索引模块 GeDBIT，结合其它模块构建了一个索引测试平台，并验证了本论文提出的多种并行方法的有效性。

关键词：度量空间索引；相似性；MVP-tree；多线程；局部索引

Abstract

Similarity indexing plays an important role in multimedia and biological data management. Confronting the challenge of "variety", traditional way is to provide customized solutions to each data types. Although these customized solutions generally performs well, it is very costly to build and maintain them. Metric-space indexing abstracts data of various types into metric space, and takes use of the triangle inequality of the distance function of data to acquire pruning. Since it requires neither the data to have coordinates, nor the distance function to be Euclidean, metric-space indexing forms a universal solution, which is widely applicable and cost effective. However, lacking of domain information, metric-space indexing is generally slower than customized solutions. In this thesis, we try to accelerate MVP-tree, a popular metric-space index, by redesigning and implementing it in parallel. Our contributions include:

(1) A parallel bulkload of MVP-tree is designed and implemented with multiple threads.

(2) A parallel query system of MVP-tree is designed and implemented with multiple threads. Mechanisms of thread allocation, index caching, and intra- or inter- query parallel processing are proposed to improve the responding time and throughput. In addition, good combinations of parameters are experimentally determined for various circumstances.

(3) A local index framework for parallel and distributed similarity indexing with MVP-tree is implemented, which is also integrated with multiple threads MVP-tree on each computing node for even better performance.

(4) All the above are implemented in the context of the UMAD project, in particular its indexing module, GeDBIT. A test platform is built. Experimental results show that the speed of MVP-tree is substantially improved.

Key word： metric space index；similarity；MVP-tree；multiple threads；local index

目　录

图表目录

第1章 绪言

1.1 研究目的和意义

基于内容的相似性搜索(Similarity query)是一种重要的信息检索类型，广泛存在于数据库和数据挖掘应用中。随着多媒体技术的发展和推广普及，基于复杂数据对象(空间数据、文本、图像、音频、视频、时空序列等)的海量数据库不断涌现，相似性搜索已经成为多媒体信息系统基于内容搜索的基本需求，其性能已经成为衡量多媒体系统查询功能的重要指标[2]，而在近年来蓬勃发展的计算生物学研究任务中，相似性搜索所占比例高达 35%[3]。传统的相似性搜索是用多维索引（multi-dimensional indexing）技术实现的。其基本思想是提取数据对象的特征向量并将其映射到向量空间，然后用L族距离计算其相似性。例如针对一维数据（多维数据的一种）的索引技术的研究成果已经很成熟而且丰富多样，如二叉树排序树、平衡二叉树、B-树[4]、B+树[5]、RB-tree[6]树以及各种排序查找方法，虽然这些成果的最初目的是实现快速的精确查找，但是可以不加修改的应用于相似性搜索领域的研究。相应的在多维与高维数据的索引技术领域已有的研究成果有多种,并且同样可以用于相似性搜索领域，如 Grid-File[7]、K-D tree[8]、R-tree[9]以及应对高维数据的方法，如空间填充曲线[10]，VA-file[11]等，这些方法广泛应用于以地理信息系统为代表的领域，取得了较好的效果。但是，随着新的数据类型层出不穷，向量空间索引的局限性也越来越明显地表现出来：

(1) 数据对象必须以特征向量来表示。

(2) 数据间的匹配必须用包括欧几里德距离在内的 L 族距离或其简单变形来衡量

越来越多的数据类型不能满足以上两个条件。例如，用于图像类型的颜色直方图等的距离函数因为存在关联而不能用欧几里德距离来表示[8]，基因或者蛋白质序列的相似性基本是用海明距离(Hamming distance)或者加权编辑距离(weighted edit distance)来计算 [9]。

度量空间索引（metric space indexing）是一种适用性非常广的解决相似性搜索的通用方法[14-17]。它把复杂的数据对象抽象成度量空间中的点，利用用户定义的距离函数

的三角不等性来去除无关数据并减少直接距离计算的次数，以实现高速搜索，因此度量空间索引也被称为是基于距离的索引（Distance-based Indexing）[15]。度量空间索引只要求用户提供满足度量空间性质的距离函数，而距离函数的具体实现和数据的表达都是透明的，同样的算法可以应用于不同的数据，因而具备了更广泛的适用范围，他既可以代替已经存在的索引技术研究提高一维数据空间，多维数据空间和高维空间数据搜索性能的方法，而且也可以应用于传统多维坐标数据无法表示但是存在度量任何两个数据点的距离方法的复杂数据对象空间，如文本、图像、音频、视频、时空序列等。

我们现实生活中日常所用的管理系统经历了从专用系统到通用系统的演变过程。专用系统是指为每种数据类型量身定做专门的索引系统，因此需要为多个数据类型开发多个索引处理系统，性能较好但是开发维护的成本高，适用范围窄。而通用系统是指利用度量空间索引等通用数据索引技术开发一个统一的系统来处理多种类型的数据，性能较差，但是具有较低的开发维护成本和较宽广的适用范围。商业软件为了追求最大的利润，一般都是采用通用系统的方法，通过高性价比来吸引用户。最近十几年，制造计算机成本越来越低，而多核计算机系统发展迅速并且日趋成熟，很多商品化的多核计算机已经问世，因而利用多核计算机以及由多台机器组成的多机系统的处理和存储资源优势成为克服通用系统性能瓶颈的重要途径。因而，本文研究通过利用多核计算机和多机系统的资源优势提高基于 MVP-tree 索引的通用相似性检索系统性能有重大意义。

1.2 相似性搜索

相似性搜索是在数据库的数据集中查找相似对象的操作，如文本匹配，基于内容的图像检索和 DNA 基因序列匹配与查找。数据对象之间的相似度通常用与数据对象类型相应的距离函数 d 计算出来的距离值来表示。通常相似性搜索的结果可以用一个集合 $\{x_i|\mathrm{d}(x_i,q)<r,x_i\in S\}$（其中，S为数据集，q为用户指定的匹配对象，r为指定相似度也可为称为查找范围值）来表示。相似性搜索的形式化数学定义如下：

给定元组(S,d,Q,q)

1. S为包含 N 个数据对象的数据集合
2. d为计算S中数据对象之间距离的函数
3. Q一个查询请求的封装，q为封装于Q中的查询参考对象。Q中除了查询请求外，

针对不同的查询类型封装不同的查询信息，比如范围查找的查找半径r

常见的相似性搜索类型主要有以下几种[14, 15]

1. 范围查找R(q,r)：从数据对象集合S中检索出所有满足$d(q,x) \le r$的数据对象x。

2. KNN（K-Nearest Neightbor Query）查找，在S中查找出与q对象距离最小的前K个对象的集合，当K为 1 时，KNN 查找同时也是最近邻查找。

3. DkNN 查找，为基于距离的 KNN 查找，相对于 KNN 查找，它不关心属于距离q且距离最小的前k个但是距离范围值大于r的对象。

1.3 度量空间相似性搜索

度量空间索引通常也称为是一种基于距离的索引，是一种通用的相似性搜索索引方法。度量空间可以看成一个非空数据对象集合 X 和度量函数 d 构成的二元组（X，d）。其中度量函数可以计算出数据对象集合 X 中的任意两个对象的距离，且距离函数满足以下条件[18]：

（1）$d(x,y) \ge 0$,并且$d(x,y) = 0$当且仅当$x = y$;

所有的距离必须是大于等于0，只有当两个数据对象是相同的数据对象时，这两个数据对象距离才为0；

（2）$d(x,y) = d(y,x)$;

数据对象之间的距离有对称性；

（3）$d(x,y) \le d(x,z) + d(z,y)$;

数据对象之间的距离满足三角不等性，任何两个对象之间的距离都不大于这两个对象与第三个对象的距离之和。

数据对象之间的相似度通过距离函数d的值的大小来确定。在查找过程中，给定一个查找对象q，在X中找出与对象q相似度为r（与对象q距离为r）的所有对象，经三角不等性排除不相似对象后得到的结果集可以表示为$\{x_i|(d(x_i,q) \le r, x_i \in X\}$。

1.4 度量空间相似性索引 MVP-tree

度量空间相似性搜索技术利用数据对象抽象和距离函数的三角不等性将数据映射到度量空间并且在查询过程中实现无关数据的快速排除，相对于遍历所有数据有性能优

势，而且可以在所有数据类型的相似搜索问题中通用，因而也称为通用相似性索引。度量空间相似性索引包括优势点树[19-24]，覆盖半径树[25-32]，超平面树[1, 20, 33-37]，以及一些使用商业数据库中已经存在的索引结构实现的度量空间索引[5, 38-41]。其中，优势点树范畴中的 MVP-tree 是最流行且性能较好的一种[22]，所以本文所并行化的通用相似性索引结构为 MVP-tree。

MVP-tree 在每一层结点中存储大于等于 1 个支撑点，每个支撑点划分的分块数目大于等于 2，索引内部结点和叶子结点中都存储了预先计算好的距离值，可以用来在索引查询过程中节省大量因计算距离而消耗的时间。图 1.1 为 MVP-tree 索引结点的结构[20, 21]。

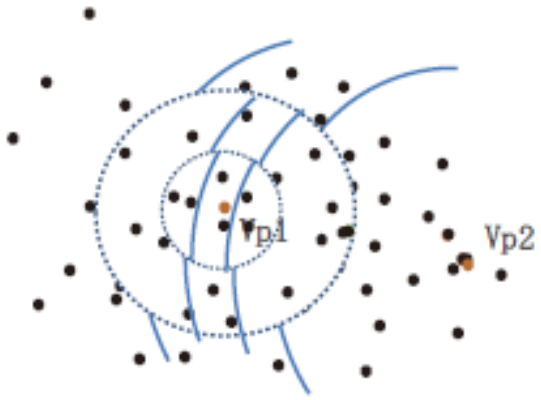

图 1.1 MVP-tree 索引结点结构
Figure 1.1 Node structure of MVP-tree

图中展示的结点包含两个支撑点Vp_1和Vp_2，每个支撑点将对应的数据块划分为三个子分块。根据第一个支撑点Vp_1将原始数据划分为三块：X_1、X_2、X_3，然后用第二个支撑点将上面的每一个分块再次划分为三块，最后得到九个子分块，分别为X_{11}、X_{12}、X_{13}、X_{21}、X_{22}、X_{23}、X_{31}、X_{32}、X_{33}，每一个子分块对应一个子树，图中的结点孩子数目为九个。所以一个 MVP-tree 索引的构建过程可以总结为下面三步：

（1）从原始数据集合中通过一定的方法找出一个或者多个数据对象作为支撑点；

（2）假设选出来的支撑点有三个，并且分别为Vp_1，Vp_2，Vp_3. 首先用Vp_1将原始的数据集划分为m个子集，其中每一个子集中的对象距离Vp_1的距离的最大值与最小值不同；其次再用选出来的第二个支撑点Vp_2将上一步划分的每一个子数据集合划分为m个，每个子数据集的性质与第一次划分后的子数据集的性质相同，即每个数据子集中的对象距离本次参考的支撑点的距离的最大值和最小值不同；最后用第三个支撑点Vp_3将

第二次划分后得到的每一个子数据集合划分为m个子数据集合。最终我们用三个支撑点将原始数据集合划分成m^3个子数据集合，本节点对应的子树也为m^3个；

（3）然后将划分后的子数据集合封装成为一个 MVP-tree 内部节点或者叶子节点。MVP-tree 索引的所有节点都包含本节点选出来的支撑点信息，但除了支撑点，在内部包含指向孩子节点的指针，每个孩子节点对应的数据分区的边界值，而在叶子节点中包含数据对象列表和列表中数据对象距离支撑点的距离，根据这些预先计算好的距离可以在查询过程中排除对部分子树的查询操作，很好的提升查询效率。图 1.3 展示了所有的查询范围与索引结构划分范围的交叉情况，图中R是已经计算好的距离值存储在结点内部，距离q大于R的数据对象在虚线圆形范围外，假设圆内范围的表示左孩子C_l，圆外范围表示C_r。

不同的覆盖情况，执行的查询操作有所不同，如图所示。

（1）$\mathrm{d}(\mathrm{vp}, q_1) \leq R - r$，　　对于$q_1$的查找只需要查找$C_l$；

（2）$\mathrm{R} - \mathrm{r} \leq \mathrm{d}(\mathrm{vp}, q_2) \leq R + r$，　对于$q_2$的查询左右子树都需要查找；

（3）$\mathrm{d}(\mathrm{vp}, \mathrm{q3}) > \mathrm{R} + \mathrm{r}$，　　对$q_3$的查找只需要查找$C_r$；

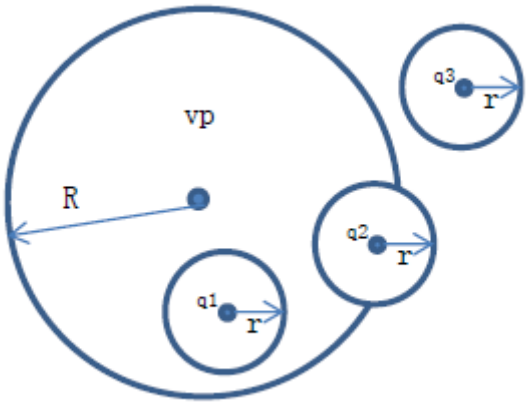

图 1.2 MVP-tree 查询排除条件

Figure 1.2 Pruning condition of MVP-tree

1.5 局部索引架构和全局索引架构

现存的分布式索引架构主要分为：全局索引架构（Global Indexing）[42]和局部索引架构（Local Indexing）[42]。局部索引架构如图 1.3 所示，在对数据构建索引或者进行其它处理之前，先将大数据集划分成多个数据子集，然后再针对每个数据子集单独建立

索引，在查询过程中查询请求由前端服务节点简单的发送到每个计算服务节点上，然后计算服务节点将各自的结果返回给前端节点，前端节点进行结果归并然后返回给用户；

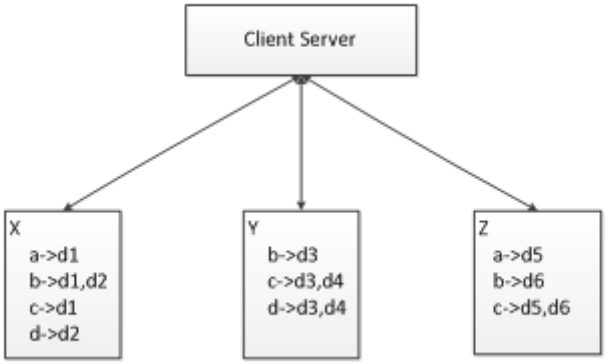

图 1.3 局部分布式索引架构[42]

Figure 1.3 Architecture of local index

全局索引架构如图 1.4 所示，在建立索引时面向的对象是整个原始数据集而不关心数据集的初始大小，最终所建立的索引是针对原始完整数据集的索引，当原始数据非常大时，最终形成的索引规模很大，所以在构建索引之后，按照一定的规则将所形成的完整索引切分成多个小索引片段然后分散存储于多台计算服务节点上，在查询过程中按照查询的请求信息定位索引片所在的机器，然后发送查询请求到目标节点上进行处理，目标计算服务节点将结果返回后需要对结果进行求交集等进一步处理后反馈给用户。

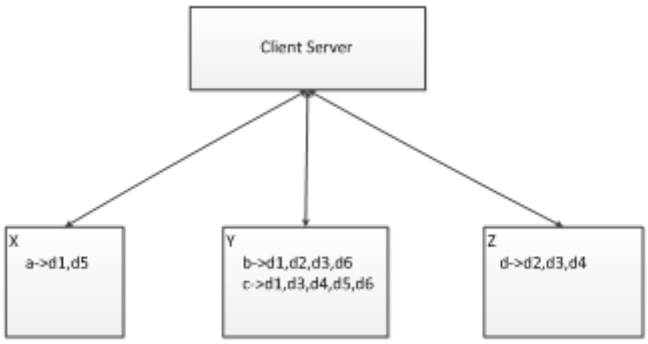

图 1.4 全局分布式索引架构[42]

Figure 1.4 Architecture of global index

1.6 并行度量空间索引研究现状

目前，对度量空间索引并行化方法研究主要集中于 CR-tree[29-32, 43]和 GH-tree[1, 36, 37]

系列，并行化方向主要包括基于单台多核计算机系统的并行化方法和基于多台计算机组成的集群的多机并行化方法。目前，对于 MPV-tree 索引的研究主要集中于通过支撑点选择和数据划分来提高索引的性能[27, 44, 45]，而对其并行化的研究工作还没有发现。

1.6.1 基于多核计算机平台度量空间索引并行化方法的研究

M-tree 在单台多核计算机系统上的并行化方法是创建与处理器个数相同的任务线程[46]，创建一个被多个线程共享的任务队列，每当有新的查询请求到达时首先插入到共享任务队列中，各个任务线程空闲时互斥访问任务队列取出一个查询然后执行查询操作。每个线程有一个副任务队列用来存储在查询过程中产生的需要查询的子任务。当存在线程的处理任务为空且共享任务队列查询对象个数为零时便被其它的线程安排子任务给该空闲线程，来保持各个线程负载均衡。如果每个线程都在忙碌，那么各子线程只处理属于自己的任务和其产生的子任务。M-tree 的另一种基于单台多核计算机系统的并行化方法是在索引构建时分别采用轮询（round-robin）和近邻（proximity index）方法将索引的节点分布到多个磁盘[36]，在查询过程中首先查找符合的节点信息存储到预取队列，然后多个节点在多个磁盘并行存取来减少 I/O 对查询的影响，这种方法性能的提高依赖于在构建索引过程中节点的分布方法。Jackub 和 Qiu Chu 提出了另外两种在单台多核计算机上并行构建 M-tree 的方法[30]，其主要思想是：根据 M-tree 覆盖范围层次结构来划分子任务并行执行。Jackub 将构建 M-tree 的任务分为三块：分别是划分结点、递归划分、子树指针汇总添加到子树根节点。然后将数据及划分为 p 块，为每一块建一个线程，每个线程执行的任务是：为每一个数据对象找到距离它最近的支撑点，然后将该数据对象归入该支撑点对应的数据集合。在上一步产生的多个数据集合上并行构建 M-tree，这是一个递归过程，子线程在构建子树的时候，父线程等待子线程创建子分支成功后返回父线程子树的根节点，父线程被激活来构建父节点，最终构成一个完整的 M-tree，此方法与本研究内容中静态 MVP-tree 并行构建有重要的借鉴意义。除了上述工作 Jackub 还提出了一个动态构建 M-tree 的方法[29]，其主要思想是采用推迟分裂的思想来保证在建树过程中并行执行的任务不会因为节点分裂需要对节点加锁而减少并发度。

1.6.2 基于多机系统的度量空间索引并行化方法的研究

目前，存在的多机并行 M-tree 的方法主要采用主从（Master-Slave）架构[47-53]和点对点（Peer-to-Peer）架构[1, 36, 47]。

Vlachou 和 Doukeridis 提出的超级-普通对等节点（Peer-to-Peer）M-tree 架构中每个超级节点可以接受查询请求[43, 54]，然后通过其超级索引结构将查询发送到普通对等节点，通过导航索引决定是否将查询发送给其它超级节点，进而其它超级节点将查询操作发送给与其连接的普通对等结点。普通对等结点负责存储各自的数据，并在各自的数据集合上构建索引。每个普通对等结点连接在一个超级对等结点中，超级对等结点连接的普通对等结点数目设定上限值，普通对等结点将其上的索引的根节点发送到与其连接的超级对等结点中。

超级对等结点收集与其连接普通对等结点发送过来的根节点，并利用这些根节点构建自己的超级 M-tree 索引，这样在查询过程中超级节点根据每个根节点信息可以判断该普通结点是否包含目标对象，如果包含，那么将查询发送到该普通对等结点执行查询操作。每个超级对等结点与其它的超级对等结点连接，连接的上限设定为一个固定值。在超级对等结点构建导航索引结构来支撑超级对等结点组成的网络，进而支持快速的相似性查找。每个超级结点将自己的 M-tree 索引根节点发送给与其连接的超级节点，接收到 M-tree 索引根节点信息后将所有的根节点信息构建成一个导航索引，导航索引可以在查询过程中决定是否将查询操作发送给该超级节点。收到 M-tree 索引根节点信息的超级节点将该信息再次送给其它与其连接的超级节点。

除 M-tree 外，其它度量空间相似性索引结构如 GHT 的研究相对较少,张兆功采用的是 Master-Slave 全局索引架构来构建分布式的 GHT[37]，Batko 提出两种点对点（peer-to-peer）分布式架构的 GHT 索引结构[1, 36],其中一种如图 1.5 所示.

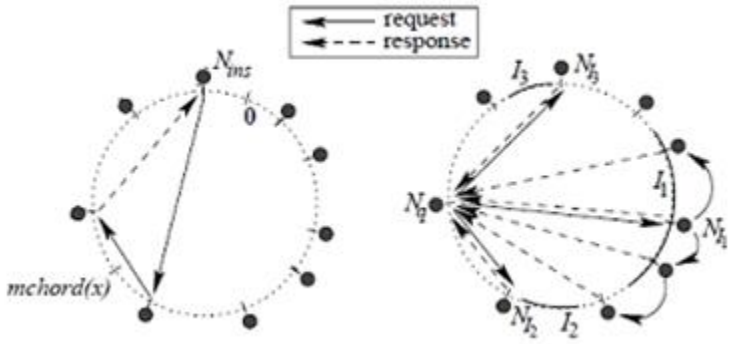

图 1.5 点对点分布式 GHT 索引架[1]

Figure 1.5 Architecture of peer-to-peer GHT

在一致性哈希点对点（Peer-to-Peer）分布式架构 GHT[1]中，首先将度量空间的数据点分割成几块，在每一个数据块中选一个支撑点，其它点依据其相对于他所在集合的支撑点的距离排序，将一个结合的对象映射到一致性哈希架构的一个区间，一个区间存储到一个哈希路径的一个处理节点上。查询过程中，在每一个节点中判断是否另一个节点包含结果，如果发现另一个结点包含需要的结果，那么将查询请求发送到该结点上，该节点启动查询操作产生出结果集，如果发现查询范围超过了本机节点的区间范围，将查询发送到前驱或者后继节点继续查询，将结果返回给发出查询的结点并通知该节点等待前驱或者后继节点的结果集合。

1.7 本论文主要工作

首先，实现一个可以支持相似性查询操作的通用数据管理和分析系统 UMAD。其次，设计和实现了 MVP-tree 索引在多核计算机平台上的索引构建和查询的并行化。第三，实现了一个基于 MVP-tree 的并行分布式局部索引架构的相似性检索系统，具体内容如下：

1.7.1 通用数据管理和分析系统的实现

完成一个通用数据管理和分析系统 UMAD 开源软件包的编码实现和测试，系统由 C++开发并发布于 GitHub 开源网站上。通过在 UMAD 中实现相似性索引模块（GeDBIT），结合其它模块如数据对象模块、距离模块、支撑点选择模块等，共同完成本论文提出的 MVP-tree 多线程并行构建和查询算法的性能测试。

UMAD（Universal Management and Analysis of Data）是一个通用的数据管理和分析系统，同时也是一个度量空间数据管理分析原型系统，其主要功能是将原始数据对象抽象成度量空间中的点，然后在度量空间中用统一的方法实现针对不同类型数据的聚类、分类、异常点监测和相似性检索等应用和研究。如图 1.6 所示，系统包括：数据类型(type)、距离函数(dist)、支撑点选择(pivot selection)三个基础模块和索引(indexing)、分类(classification)、聚类(clustering)、异常点检测(Outlier detection）四个功能模块：

(1) 索引模块（indexing）包含两个部分：索引构建和查找。通过索引构建模块提供的接口，用户可以构建基于不同数据类型的 MVP-tree 索引结构，而通过查找部分提供的接口可以在所构建的 MVP-tree 索引进行查找操作；

(2) 分类模块（classification）包含多种在度量空间数据集合中训练出来的分类模型，这些模型可以用来作为在度量空间新数据集上做分类操作和研究

(3) 聚类模块（clustering）包含针对度量空间中数据的四种聚类算法；

(4) 异常点监测模块（Outlier detection）包括多种离群点定义和检测算法的实现

(5) 数据类型（type）将多种数据类型如向量，文本，图像等原始数据封装成为度量空间数据对象，并且提供了读写文件的接口；

(6) 距离函数模块（dist）包含与多种数据类型相应的距离函数的定义，用于在 UMAD 各个功能模块执行过程中计算对象之间的距离；

(7) 支撑点选择（pivot selection）包含了多种常见的支撑点选择算法，是 UMAD 中各个功能模块的基础算法库。

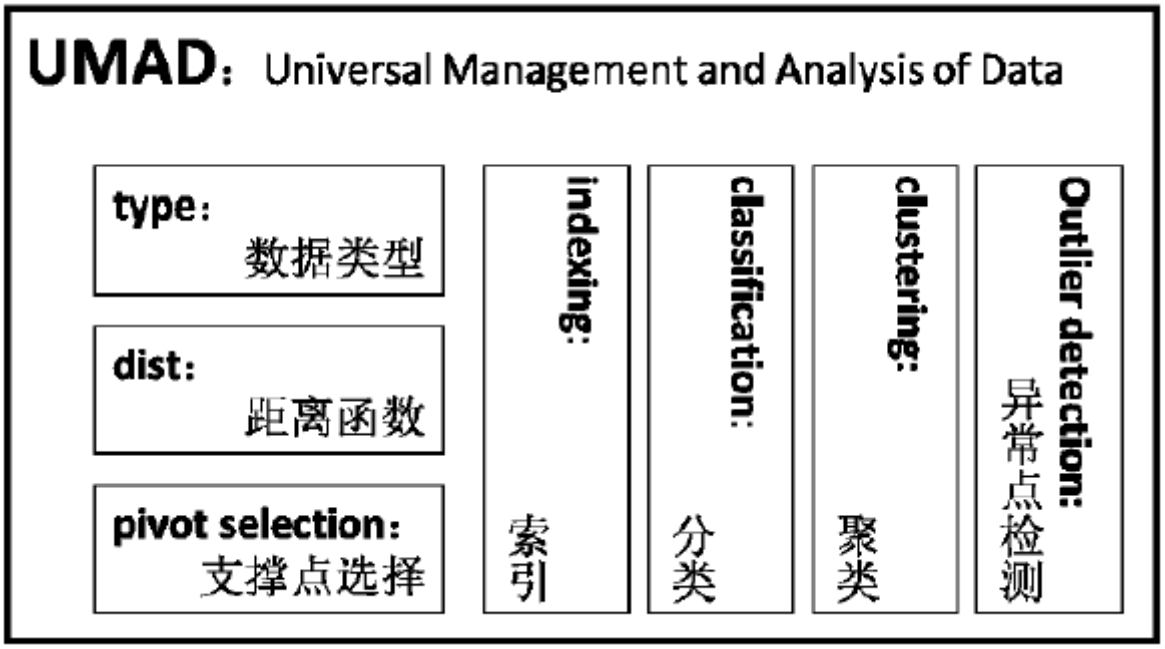

图 1.6 UMAD 系统架构

Figure 1.6 Architecture of UMAD

在 UMAD 系统中，索引模块一个基于度量空间相似性索引 MVP-tree 的通用相似性搜索模块，同时也称为 GeDBIT(Generalized Distance-Based Index Tree)，可以支持多种数据类型如向量、数字图像、蛋白质/DNA 序列和文本序列的相似性查找操作和研究。UMAD 系统的用户可以通过继承数据类型模块中度量空间数据类型和距离函数模块中度量空间函数类型，为自己的特有数据类型以及相应的距离函数定义子类，就可以调用 UMAD 的功能模块完成相应的数据管理和分析操作和研究。

1.7.2 多线程并行化 MVP-tree 的设计和实现

分析 MVP-tree 索引结构特点和索引构建过程，找出各个节点及其子节点之间的关系，将整个索引构建任务分解成多个可独立执行任务并由多个线程并行处理，加快索引构建的速度；把减少响应时间和提高吞吐量作为出发点，设计了多种查询线程创建和分配的方式，多种缓存机制和多种并行化方式，找出查询过程中主要消耗处理器时间的操作，然后分解成多个独立任务并利用多线程并行执行，加快查询处理速度；最后，在实验过程中调整并行线程数目等参数，通过分析比较各个方法的响应时间，吞吐率，为各个并行方法确定在 UMAD 的相似性检索模块 GeDBIT 使用过程中，使其达到最佳性能时的参数值与各个参数的搭配方式。

1.7.3 并行分布式 MVP-tree 的设计和实现

在由多台独立计算机组建的机群上实现一个基于 MVP-tree 的并行分布式局部索引架构的相似性检索系统，将用于构建索引的原始数据划分为多个分块并存储于机群中每一个计算节点上。然后创建客户端，通过客户端将索引构建和检索请求发送到机群中的计算节点上，各个计算节点执行各自接收到的请求，结合各个计算节点的多线程并行化算法进一步加快相似性检索系统的性能。

1.8 主要创新点

目前，对于度量空间索引并行化的研究主要集中于覆盖半径树和超平面树，而对于支撑点树范畴 MVP-tree 的研究主要集中于通过寻找最佳的支持点选择和数据划分方法来改善索引性能，而本论文研究主要是用并行化的方法改善 MVP-tree 索引构建和检索性能，其创新性主要体现在以下几个方面：

第一 针对单台计算机系统上串行 MVP-tree 索引构建的时间性能问题，设计和实现了一种多线程并行构建算法；

第二 设计和实现了多种 MVP-tree 多线程并行检索算法。设计多种并行查询线程的创建和分配方式，并为 MVP-tree 查询过程设计和实现一种结点缓冲方法提高 MVP-tree 查询的整体性能，通过实验测试和分析比较各个算法的性能发现各个算法性能随着参数变化而变化的规律；

1.9 本文结构安排

本文内容主要分为五章，其中本研究的内容介绍分布在第二章至第四章，论文结构安排如下：第一章 绪论，概括讲述了本论文的科研背景和意义，分别介绍了相似性搜索，度量空间相似性搜索以及 MVP-tree 索引结构等。重点介绍了本论文研究内容的研究现状，工作内容和创新点；第二章 UMAD 系统和 GeDBIT，讲述了本论文研究开发的相似性搜索系统，作为后续 MVP-tree 并行算法性能评估的测试平台；第三章介绍了基于单台多核计算机系统的 MVP-tree 索引并行构建和检索算法、结点缓冲方法、查询之间并行化算法、查询内部并行化算法以及查询之间和查询内部同步并行化算法，并介绍了

各个算法的实验测试结果和分析；第四章 介绍了多机并行 MVP-tree 索引相似性检索系统的设计和实现，并通过实验测试比较了相对于单台多核计算机系统上 MVP-tree 并行方法和串行索引构建和检索的性能。第五章 讲述了本论文研究的工作总结和取得的效果，并对本论文未来的工作提出几条路线和建议

第2章 多核计算机平台 MVP-tree 并行化的设计和实现

2.1 引言

度量空间索引把数据抽象成度量空间的点，利用用户定义距离函数的三角不等性实现数据排除。它不要求数据具有坐标，距离函数也不限于欧氏距离，具备高度的通用性，可以利用度量空间索引技术开发一个统一的维护成本低且使用范围广的系统来处理多种类型的数据，但度量空间索引系统追求通用性导致其搜索性能逊色于传统的专用系统。最近十几年，多核计算机系统发展迅速并且日趋成熟，很多商品化的多核计算机已经问世。多核计算机拥有普通计算机无法比拟的数据处理系能，这使得通过将相似性检索任务移植到多核计算机上成为克服性能瓶颈的重要途径。本章中包含的内容为 MVP-tree 索引在单台多核计算机平台上的并行化问题，关于 MVP-tree 索引在由多台计算组成的集群上的并行化问题将在第四章中将讲述。

2.2 MVP-tree 构建多线程并行化

在 MVP-tree 构建过程需要在数据块中进行支撑点选择和数据划分，其中包含大量的距离计算和排序操作，本小节内容为利用多个线程并行执行多个分支的构建任务方法加快索引的构建速度。

2.2.1 MVP-tree 构建并行化分析

MVP-tree 索引构建的过程如第 1.4 小节中的介绍的那样主要包含两步：第一步为支撑点选择：通过特定的方法从原始数据集合中选出特定数目的数据对象作为支撑点，然后从原始数据集合中将支撑点删除；第二步为用选出来的支撑点计算出各个支撑点与数据集合中其余数据对象之间的距离，根据计算的距离值大小将原始数据集划分为一定数目的子数据集，用于递归构建子树。在这两步索引构建过程中，数据对象构建操作和距离比较操作是两个主要的处理资源消耗的部分，将所有的这些计算与比较操作划分到多

个任务中并行的执行，可以有效的加快索引的构建操作。

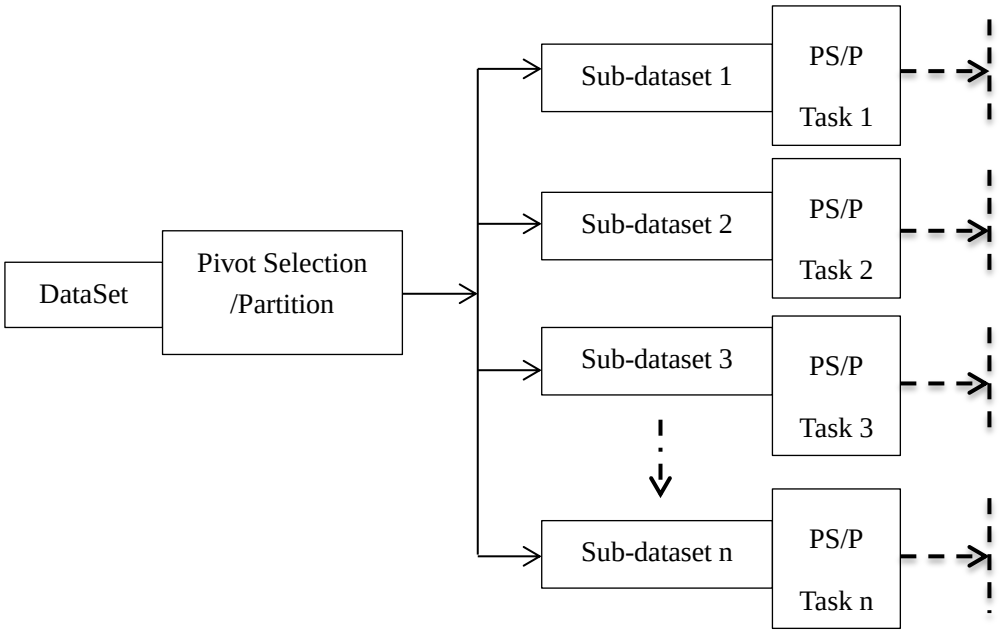

图 2.1 MVP-tree 索引构建任务分解

Figure 2.1 Partition of MVP-tree building task

MVP-tree 索引构建的思想是将整个构建任务划分为多个子任务，然后各个击破，并行化方法是将这些子任务分布到不同的构建线程中并行执行，如图 3.1 所示。

```
//parallel selecting pivot and partition the data set
void* parallelTask (void *vpb)
{
    ParallelBlock *pb = (parallelBlock*) vpb;
                ... ...
    BTask task=pb.taskList.back();
                ... ...
    int *pivotAddress=pivotSelectionMethod (task);
                ... ...
    list<list<object>*> partitionList=partition (task);
                ... ...
    for(int i=0;i<partitionList.size();i++)
    {
            BTask *task=new Task (partitionList[@i]);
            pb->taskList.push_back (task);
    }
}
```

```
//initialize the parallel task
Void parallelBuildIndexStructureInitialization
      (list<list<Object>> objList, Metric *metric)
{
    BTask task (&objList);
    List<BTask> taskList;
    taskList.push_back (task);
    ParallelBlock pb (taskList, metric, index...);
          ... ...
    while (pb.taskList.size>0)
    {
       pthread_t pid = 0;
       pthread_create (&pid,  paralelTask, (void*)&pb);
    }
          ... ...
}
```

图 2.2 并行构建 MVP-tree 索引

Figure 2.2 Building MVP-tree with multiple threads

图 3.2 为并行构建 MVP-tree 索引的主要思路的伪代码，主要包含两个方面的并行：

（1）支撑点选择：有多种方法可以用来选择支撑点，比如 FFT, Incremental, PCA 等。在所有的这些选择方法中，包含候选点与数据集合中其余数据对象之间的距离计算和比较，所以我们可以给每一个节点的支撑点选择划分操作到不同的索引构建线程中并行执行；

（2）数据划分：首先需要计算数据集合中每个对象与支撑点的距离，然后根据支撑点的距离值信息，将原始数据集中的对象按照距离值大小排序，所以我们给不同节点的数据划分操作一个执行任务，所有这些任务并行执行；

2.2.2 并行构建算法性能测试

图 3.3 至图 3.6 为 MVP-tree 索引串行构建性能和并行构建性能的比较。本实验的数据分别为 20 位的浮点向量，实验过程中数据的大小由 1 万到 10 万,10 万到 100 万变化；蛋白质数据 50 万，实验过程中数据由 1 万—10 万—50 万变化；DNA 序列 100 万，实验过程中数据大小由 1 万到 10 万,10 万到 100 万变化；文本序列 6 万条，实验过程中数据大小由 1 千到 1 万,1 万到 6 万变化。实验中的索引结构中，我们设定的支撑点个数为 2，单个支撑点划分的数据块最大数目为 3，单个内部结点最大的分支数目为 9。并行化方法是为根节点的每一个孩子分支创建一个构建任务，所有任务之间并行执行。下面四幅图为实验结果，结果中的比较对象为串行构建索引的总时间和并行化构建索引所需的总时间。

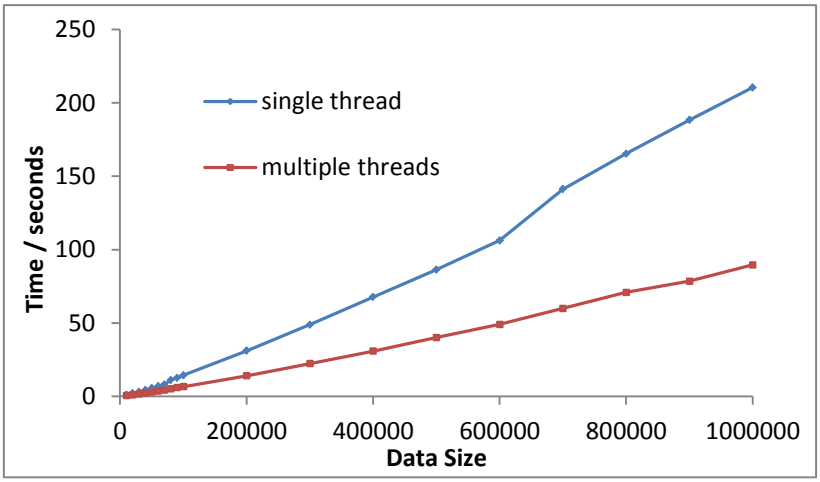

图 2.3 向量数据索引构建性能

Figure 2.3 Build performance on vector

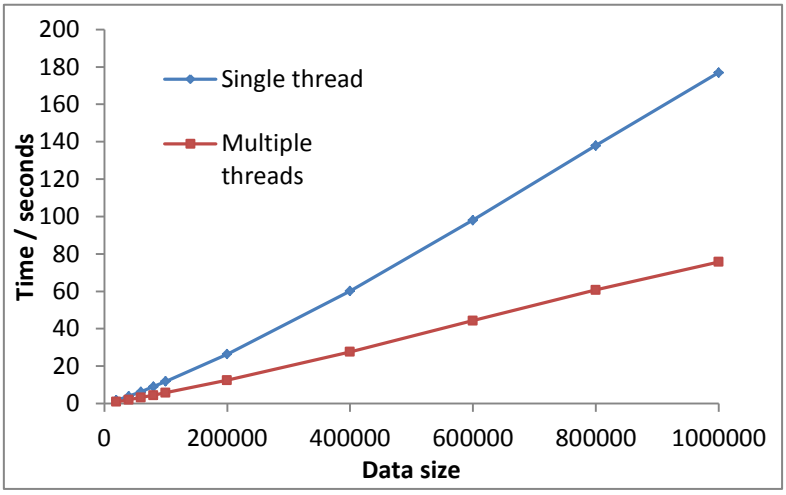

图 2.4 DNA 序列索引构建性能

Figure 2.4 Build performance on DNA

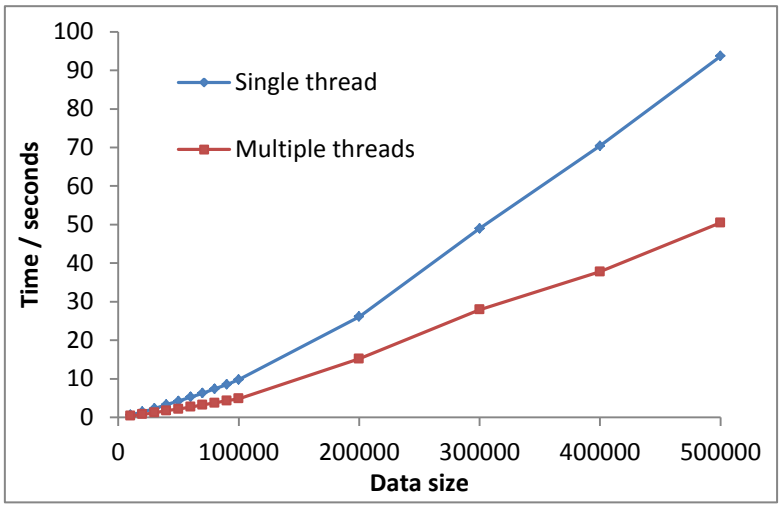

图 2.5 蛋白质数据索引构建性能

Figure 2.5 Build performance on peptide

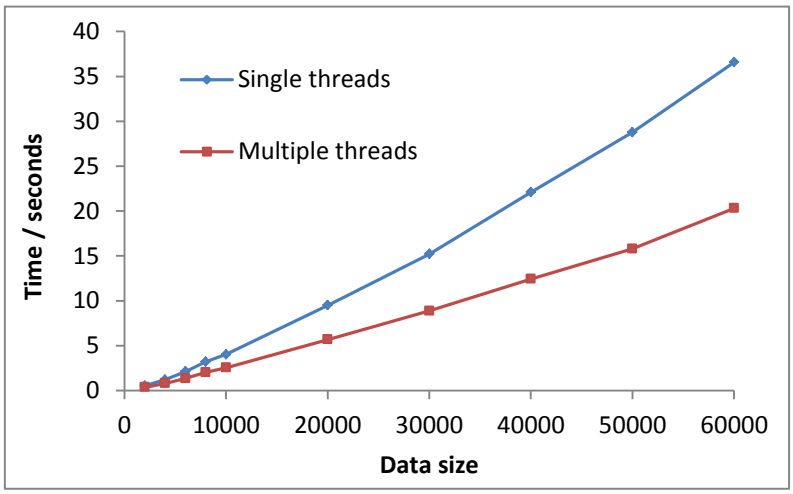

图 2.6 文本数据索引构建性能

Figure 2.6 Build performance on string

图 3.3 到图 3.6 展示的各个数据类型实验结果可以看出，在原始数据量小的情况下整个索引需要的时间少，由于构建和撤销多个线程消耗了一定的时间，导致并行索引构建算法加速比小，当数据量增大时，整体索引构建所需要的时间比较大，而此时创建和撤销多个线程的时间可以忽略不计，所以并行构建索引的性能越来越明显。总体所有数据类型并行化构建索引的性能快于串行构建最多 2.4 倍是因为构建根节点的时间占用整个索引构建的时间的三分之一，所以 2.4 倍是一个理想的并行化效果。

2.3 MVP-tree 查询系统的缓冲机制

在索引构建完成后索引被写入到文件，当在系统上执行查询任务的时候，判断结点每个子分支是否与查询请求的查询范围有交集，当确定有交集时，首先将结点从磁盘中加载到内存中，然后再计算确认是否属于结果集。当没有缓冲时，GeDBIT 在处理查询请求过程中需要停止等待从磁盘中读取一个一个目标节点信息，读磁盘占用了查询过程大部分时间。创建缓冲的另一个目的是为本论文后续的 MVP-tree 索引检索系统的并行化做准备，在没有缓冲时，即便使用了并行化方法，多个并行执行线程会相互竞争磁盘读取索引信息，而底层磁盘是一个一个的响应磁盘读取请求，这样极大的减少了系统的并行化效果。使用缓冲后，多个查询从缓冲获取结点信息，并且可以不加任何互斥机制的访问结点并计算，这样可以极大的提高多个查询执行线程之间的并行化。

2.3.1 MVP-tree 缓冲映射方法

索引在构建过程中将完成的索引结构写入了磁盘，在查询过程中获取节点的方法是通过文件相对偏移量读取索引文件。为了将文件中的父子结点关系信息保存下来进而保存为一个完整的树结构，并且能够快速的判断结点是否已经加载在内存中，在 GeDBIT 系统中采用内了映射的方法，映射的键值为结点所在文件中的文件偏移量，通过判断文件偏移量值是否在映射中来判断结点是否已经在内存中。在 GeDBIT 系统中用户可以通过参数设定索引缓冲的高度，本论文采用了两种缓冲方式：

(1) 缓冲所有节点(Buffer all nodes - BAN)：通过参数设定要缓冲的 MVP-tree 的高度，在处理查询请求之前，将所有的结点加载到内存中；

(2) 只缓冲叶子结点(Buffer leaf nodes only - BLNO)：内存的缓冲中只加载

MVP-tree 叶子节点，在查询过程中访问内部结点时，需要从磁盘中读取。

下面 3.3.3 小节测试并分析对比了二者的性能以及缓冲对查询性能的影响。

2.3.2 缓冲性能测试

系统缓冲是在查询过程中将一部分索引的结点预先加载到内存的容器中，在查询过程中判断结点是否在内存中，如果在内存中那么就用内存中的结点信息，如果结点不在内存，然后才去读磁盘。本小节实验测试的平台为 1000000 数据量的 MVP-tree 索引系统，每条数据为 20 维的浮点数向量，索引结构中内部结点包含的支撑点个数为 2，单个支撑点对应的数据分块为 3，每个内部结点的孩子结点最大个数为 9。

（ 1 ）本实验的查找请求个数由 10 到 100，100 到 1000 变化。图 3.7 中的曲线为只缓存叶子结点（BLNO）和缓冲所有节点(BAN)查询时间的比较，从曲线的变化幅度上可以看出二者的查询性能相差小于 0.1，这是因为每个索引中内部结点的扇出大，索引的整体高度低，基于 1000000 向量数据集构建的索引中，内部结点数目 7381，叶子结点数目为 59049，叶子结点占整个索引结点数目的 88.89%。

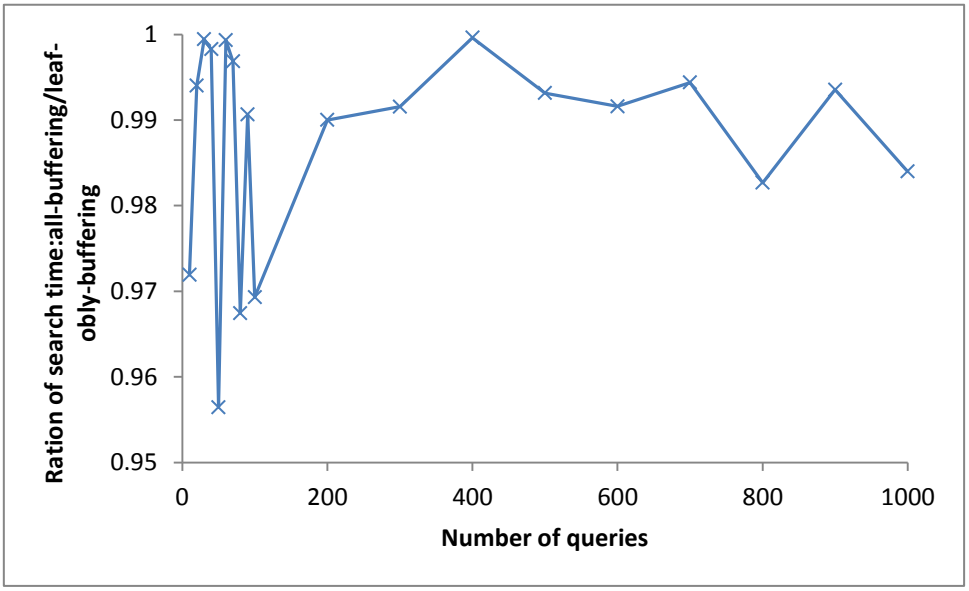

图 2.7 缓冲所有节点与缓冲叶子结点查询时间对比

Figure 2.7 Throughput time of buffering all nodes vs only leaf nodes

在每个内部结点中数据对象个数为支撑点个数 2，那么所有内部结点中数据对象的个数是 14762，而在叶子结点中包含的数据对象有支撑点和数据对象链表两部分，叶子的支撑点数目同样是 2，数据对象列表中的对象个数最大为 100，叶子节点中数据对象

个数为 985274，因而叶子中数据对象个数占据索引中所有数据对象个数的 98.53%。鉴于二者性能相差不大且内部结点中的数据对象数目相对于整体索引结点来说可以忽略，所以在后面的所有实验中都采用缓冲所有节点的方式。

（ 2 ）本实验测试缓冲对串行查询性能的影响。在实验中分别测试记录了使用缓冲查询和不使用缓冲查询的吞吐率时间。图 3.8 中的曲线为二者时间之间的比值（不使用缓冲的时间和使用缓冲的查询时间的比值）。查询请求的个数从 1-10，10-100 增量变化。从图 3.8 可以看出当查询个数在 1-10 增加时，缓冲时查询性能的增加与查询个数的增加成正比，当查询数目大于 10 时，缓冲的加速性能趋于平缓状态。这是因为我们在执行查询请求时首先加载缓冲到内存而且是缓冲所有节点，这样随着缓冲数目的增加，缓冲加载的时间相对于执行完所有查询需要吞吐率时间来说可以忽略，但是总体上在查询数目比较大的情况下，缓冲可以将 GeDBIT 系统检索的性能提高大于 5 倍。

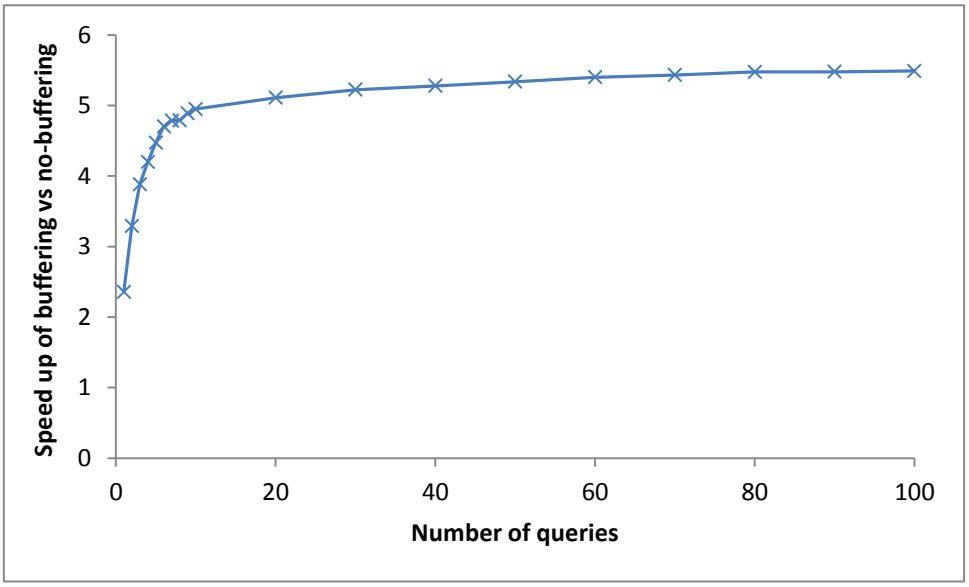

图 2.8 缓冲对串行查询过程的影响

Figure 2.8 Effect of buffer to search process with single thread

（ 3 ）下面这个实验测试了缓冲对多线程并行查询性能影响。在 GeDBIT 系统缓冲中的另一个作用就是在多线程并行查询过程中减少多个并行查询线程之间对磁盘的竞争，使得多个查询线程达到高度的并行化，这样 GeDBIT 处理大批量的查询请求的性能将会大大的提高。图 3.9 展示了实际性能结果。实验平台 GeDBIT 系统中 MVP-tree 索引大小为 1000000，并行化方法为每一个查询创建一个线程。在实验中记录了并行查询方法在不使用缓冲情况下和使用缓冲情况下的吞吐量时间，并用二者的比值作为缓冲对多线程并行情况下的加速比。

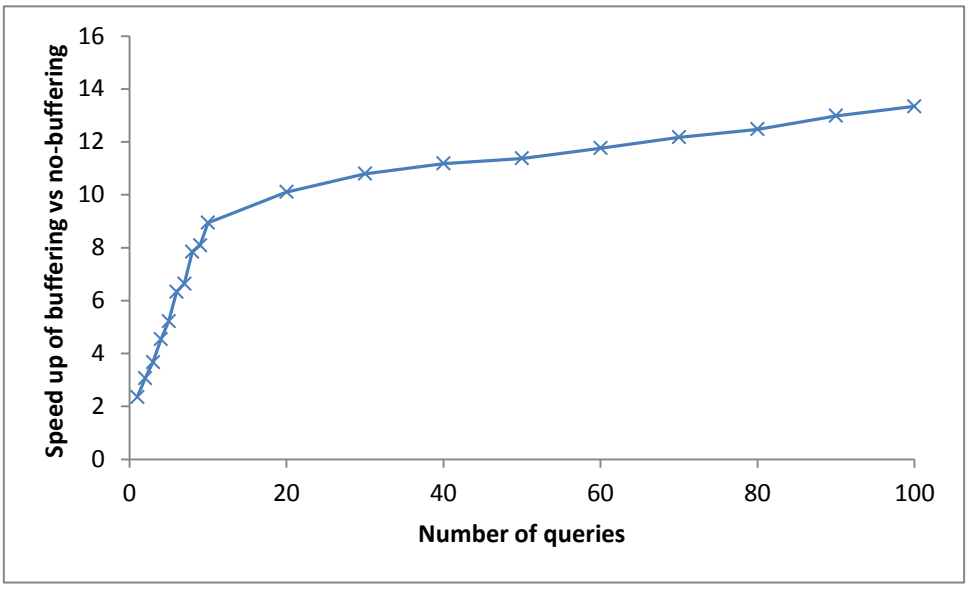

图 2.9 缓冲对多线程并行查询过程的影响

Figure 2.9 Effect of buffer to search process with multiple threads

从图中曲线走势可以看出，与串行查询使用缓冲的加速效果不同，多线程并行查询情况下缓冲的加速性能一直随着查询个数的增加而增加，即便查询数目超过了 10 个。同时我们可以看出多线程并行查询使用缓冲情况下比不用缓冲时的加速比大，这是因为不使用缓冲的多线程并行查询过程中读磁盘请求被单个磁盘驱动器一个一个的响应，这样多线程并行的查询请求退化为串行查询。而在多线程并行查询过程使用缓冲后，省去了读磁盘的时间又实现了查询之间的并行化。

2.4 MVP-tree 并行化提高吞吐量

通常情况下，GeDBIT 系统批量的处理多个查询请求。当原始串行的 GeDBIT 系统部署于单台计算机并且由单个线程处理查询请求时，系统必须一个一个的处理同时到达的多个请求。如果系统部署于具有多个处理器和多个逻辑处理核的计算机平台并且多个查询请求可以被多个查询线程同时处理时，系统的吞吐量可以大幅度的提升。当系统可以创建多个查询线程处理查询请求时，即便 GeDBIT 部署于具有单个处理核的计算机上，仍然可以提高系统吞吐量,因为其中一部分查询线程可以在CPU上被轮转调度进行计算，而其它的一部分任务则等待磁盘数据读取返回，这样整个系统处理所有查询请求的时间很明显的减少了。

2.4.1 MVP-tree 查询请求之间并行化方法

为了提高以 GeDBIT 为代表的 MVP-tree 索引相似性检索系统的吞吐率下面提出三种方法。这三种方法的总体思想如图 3.10 所示，给每一个查询请求一个查询线程，在查询请求中包括用户给定的查询对象和要查询的范围值，凭借多处理器或者多逻辑处理核平台的处理资源优势，多个查询线程并行执行实现查询请求之间的并行处理。下面的三种方法减少了索引检索系统的吞吐量，但是三种方法的效率不同。三种方法的思路如下：

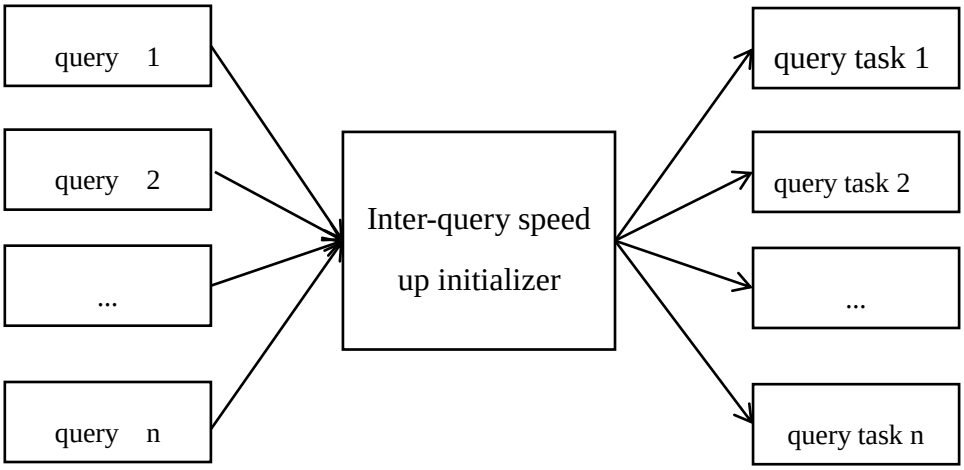

图 2.10 MVP-tree 查询请求之间的并行化

Figure 2.10 Inter-query process speed up

（1） 单个查询一个线程（One Thread Per Query - OTPQ）:这种方法简单的为索引检索系统接收到的每一个查询请求创建一个任务线程而完全不考虑同时到达的查询请求的数目，如果某一时间的查询请求数目为 10000 个，那么将会创建 10000 个线程，线程的数目始终与查询请求的数目相同。图 3.11 是本思路的一个伪代码展示

```
Void multiThreadsSearchInitializer (queryList)
{
   Thread tList [querynum];
   Iterator ite= queryList.begin ();
   while (ite! =queryList.end ())
   {
      CreateThread (&tList[i], queryTask, *ite);
      i++;
   }
```

```
      for (i<threadList.size ())
      {
          join (threadList[i]);
          output (queryList[i]. result);
      }
   }
```

图 2.11 每个查询对应一个查询线程

Figure 2.11 Inter-query speed up method: OTPQ

（2） 最大线程数（Maximum Query Thread - MQT）:同第一种方法不同，在这种方法中限制了检索系统中针对查询请求建立的最大线程数目为 N。当查询的数目小于 N 的时候创建与查询请求同样多数目的线程；当查询请求大于等于 N 的时候，首先为前 N 个查询请求创建 N 个线程，当这 N 个线程处理完查询结束后，在此创建 N 个线程继续处理后续的 N 个查询请求，直到所有的查询请求。我们开发的 GeDBIT 相似性系统的用户可以通过参数来设定最大线程数 N 的值。下面的伪代码展示了本方法的思路。

```
Void multiThreadsSearchInitializer (queryList, tNum)
{
   Thread *threadList=new Thread[tNum];
   while (Iterator ite !=queryList.end ())
   {
      if (counter<tNum)
      {
         CreateThread (&threadList[n], queryTask,*ite);
         n++;
      }
      else
      {
         for(i<tNum)
         {
```

```
                join (threadList[i]);
                output (queryList[i]. result);
            }
            n=0;
        }
    }
    for (i<counter)
    {
        join (threadList[i]);
        output(queryList[i]. result);
    }
}
```

图 2.12 MVP-tree 查询设定最大线程数目

Figure 2.12 Inter-query speed up method: MQT

（3） 静态线程池（Static Thread Pool - STP）:在这种方法中，我们在 GeDBIT 系统上设计实现了一个线程池来处理相似性查询请求。这种方法与 MQT 相似，系统中的最大查询线程数目同样被设定为 N,但是不同的地方是这 N 个线程在启动 GeDBIT 系统后

查询请求到达之前就创建就绪等待查询请求的到来，在整个系统启动到最后系统退出，系统中的查询任务线程始终为 N。开始时线程池中的每个线程睡眠等待，当有查询到来时被唤醒，所有线程互斥访问查询请求队列获取查询请求。图 3.13 简单展示了本方法的总体思路。

```
//intra-query speed up with STTP
Threadpool.start ();
      … …
ResultistmultiThreadsSearchInitializer
                    (Query q, index index)
{
    Resultlist re;
    subTaskList=index.root.search (q);

    ThreadPool.addChildTask (subTaskList);
          this.wait ();

          for (task: subTaskList)
          {
                re.insert (task.resultList);
          }
          return re;
    }
          … …
    Threadpoo.stop ();
```

图 2.13 MVP-tree 查询静态线程池

Figure 2.13 Inter-query speed up method: STP

从第一种方法 OTPQ 的介绍可以看出，系统直接为每一个接收到的查询创建一个查询线程，这种方法在查询数目相对较少的时候性能较好，当同时到达查询数目比较大时，创建过多的线程会耗尽系统的所有处理资源，导致系统负载大，性能减小，因而第二种方法 MQT 中设定的查询线程数目上限是对第一种方法 OTPQ 的一个改进。在这种方法中无论查询数目多少，系统创建的最大线程数目为 N，大于 N 个查询的请求由下一批次的查询线程进行处理。在 OTPQ 和 MQT 这两种方法中有一个共同点：给每一个接受到的查询创建线程，查询处理完后撤销线程，新的查询由新创建的线程处理。这个共同点同时也是这两种方法相对于第三种方法 STP 的共同缺点，不停线程创建和撤销，尤其是数目巨大时对系统的处理性能影响很大。在 STP 中正好弥补了第一二中方法的缺点，在整个系统启动和结束退出过程中只会创建 N 个线程，这 N 个线程或者睡眠等待查询请求到来或者进行处理查询请求，所有的查询请求均由线程池中的这 N 个线程处理。因此，通过理论分析可以得出在这三种方法中 STP 的系统吞吐量最佳。用户在使用 GeDBIT 系统可以设定第二种方法的最大线程数目和静态线程池中的线程数目 N。

2.4.2 查询请求之间并行化方法性能测试

（1）下面这个实验我们在 GeDBIT 系统的 MVP-tree 索引检索过程中实现 3.4.2 中的三种并行化检索方法，并比较了三种方法相对于串行查询方法的吞吐量，最终通过效率比对验证 3.4.2 中的分析结论。本实验在 GeDBIT 系统平台完成测试，测试的 MVP-tree 索引内部结点扇出为 9，索引构建的数据类型为浮点向量，数据量大小为 1000000.

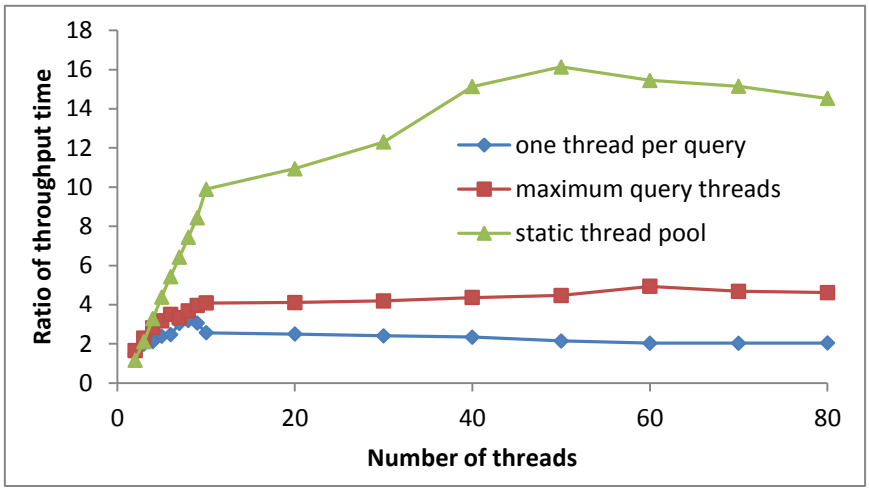

图 2.14 TPQ、MQT、STP 三种并行化方法加速比

Figure 2.14 Throughput time of single thread vs OTPQ, MQT, STP

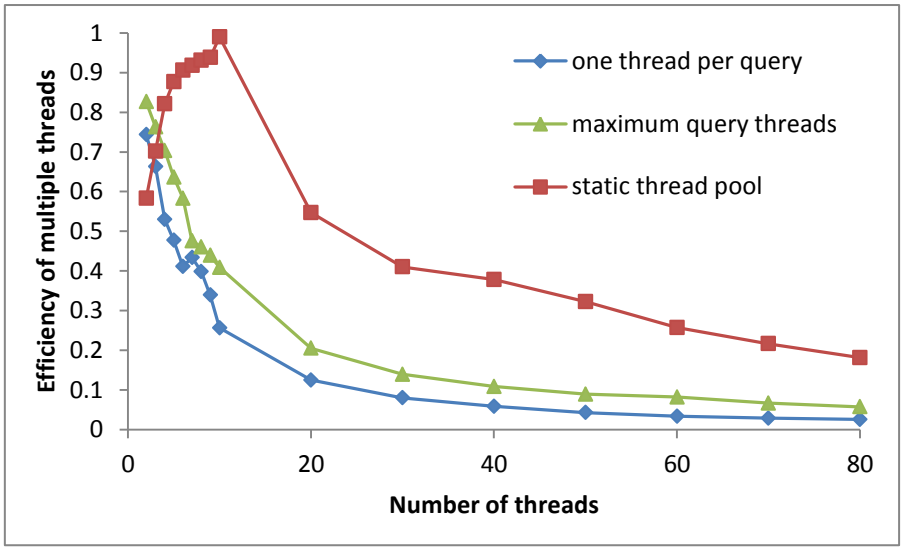

图 2.15 Efficiency of OTPQ, MQT, STP

Figure 2.15 TPQ、MQT、STP 三种并行化方法效率

图 3.14 为三种并行化方法在使用缓冲的情况与串行查询使用缓冲情况下吞吐量时间的比较。在实验中同时提交给 GeDBIT 系统 1000 个查询请求，并分别记录了三种方法在线程数目从 2-10,10-80 变化的情况下完成 1000 个查询的总时间。图 3.15 为三种方

法的效率，即三种方法各自相对于串行的加速比与线程数目的比值。从图 3.14 我们可以看出当线程数目为 8 时，OTPQ 达到了加速比最大值，但是仅仅是 3.2 倍；而当线程数目为 53 时，MQT 方法的最大加速比为 5.0，这种方法略好与第一种方法，但是相对于创建的线程数目来说，加速效果很小，而第三种方法当静态线程池的线程数目为 31 时，并行方法的加速效果可以达到 16 倍。在图 3.14 中当线程池线程数目为 10 时，计算机的效率可以达到 0.98，远远超过了其它两种方法，但是所有三种方法的效率随着线程数目的增加减少，这是因为当创建过多的线程时实验测试平台负载过大，导致整体性能下降。在 3.4.2 已经中分析过，第一种方法 OTPQ 和第二种方法 MQT 方法的共同点为每一个查询请求创建线程是相对于第三种方法静态线程池 STP 来说是致命的弱点，在使用 STP 方法时，系统中的线程数目被限定为一个特定的值，且 GeDBIT 系统始终只批量创建一次线程，然后在系统退出时撤销一次线程，所以吞吐量改善效果最佳，图 3.14 和 3.15 的结果正式验证了这一分析结论。

（2）从上面实验结果可以得出 STP 的效果最好，下面将通过改变提交到系统中的查询数目来测试查询数目的不同对 STP 的加速性能有何影响。结果如图 3.16 所示。

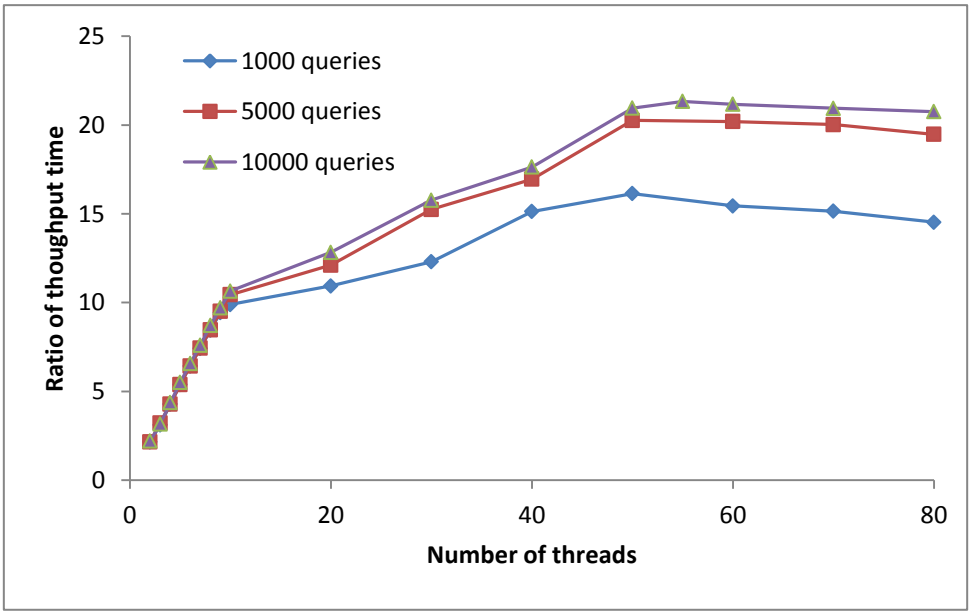

图 2.16 查询个数对静态线程池的影响

Figure 2.16 Influence of query number to the performance of STP

本实验测试的数据类型为 20 维浮点数向量，数据量大小为 1000000，查询半径为 2（可以查找出 MVP-tree 索引中 80%—90%的数据），使用缓冲预先将 MVP-tree 索引结构加载进内存。实验中查询请求的数目分别为 1000、5000、10000，线程池中的线程数目

分别从 2—10—80 变化，实验统计的数据为串行查询的吞吐量时间和线程池中的吞吐量时间的比值作为加速比。从图 3.16 可以看出查询请求数目从 1000 变化到 5000 时，并且加速比效果的比较大，后者远远超过了前者，二者最大加速比之差达到 4.81 倍。当查询请求数目从 5000 变化到 10000 时，加速比的变化不明显，后者的最大加速比仅仅比前者的最大加速比多出 0.4 倍。这是因为当处理的查询请求过多时，多个线程需要大量的互斥操作访问任务队列获取查询任务，线程池并行查询节省的时间相对于处理完成所有查询的总时间可以忽略。

（3）本实验在前面几个实验的基础上测试了线程池和缓冲同时对 GeDBIT 系统的查询请求的影响。而本实验比对的对象是串行查询在不使用缓冲和线程池使用缓冲的情况下的吞吐量时间的比较。在实验过程中使用的线程池线程为 50，索引的大小为 1000000 条浮点向量，维度 20。查询请求的数目从 10—100—1000 变化。

从图 3.17 的加速比中我们可以发现本实验结果的加速值远远大于本小节实验二中的加速比，这是因为线程池不仅比串行查询省去了大量的磁盘读取操作而且还实现了查询请求之间的并行化，大大的节省了整体的查询处理时间。图 3.18 为与图 3.17 相对应的效率图，可以看出效率一直在随着查询请求数目的增加而增加，这是因为线程池中的线程数目被固定为一个最大值，而查询请求的加速比随着查询数目的增加而增加。

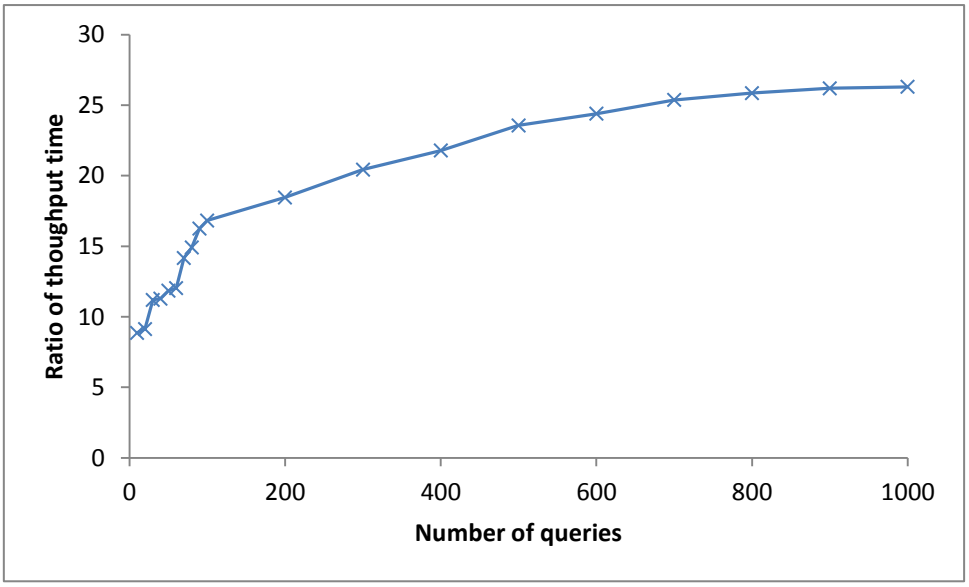

图 2.17 STP 吞吐量时间加速比

Figure 2.17 Speed up of STP vs single thread without buffer

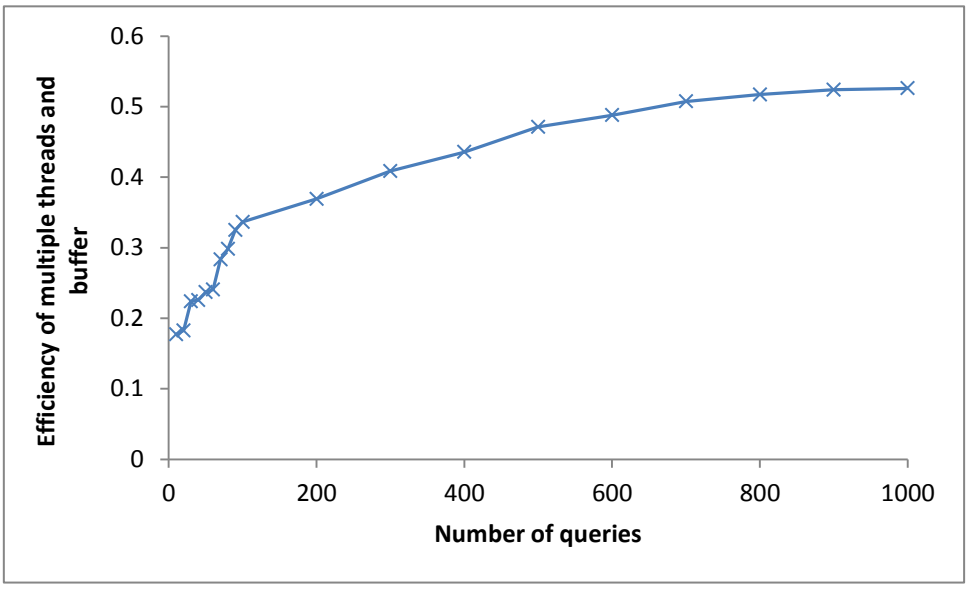

图 2.18 STP 的效率

Figure 2.18 Efficiency of STP relative to single thread without buffer

2.5 MVP-tree 并行化加快响应速度

MVP-tree 为基于距离的索引，由第 1.4 小节介绍可知，在索引构建过程中根据所选出来的支撑点与原始数据集中的其它数据对象距离的距离值来将数据集划分为多个子分块，相应于每一个分块递归构建子树。当我们在使用 GeDBIT 构建 MVP-tree 索引并设定每个结点的支撑点最大数目为p（实际选出来的支撑点可能小于 p）时，每个支撑点划分得到的最大数据块数目为F（随着数据类型和分布不同，实际分块结果不同），那么结点的总分支数目为F^p。当查询请求的查询范围比较大时，需要一个一个的遍历每一个子树，本小节的任务是实现子分支查询之间的并行化，即实现查询过程内部的并行化来加快响应速度。

在 GeDBIT 所采用的查询过程内部并行化方法为给每个根节点每个分支一个查询线程，所有分支的查询任务并行执行，加快查询的响应速度。图 3.19 展示了本小节并行化的思路。

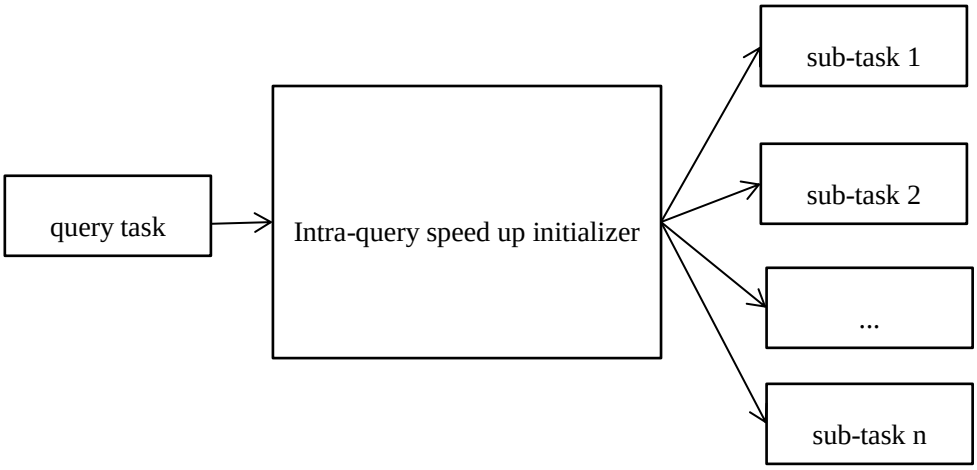

图 2.19 MVP-tree 索引子分支查询并行化
Figure 2.19 Intra-query speed up process

2.5.1 MVP-tree 查询内部并行化方法

为了加快 GeDBIT 系统查询的响应速度，我们在 GeDBIT 系统上设计和实现了如下两种子分支并行化方法：

```
//intra-query speed up with CTP
ResultistmultiThreadsSearchInitializer
                    (Query q, index index)
{
    Resultlist re;
    subTaskList=index.root.search (q);
    Thread tList[subTaskList. size ()];
    for (subtask: subTasktList)
    {
        CreateThread (&tList[i], search, &subtask);
    }
    for (i<threadList.size ())
    {
        join (tList[i]);
        re.inseartresult (subTaskList[@i].result);
    }
    return re;
}
```

图 2.20 MVP-tree 索引每个子分支对应一个线程
Figure 2.20 Intra-query speed up method: CTP

（1） 单个分支一个线程（Creating Thread Promptly - CTP）:在这种方法中，判断根节点的每个分支是否与查询请求的查询范围有交集，如果有交集那么为根节点的这个孩子分支的查询任务创建一个查询线程。我们初步设想为每一个内部结点的查询分支创建一个线程，也就是说查询线程数目等于 MVP-tree 索引的分支数目，但是每个查

询所面向的处理任务仅仅为一个节点，由于查询的任务粒度太小，大部分的线程都会处于等待子任务完成的状态，浪费大量查询时间进行线程创建和内存空间分配，所以只在根节点出进行任务划分，加大并行查询线程的任务粒度。图 3.20 是本方法的伪代码。

（2） 子分支查询间共享线程池（Sharing Thread in Thread Pool - STTP）：同 CTP 不同，本方法是在 GeDBIT 系统中创建一个线程池专门处理所有查询内部的子分支的查询任务。在查询过程中，当确定一个根节点的分支与查询请求的查询范围有交集时，那么将该分支信息封装为一个查询任务提交到线程池的任务队列并唤醒等待任务到来的线程。当一个查询将所有要查询的子分支查询任务提交到线程池后睡眠等待，在最后一个完成的子分支查询任务返回后唤醒一睡眠的父线程，父线程进行结果信息统计和收集然后返回给用户。本方法中的线程池是在 GeDBIT 系统启动后时就预先创建，线程池中的线程数目可以由 GeDBIT 用户通过参数制定，用户提交的所有查询请求均可以通过使用此方法加快查询的响应速度。图 3.21 展示了本方法的思路。

```
//intra-query speed up with STTP
Threadpool.start ();
      … …
ResultistmultiThreadsSearchInitializer
                  (Query q, index index)
{
    Resultlist re;
    subTaskList=index.root.search (q);

    ThreadPool.addChildTask (subTaskList);
          this.wait ();

          for (task: subTaskList)
          {
               re.insert (task.resultList);
          }
          return re;
    }
          … …
    Threadpoo.stop ();
```

图 2.21 MVP-tree 子分支查询共享线程池线程

Figure 2.21 Intra-query speed up method: STTP

在由 CTP 介绍中可知，为到来的每一个查询的根节点分支创建查询线程，当 GeDBIT 用户同时提交的查询请求数目大时，系统都在不停的创建子分支处理线程，虽然此时查询请求的响应加快了，但是为每个查询都创建线程和撤销已完成的线程，从系统的整体吞吐量时间上考虑仍然有改善的余地。而且当所查询的索引小时（索引构建时所基于的原始数据量小），CTP 方法对查询响应速度的改善很小并且有时可能慢于串行查询。从另一方面考虑，MVP-tree 内部结点的扇出随着 GeDBIT 系统用户指定的支撑点数目和与支撑点对应的数据分块数目变化，当扇出比较大时，那么的针对查询创建的线程数目比较

多，创建和撤销线程的时间很可能占用查询时间的很大一部分，这将导致查询多线程并行化的效果适得其反。因而可以初步推断查询请求间共享线程池中的子分支查询处理线程方法对响应速度的改善效果较好。

2.5.2 查询内部并行化方法性能测试

（1）本小节在 GeDBIT 检索系统上实现了两种加快查询响应速度的方法，一个是为根节点的孩子分支直接创建线程，另一个是所有查询共享线程池中的线程避免创建线程和撤销线程的开销。下面通过实验测试二者的效果。在进行本实验过程中，使用的数据为 1000000 条 20 维浮点数向量并且在 GeDBIT 系统实现了内部结点扇出分别为 4、9、16、25、36 的 MVP-tree 索引结构，对应的支撑点数目均为 2，与支撑点对应的分块数目分别为 2、3、4、5、6。查询请求数目为 100，每个查询的查询范围值设为 2。在实验过程中记录了每个查询的响应时间并与串行查询的响应时间作对比，在使用线程池时，线程池中线程的数目与查找操作所进行的 MVP-tree 索引内部结点的数目一致，分别为 4、9、16、25、36，图 3.22 与图 3.23 展示两种方法的加速比，3.24 与 3.25 分别为二者效率走势。

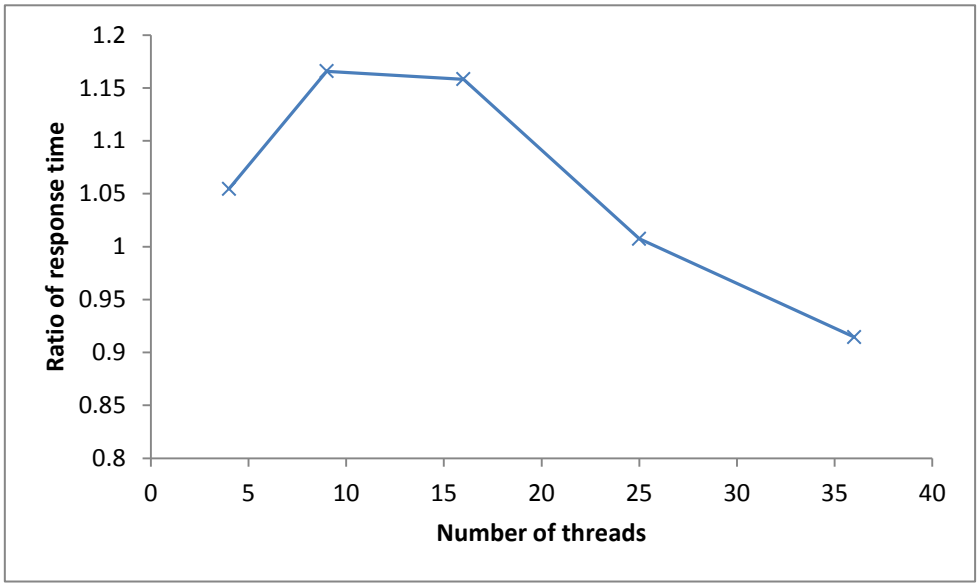

图 2.22 单个子分支对应单个线程加速比

Figure 2.22 Speed up of CTP

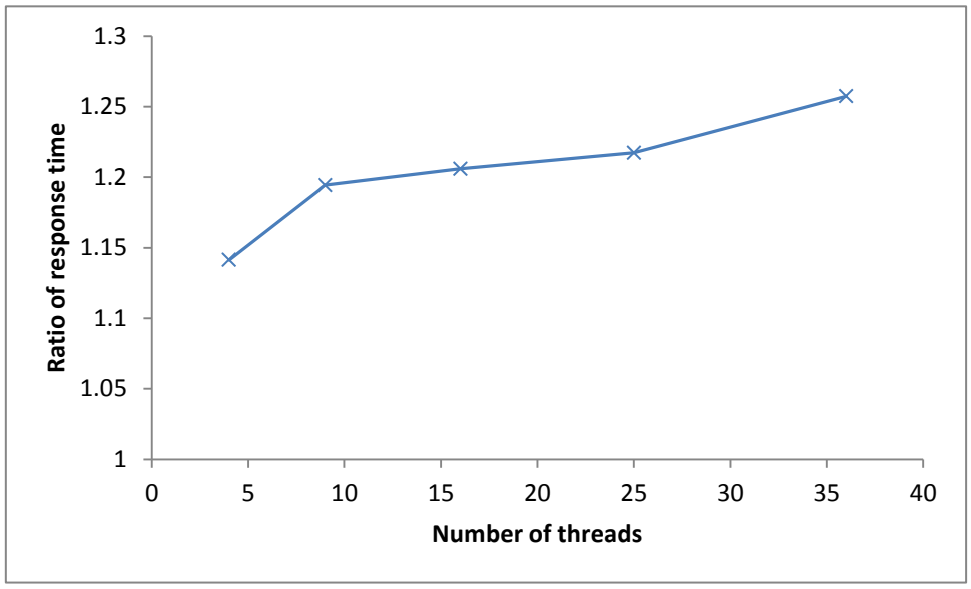

图 2.23 子分支查询共享线程池线程加速比

Figure 2.23 Speed up of STTP

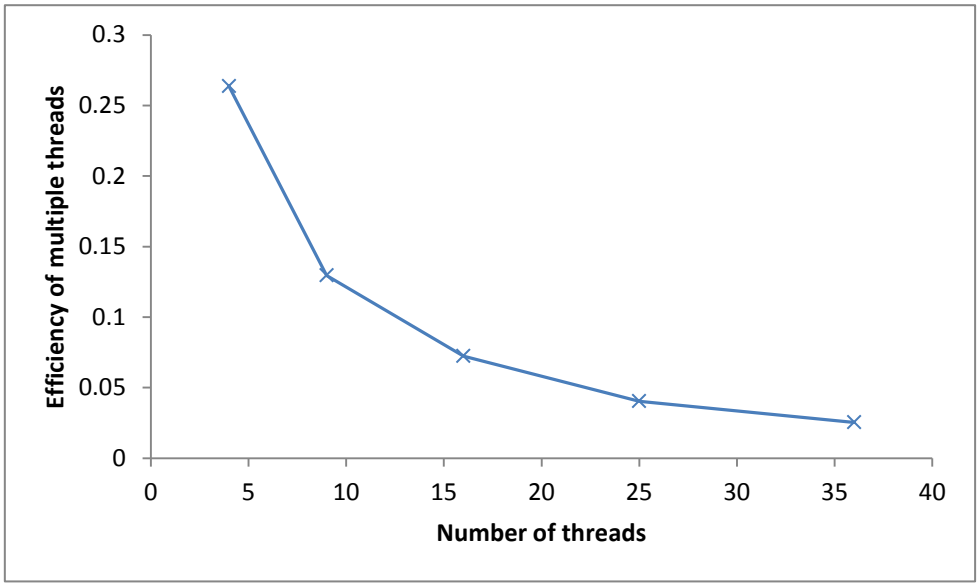

图 2.24 单个子分支对应单个线程效率

Figure 2.24 Efficiency of CTP

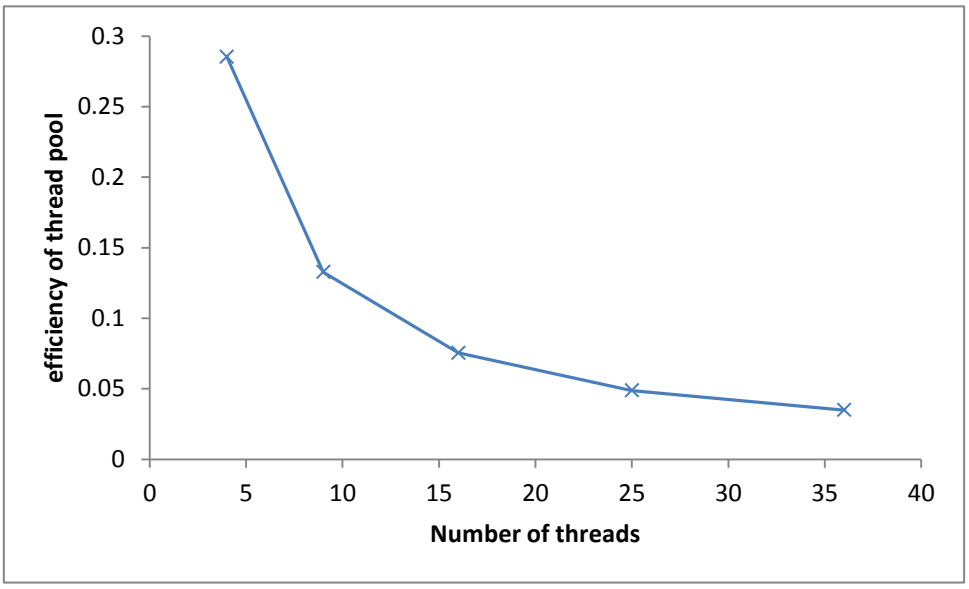

图 2.25 子分支查询共享线程池线程效率

Figure 2.25 Efficiency of STTP

从图 3.22 可以看出,当索引的内部结点扇出小于 16 时,CTP 的响应速度加速比大于 1 表示并行查询响应速度快于串行查询，而当扇出大于 16 时，CTP 的响应时间开始下滑并且慢于串行查询。这是因为索引内部结点扇出比较大时，索引创建子分支查询线程的时间和撤销线程相对于完成查询的响应时间来说比例很大，导致多线程并行化的查询响应速度慢于串行查询，并且可以从图 3.22 看出随着内部结点分支的扇出的增大，并行化查询的响应速度下降。图 3.23 为 STTP 的加速效果，与图 3.22 的结果相比，图 3.23 中内部结点扇出大于 16 时，并行化查询的响应速度随着子分支的增加而增加，这是因为所有的子分支查询任务均有线程池中的线程完成，索引的子分支数目越大，STTP 相对于 CPT 节省的线程创建和撤销时间越多，并且相对于串行查询子分支查询任务的并行化程度越高。图 3.24 与图 3.25 分别是图 3.22 与图 3.23 对应的效率图，从二者的走势可以看出 CPT 与 STTP 的总体加速效果的增加慢于线程数目的增长速度。

（2）基于本小节第一个实验可以看出 STTP 的响应速度加速效果比较好，本实验测试缓冲和 STTP 方法同时对 GeDBIT 索引检索系统的响应速度的改善效果的影响。实验平台 GeDBIT 系统中 MVP-tree 索引内部结点的扇出为 9，数据为 20 维浮点数，数据量为 100 万，索引结构支撑点数目为 2，单个支撑点划分的数据块数目最大为三，所以内部结点的最大数目子分支数目为 9。本实验 STTP 的线程数目为 9，查询个数从 10—100—1000 变化。实验中记录了 STTP 使用缓冲时每个查询的响应时间和不使用缓冲时串行查询的响应时间。

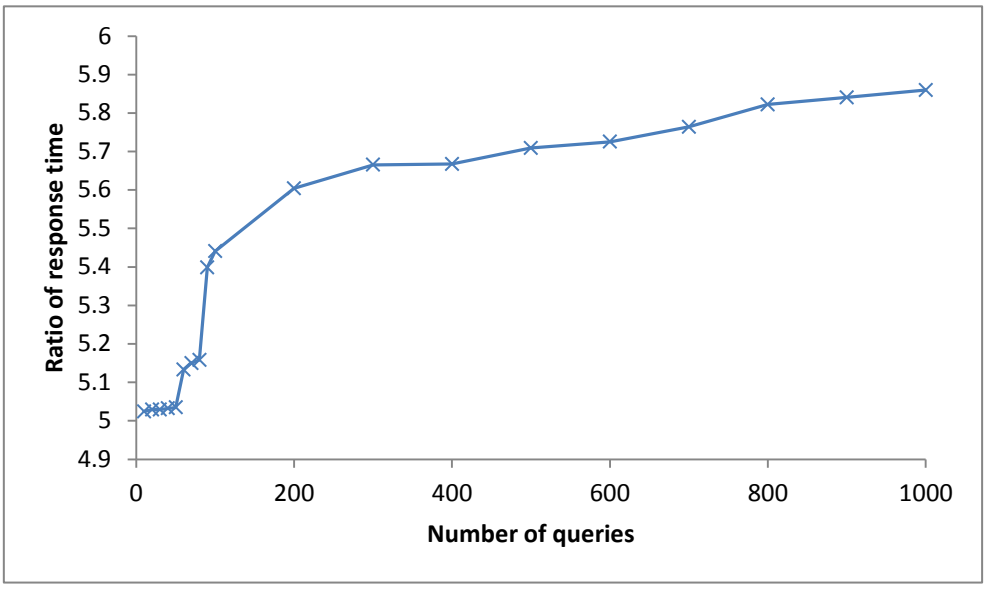

图 2.26 子分支查询共享线程池线程加速比

Figure 2.26 Speed up of STTP vs single thread without buffer

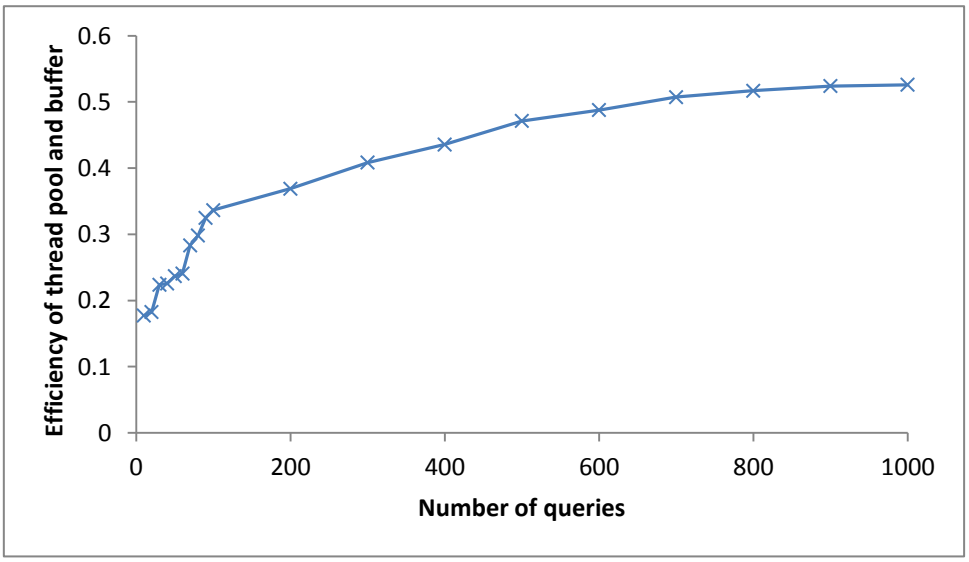

图 2.27 子分支查询共享线程池效率

Figure 2.27 Efficiency of STTP relative to single thread without buffer

图 3.26 展示了响应速度加速比，比较对象是实验中记录的响应时间的比值，图 3.27 对应的效率走势，有加速比值与线程数目相比得出。从图 3.26 可以看出 STTP 与缓冲的响应速度加速效果随着查询数目的增加加快，这是因为随着查询请求的增多，STTP 相对于串行查询的读磁盘和串行处理子分支查询节省的时间越多，最大加速比达到 5.85 倍。由于响应加速效果在增加，STTP 中的线程数目固定为 9，所以图 3.27 中 STTP 与缓冲的效率持续增加，最大效率达到了 52.38%。

2.6 MVP-tree 并行化同时改善吞吐量和响应速度

在本章第四节和第五节当中分别介绍了提高吞吐量和加快响应速度的方法。其中都提到了使用线程池的方法，并从实验结果中可以看出使用线程池的方法的效果最佳。但是这两节中的线程池同属于 GeDBIT 系统却是相互独立的模块，用户不能同时通过第四章中的线程池提高吞吐量的同时使用第五节中的相同方法加快响应速度，本节的主要内容是设计一种新的方法同时提高和加快 GeDBIT 相似性索引系统的吞吐量和响应速度。

2.6.1 MVP-tree 查询内部和查询之间同时并行化方法

本方法是结合了本章第四节和第五节中线程池的优势同时实现了查询之间和查询内部的多个子分支查询之间的并行化。其思想是设计一个线程池，线程池中包含两部分线程，其中一部分线程用于实现查询之间的并行化，另一部分线程用于实现查询过程中子分支查询之间的并行化。两部分线程之间的信息交互通过互斥访问任务队列实现，且这两部分线程同属于一个线程池，所以本方法中的这种线程池称为拥有两部分线程的线程池（Thread pool with two parts of thread - TPT)。图 3.28 为本思路的一个流程展示：

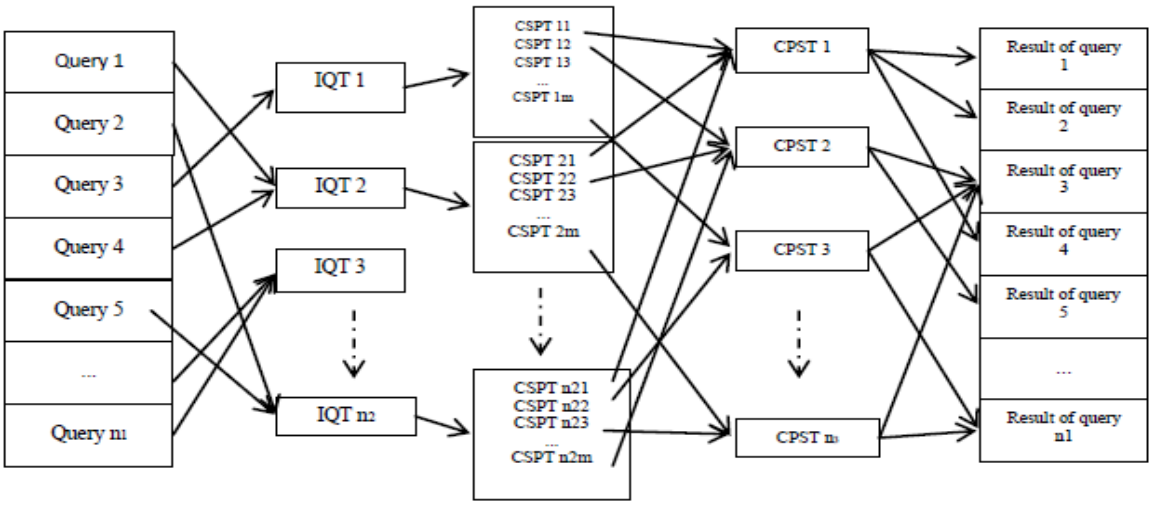

图 2.28 TPT 架构图

Figure 2.28 Architecture of TPT

图 3.28 表示了拥有两部分线程的线程池，两部分线程分别是 IQT（Initial Query Thread） 和 CPST（Child Path Searching Thread)。 当用户提交查询请求时，IQT

即原始查询任务处理线程被唤醒并从用户提交的查询请求列表中提取任务，然后执行查询过程，当确定一个根节点的子分支与本线程提出的查询请求的查找范围相交时，将子分支的信息封装成为一个子查询任务 CSPT（Child Searching Path Task）并提交到子查询任务列表当中，子查询任务提交的同时，CPST 也被唤醒，多个子查询处理线程互斥访问子查询任务列表提取任务。当其中一个 CPST 执行完查询过程并判断该线程执行的任务为某个查询的最后一个子任务时，返回结果并唤醒等待子任务完成的 IQT，IQT 将所有子查询任务的结果信息进行汇总，然后从用户提交的查询列表中重新提取一个新的查询请求重新执行。TPT 同第三节中的 STP 一样限制了整个线程池的线程最大数目，并且从系统启动到最后系统退出线程池中的线程只创建和撤销一次。同本章第四节和第五节不同的地方是本节的线程池即实现了查询请求之间的并行化又实现了查询子分支之间的并行化，因而可以提高 GeDBIT 系统吞吐量的同时也加快了系统的响应速度。

2.6.2 查询请求之间与查询内部同时并行化方法性能测试

本实验测试的 GeDBIT 系统中，MVP-tree 构建于 100 万条 20 维浮点数向量，提交到系统的查询数目为 5000，查询半径为 2，比较的对象中，第一部分是 TPT 方法处理所有查询请求的吞吐量时间和串行查询不使用缓冲时处理查询请求的吞吐量时间的；第二部分为 TPT 方法处理查询请求的响应时间和串行查询不使用缓冲时查询请求的响应时间。下面为实验结果展示。

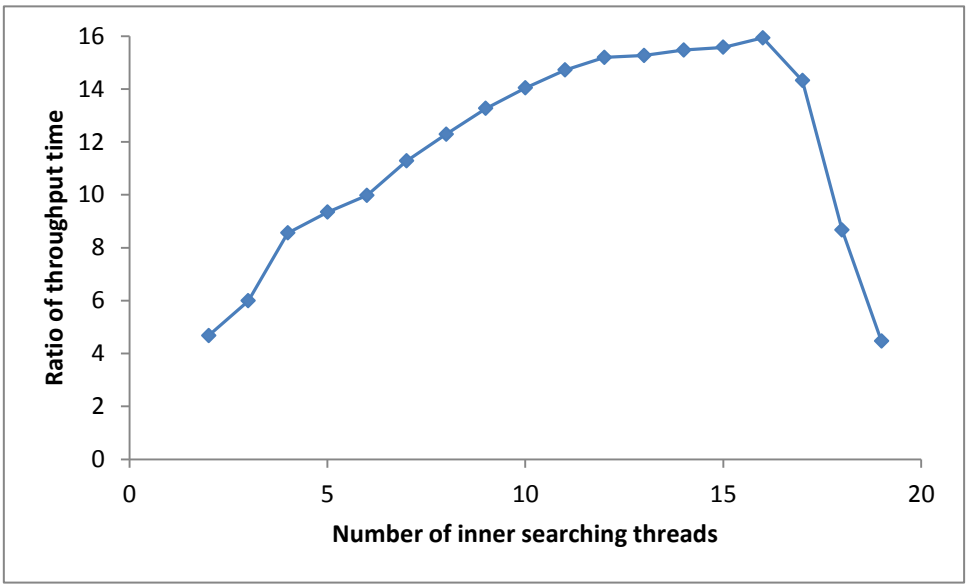

图 2.29 四分支 TPT 吞吐量时间加速比

Figure 2.29 Improvement of throughput time on MVP-tree with 4 children in internal nodes

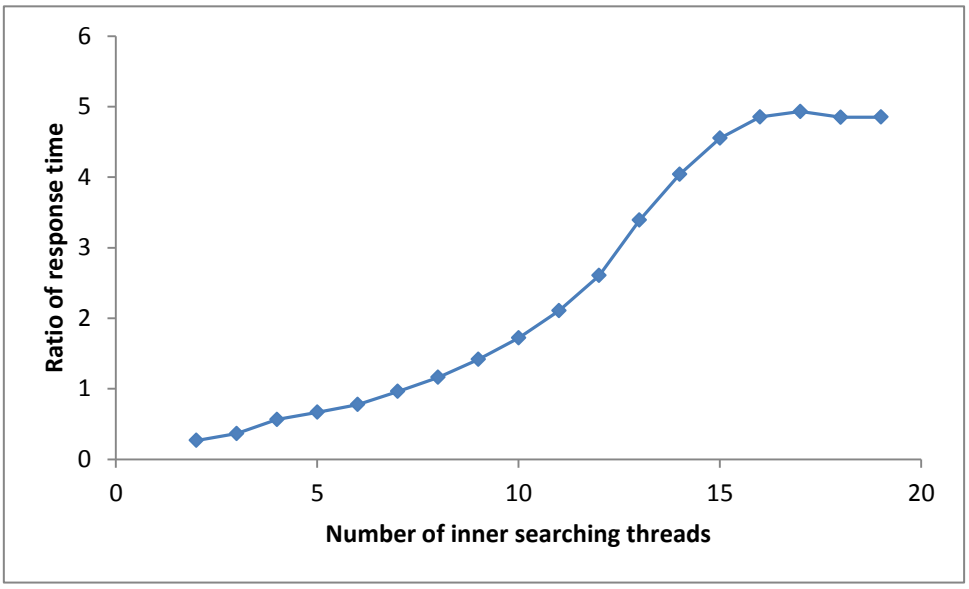

图 2.30 四分支 TPT 响应速度加速比

Figure 2.30 Improvement of response time on MVP-tree with 4 children in internal nodes

（1）第一组实验测试的 MVP-tree 索引支撑点个数为 2，单个支撑点个数的数据划分个数为 2，内部结点最大子分支数目为 4，实验过程中 TPT 的总线程数目为 20，CPST 线程数目从 2-19 调整，响应的查询之间并行化的线程数目为 20 - CPST。图 3.29 中为 TPT 吞吐量时间的加速比，比较对象是串行查询的吞吐量时间。从图中可以看出单子分支查询线程数目为 16 时，吞吐量时间的加速效果达到顶峰值 15.38 倍。因为此时，查询之间和查询内部达到了最大的的并行化。图 3.30 为串行查询响应时间与 TPT 查询响应时间的比值，可以发现 TPT 的加速效果随着子分支查询线程数目的增加而增加。

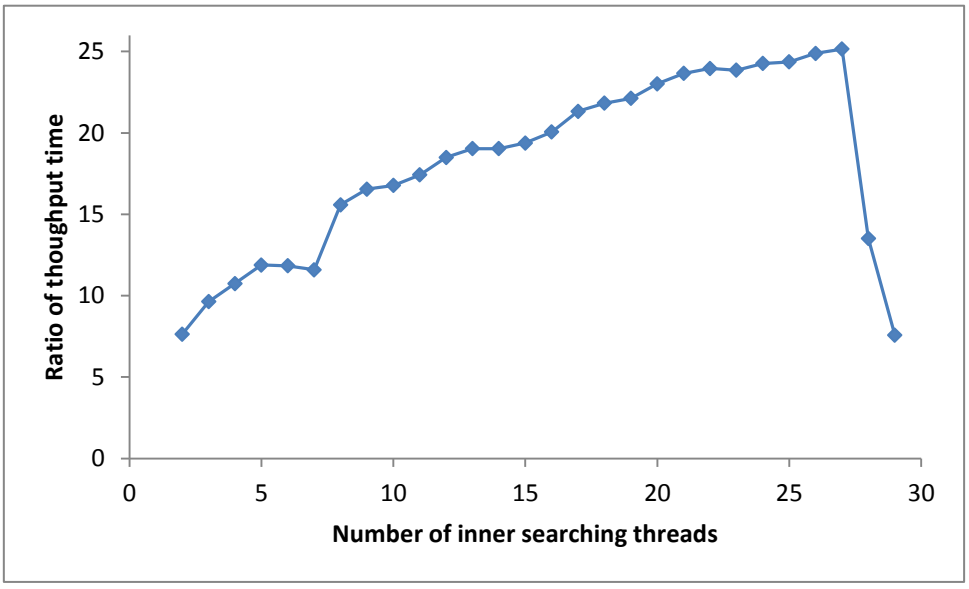

图 2.31 九分支 TPT 吞吐量时间加速比

Figure 2.31 Improvement of throughput time on MVP-tree with 9 children in internal nodes

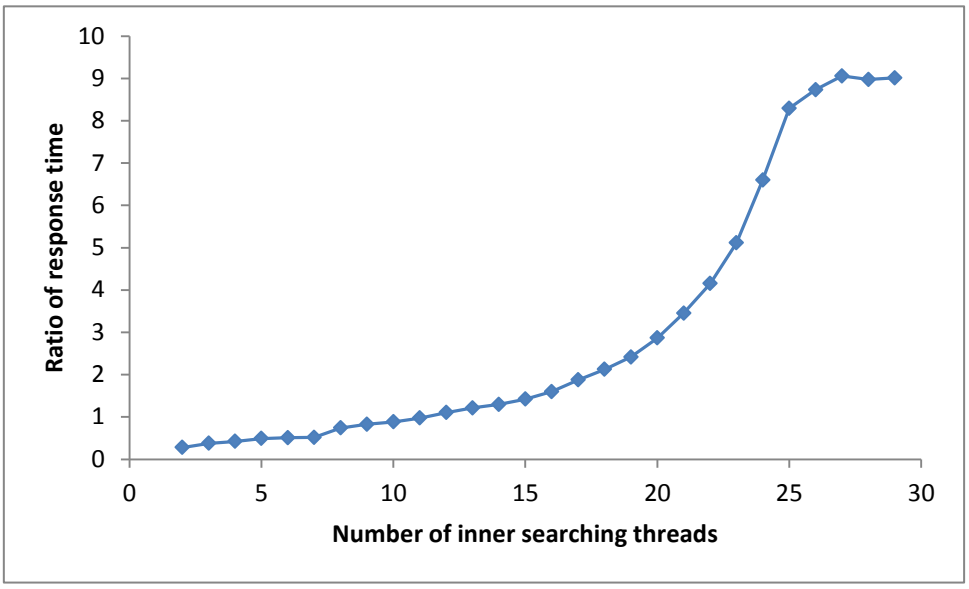

图 2.32 九分支 TPT 响应时间加速比

Figure 2.32 Improvement of response time on MVP-tree with 9 children in internal nodes

（2）为了进一步验证 TPT 的对 GeDBIT 系统吞吐量和响应速度的提高和加速效果，我们第二组实验的方法与第一组实验一样，不同地方为第二组实验中 MVP-tree 索引的支撑点数目为 2，单个支撑点对应的数据分块为 3，内部结点对应的孩子分支数目最大个数为 9。实验过程中 TPT 总线程数目设定为 30，CPST 线程数目从 2-29 调整，响应的查询之间并行化的线程数目为 30 - CPST，图 3.31 和 3.32 是结果信息。本组实验的结论与上一组实验的结论相似，当子分支数目调整到一定数目时，TPT 的吞吐量时间加速度达到最大值，因为本组实验与上组实验的所测试的索引结构中内部结点子分支的数目不同，所以本组实验是子查询分支处理线程数目为 25 时，吞吐量时间的最大加速比为 25.15 倍。图 3.32 为串行查询处理查询的响应时间和 TPT 处理查询的响应时间的比值，比值越大说明 TPT 查询的响应时间相对于串行查询的响应时间越小，从图中可以看出，本组实验与上组实验结论相同，响应速度的加速比随着子分支查询任务线程数目的增加而提高。

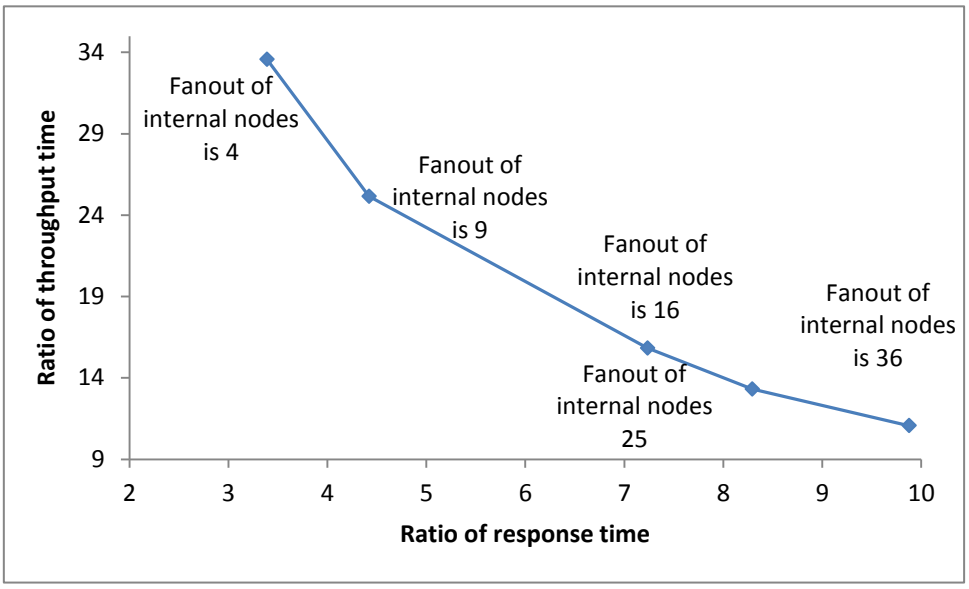

图 2.33 固定线程数目的 TPT 吞吐量和响应速度的加速比

Figure 2.33 Improvement of throughput and response time by TPT with fixed number of threads

（3）本小节最后一组实验我们在 MVP-tree 索引结构测试 TPT 的效果。我们测试的索引结构中，内部结点与其子分支结点数目的比值分别为 1:4、1:9、1:16、1:25,、1:36，相应的我们分别将 TPT 中实现查询之间并行化的线程与实现子分支查询的线程数目设为 40:10（共 50 个线程）、45:5（共 50 个线程）、48:3（共 51 个线程）、1:36（共 37 个线程）。结果如图 3.33 所示。试验中所使用的查询数目为 5000，查询半径为 2。图 3.33 中横坐标为串行查询不使用缓冲的响应时间和 TPT 处理查询请求的响应时间的比值，纵坐标为串行查询处理查询请求的吞吐量时间和 TPT 处理完所有查询请求的吞吐量时间的比值。从结果走势可以看出，当 MVP-tree 索引内部结点扇出为 4 时 TPT 的吞吐量改善效果最好为 33.57 但是响应时间的加速比最小且仅有 3.39，而扇出为 36 时 TPT 的响应速度提高最好为 9.89 但是吞吐量改善效果最差切仅有 11.06 倍。这是因为当扇出为 4 时，实现查询之间并行化的线程数目占用总线程数目的比例最大并且为 20%，随着扇出的增加，处理查询内部并行化任务的线程占用总线程数目比例增加，当索引扇出为 36 时，查询内部线程数目比例最大为 36/37，所以其响应时间的加速比最高。从这个结果可以看出 UMAD 用户在使用系统进行相似性操作时，如果用较快的时间处理大批量的查询可以将 CPST 的线程数目设定小一些，而在追求单个查询快速的相应速度时可以将 CPST 的数目设定的大一些。

2.7 多种数据类型 MVP-tree 查询并行化性能测试

本实验使用其它三种数据类型：文本序列、DNA 序列和蛋白质序列在 GeDBIT 系统上构建 MVP-tree 索引并测试各个并行查找方法的性能，索引基于的数据量大小分别是 6 万，100 万和 50 万。索引支撑点数目为 2，单个支撑点对应的数据分块为 3，内部结点对应的最大分支数目为 9。查询请求的数目分别从 10—100—1000 调节，实验过程中记录了 STP 与 TPT 处理所有查询请求的吞吐量时间和响应时间，并将其与串行查询不使用缓冲情况下处理相同个数的查询请求的吞吐量和响应时间作比得出加速比值。

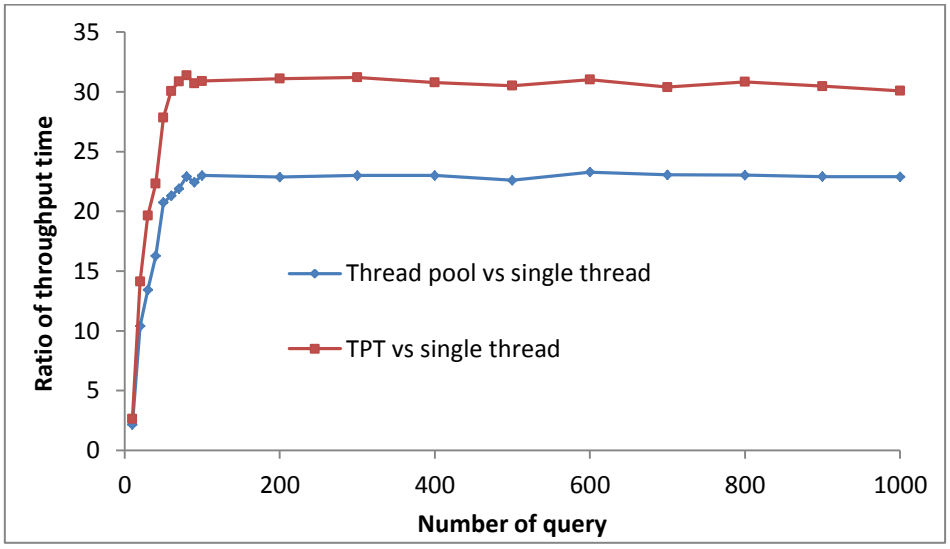

图 2.34 DNA 序列吞吐量时间

Figure 2.34 Improvement of throughput time on DNA

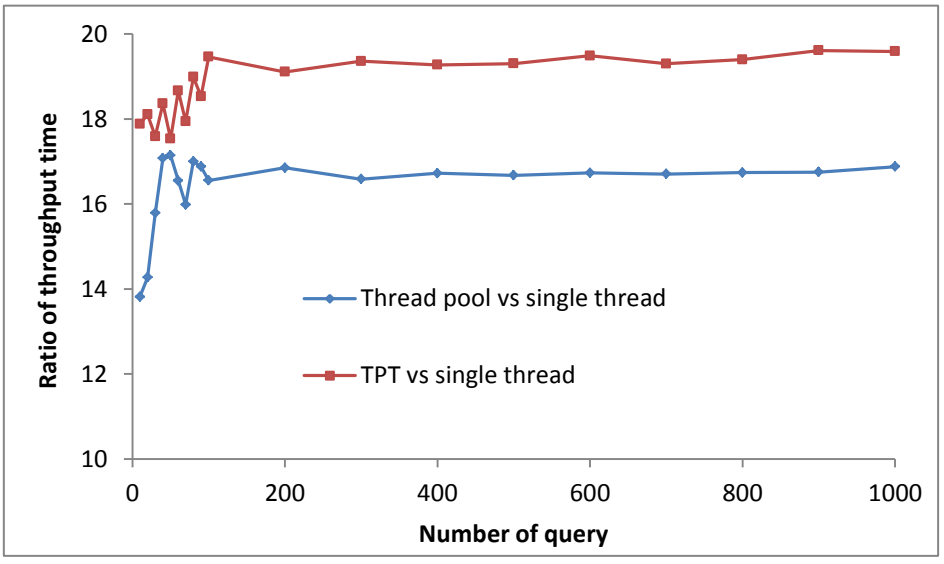

图 2.35 文本序列吞吐量时间加速比

Figure 2.35 Improvement of throughput time on string

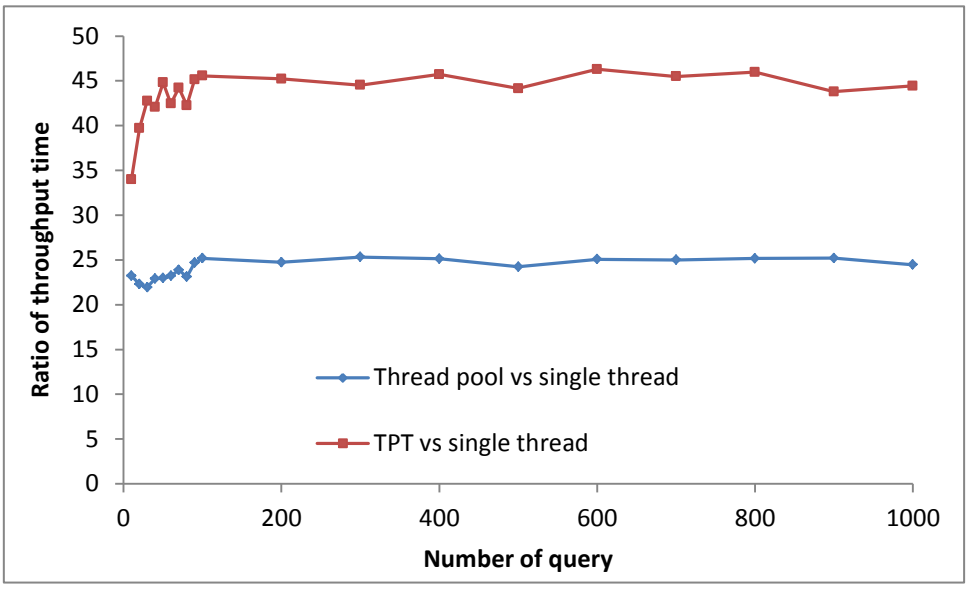

图 2.36 蛋白质序列吞吐量时间加速比

Figure 2.36 Improvement of throughput time on peptide

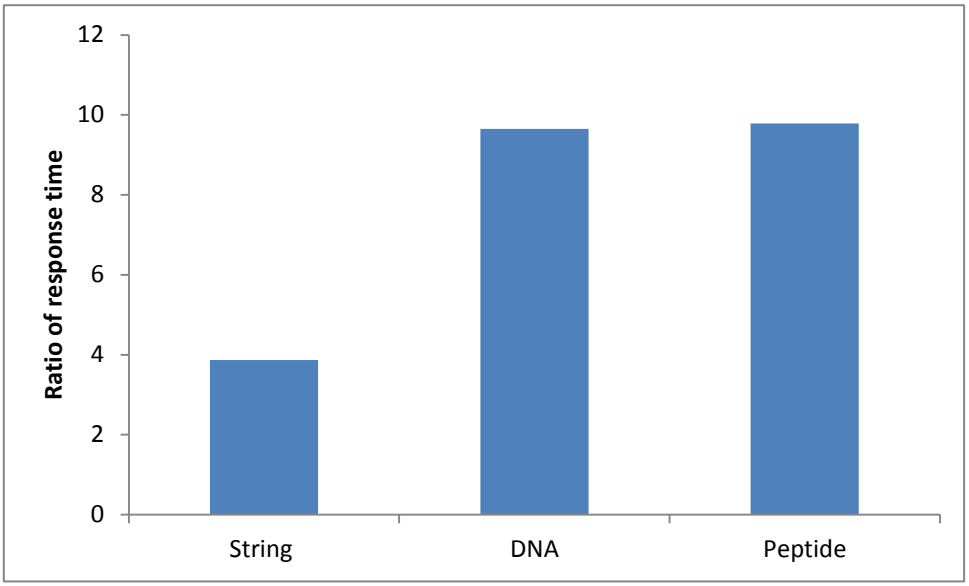

图 2.37 响应速度加速比

Figure 2.37 Improvement of response time on string, DNA, peptide

图 3.34、3.35、3.36 为吞吐量时间的对比图，图 3.37 为响应时间的对比图。实验中静态线程池中线程的数目为 30，而 TPT 中的线程总数为 30，子分支查询的数目为 20。从吞吐量时间的对比图中我们可以发现静态线程池和 TPT 都很好的提高了 GeDBIT 系统的吞吐量时间，并且二者的提高效果相差不大。从响应时间的对比图（图 3.37）可以很明显的看出 TPT 加速效果，蛋白质数据类型的索引查询响应速度加快了 9.79 倍。

2.8 本章小结

本章首先在 GeDBIT 上设计和实现了一种 MVP-tree 索引构建的并行化方法；其次，分别从改善 GeDBIT 系统吞吐量和响应速度的角度出发，设计和实现了三种查询请求之间并行化方法和两种查询内部并行化方法；最后设计和实现了一种同时提高和加快 GeDBIT 系统吞吐量和响应速度的并行化方法。通过使用多种数据类型测试 MVP-tree 索引三种并行化方法，可以得出本章设计的多种并行化方法很好的改善了 GeDBIT 相似性搜索系统的性能。

第3章 并行分布式 MVP-tree 设计和实现

3.1 引言

目前，计算机相当的普及，用普通计算机组建集群的成本越来越低，而集群有单台计算机机系统无法相比的处理存储资源和可扩展性优势，这突出了在集群上实现基于度量空间索引的通用相似性搜索系统的并行化进而提高相似性检索性能的重要性，本章讲述了在由多台计算机组建的集群上实现 MVP-tree 的多机并行化方法。

3.2 MVP-tree 多机并行构建思路

在所有的分布式索引架构中，一般认为局部索引架构的性能最好，所以本论文研究采用局部索引架构构建多机并行的 MVP-tree 相似性检索系统。图 4.1 展示的是本论文采用局部索引架构构建索引时的数据划分与构建过程。首先，用户在客户端提交原始数据，客户端根据用户访问的集群配置信息发送连接请求到集群中所有的计算节点以确认计算节点是否完好，然后发送索引构建参数（包括数据类型，支撑点数目，数据分块数目，索引名字等）到计算节点，参数发送完毕后根据完好节点的数目将用户提交的数据划分为相应数目的分块并发送到每一个计算节点上。计算节点接受到客户端发送的参数后启动与数据类型相应的数据接收过程，接受完数据并将数据存储到本地磁盘后开始执行索引构建任务，在索引构建过程中，根据用户指定的参数调用相应的串行构建或者多线程并行构建方法。在索引构建完毕后，将构建索引的信息反馈到客户端，客户端在发送完参数和数据后进入睡眠等待状态，当计算节点返回索引构建信息后唤醒客户端，客户端汇总所有的索引构建信息后反馈到用户。

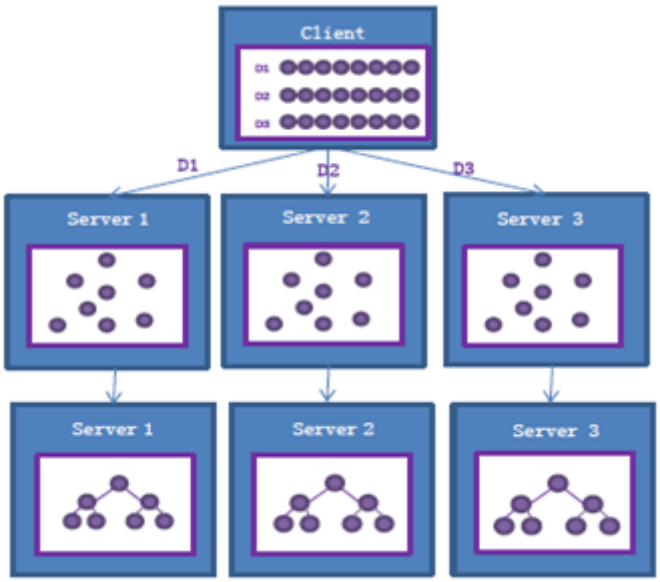

图 3.1 MVP-tree 索引多机并行构建示意图

Figure 3.1 Building process of multi-computer parallelized MVP-tree

3.3 MVP-tree 多机并行查询思路

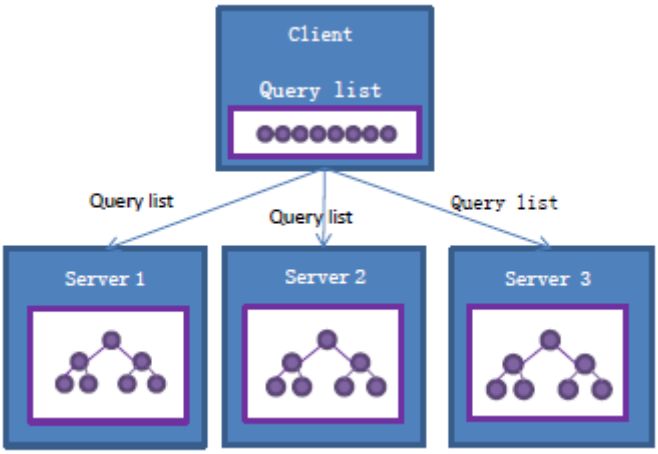

图 3.2 MVP-tree 索引多机并行查询示意图

Figure 3.2 Searching in multi-computer parallelized MVP-tree

多机并行 MVP-tree 局部索引构建完成后，各个局部索引信息都存储在了集群中的各个计算节点的本地磁盘。用户在已经构建好的索引上执行查询请求时，同样是先将查询请求信息提交到客户端，客户端判断计算节点的状态并根据状态发送查询参数到各个

计算节点，然后将查询请求的数据对象向每一个计算节点发送一份，然后客户端进入睡眠状态等待计算节点的返回结果。计算节点接受完参数以及查询数据对象后启动查询过程执行查询操作，在每个计算节点上根据客户端传来的参数调用相应的串行查询或者多线程并行查询，然后将结果返回给客户端。客户端被唤醒后汇总所有查询请求的结果信息然后反馈给用户。

3.4 并行分布式 MVP-tree 性能测试

本章实验分为两大模块，分别是多机并行的 MVP-tree 索引构建部分和查询部分。本实验集群为 3 台安装有 64 位的 RedHat 操作系统的多处理器计算机组成，每台机器有 4 个物理 CPU，32 个逻辑处理核，内存为 64G，物理磁盘各 300G。

3.4.1 并行分布式 MVP-tree 构建性能实验测试

下面两图分别展示了在两个和三个计算服务节点组成的集群上的索并行引构建方法行性能测试结果，测试的最大数据集为 100 万条 20 维浮点数向量，构建的多机并行的 MVP-tree 索引支撑点数目为 2，单个支撑点划分的数据块为 3。

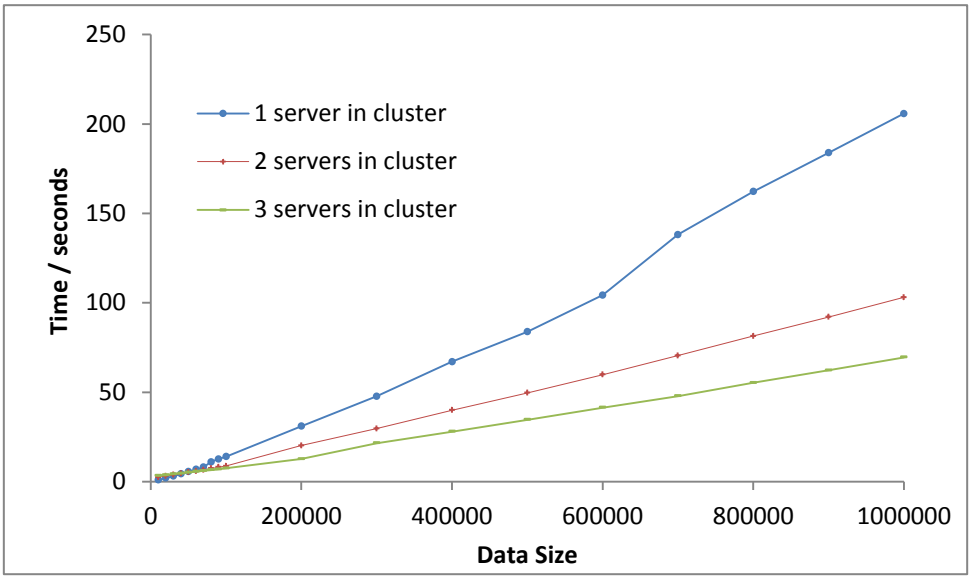

图 3.3 集群中 MVP-tree 局部索引串行构建性能对比

Figure 3.3 Performance comparison of building MVP-tree sequentially upon cluster with 1、2、3 servers

在图 4.3 的实验中，集群计算节点数目由 1、2、3 变化，构建索引数据集合由 10000—100000—1000000 变化，从图中可以看出当数据集合小于等于 40000 时，2 计算节点

和 3 计算节点集群并行构建 MVP-tree 索引的性能慢于单节点，当数据集大于 40000 时，随着集群中计算结点数目的增加，索引构建的速度也随之加快，这是因为结点数目比较小时，整体索引的结构比较小，在多机并行构建索引过程中需要首先将数据分块，然后发送到集群中的各个计算节点上，然后计算结点启动索引构建过程，所有这些过程需要消耗时间导致多台机器并行构建索引的时间超过了单台机器，并且在两台机器上构建索引的速度快于三个节点。当数据集为 100 万时，2 个结点的索引构架速度是单个结点的 1.997 倍，而 3 个结点时索引构建的速度是单个结点的 2.99 倍。

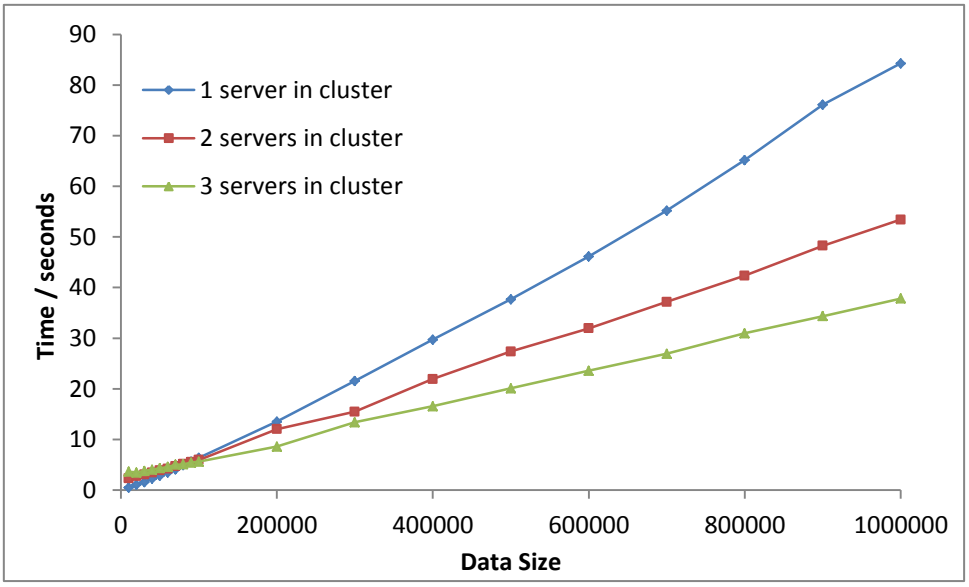

图 3.4 集群中 MVP-tree 局部索引并行构建性能对比

Figure 3.4 Performance comparison of building MVP-tree using multiple threads upon cluster with 1、2、3 servers

图 4.4 实验中使用的集群和数据集合变化规律与图 4.3 中实验中的数据变化规律一样，不同地方是图 4.4 实验集群中每个结点构建索引的方式为第三章第一节多线程并行索引构建方式。在前面多线程并行构建索引的实验中我们已经知道，当使用的数据集比较小时，并行构建索引的速度相对于串行构建提升幅度小，这是因为创建和撤销线程消耗的时间导致并行构建索引使用的时间比较大。我们现在测试的集群包含 3 个计算节点，当用户提交索引构建请求时需要将数据切分并发送到计算节点，因而当数据量小时，由于网络通信开销和结点线程创建和撤销开销，集群并行化构建索引的速度反而慢于单机的速度。由图 4.4 的结果可以看出当客户端提交的数据小于等于 80000 时，2 个结点和 3 个结点构成的集群构建索引的时间大于串行构建。当数据集大于等于 90000 时，随着数据集的增大和集群结点数目的增加，索引构建的速度也随之加快。

下面实验我们将集群中机器数目固定，比较集群中串行与多线程并行构建索引的性能。图 4.5 与图 4.6 分别为在由两个计算节点和三个构成的集群中分别用串行和多线程并行方法构建索引使用的时间的对比，可以看出两幅图的结果与在单台机器上二者的性能对比效果相同，在原始数据量小的情况下整个索引需要的时间少，由于网络传输和构建多个线程消耗了一定的时间，导致并行索引构建算法加速比小，当数据量增大时，整体索引构建所需要的时间比较大，而此时网络传输和创建和撤销多个线程的时间可以忽略不计，所以并行构建索引的性能越来越明显。当数据量为 100 万时，集群索引构建的速度是单机的 1.84 倍。

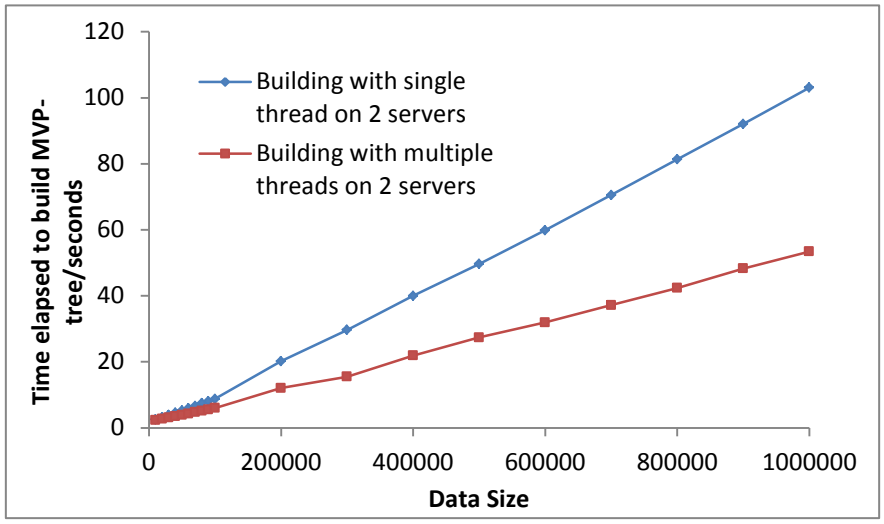

图 3.5 两个计算节点集群上 MVP-tree 局部索引串行与并行构建性能对比

Figure 3.5 Performance of sequential vs multiple-thread parallelized MVP-tree building process upon cluster with 2 server nodes

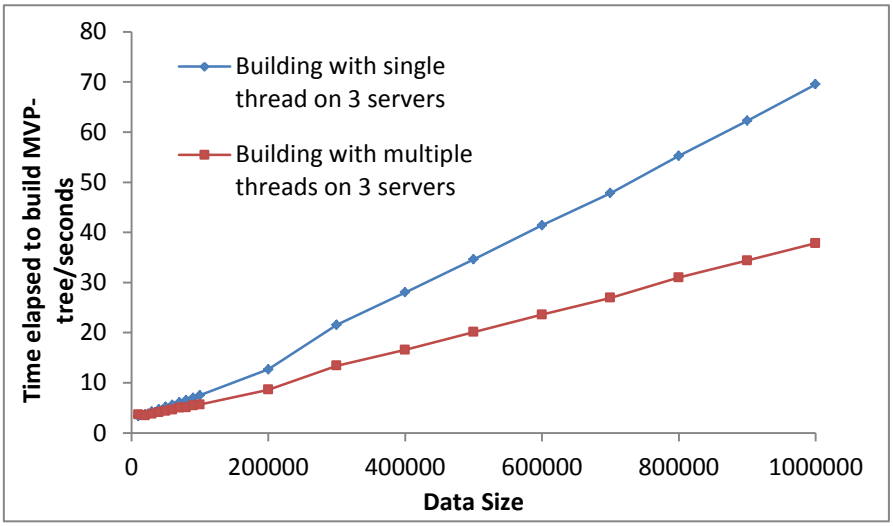

图 3.6 三个计算节点集群上 MVP-tree 局部索引串行与并行构建性能对比

Figure 3.6 Performance of sequential vs multiple-thread parallelized MVP-tree building process upon cluster with 3 server nodes

3.4.2 并行分布式 MVP-tree 查询性能实验测试

本实验主要包含 3 部分，第一部分实验在集群中各个计算节点进行本地 MVP-tree 索引的串行查询性能测试；第二部分实验在集群中各个计算节点进行本地 MVP-tree 索引的查询之间并行化性能测试;第三部分实验在集群中各个计算节点进行本地 MVP-tree 索引的查询之间和查询内部同时并行化（即 TPT）的性能。本实验中的多机并行 MVP-tree 索引所基于的数据集合为 100 万条 20 维浮点数向量，索引中支撑点数目为 2，单个支撑点的数据分块数目为 3。

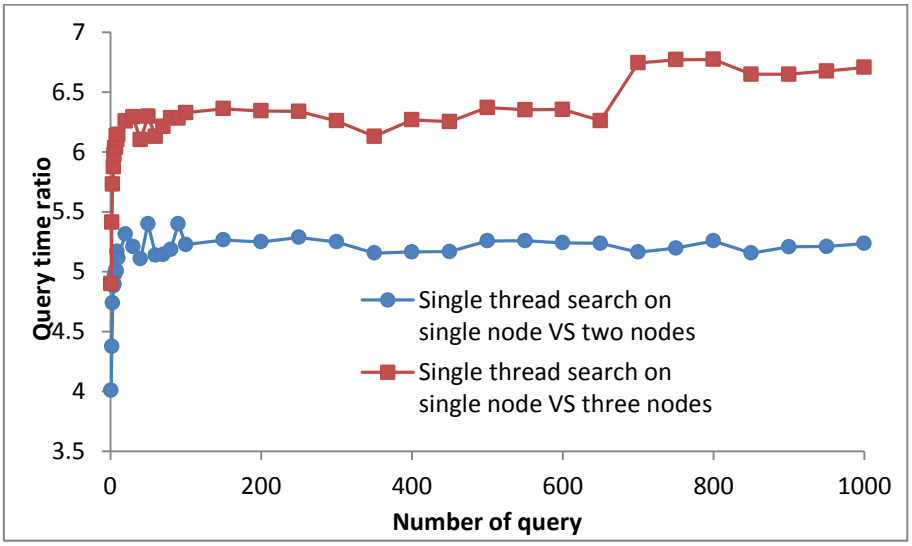

图 3.7 两个和三个计算节点集群上 MVP-tree 索引串行查询性能对比

Figure 3.7 Performance of sequential vs multiple-thread parallelized MVP-tree building process upon cluster with 3 server nodes

图 4.7 实验比较对象为 2 计算节点集群和 3 计算节点集群执行串行查询相对于单节点执行串行查询的加速比。图中横坐标为查询请求的个数，纵坐标为加速比值。从图中可以看出在查询请求个数为 800 时，三节点集群最大加速比为 6.77 倍，2 结点集群当查询个数为 90 时最大加速比为 5.40。图 4.8 和图 4.9 实验测试的数据集和集群配置与图 5.7 实验相同，不同点为图 4.8 实验集群各个机器执行的查询为 3.4 节中介绍的静态线程池，而图 4.9 实验集群中各个结点执行的查询为 3.6 节中介绍的 TPT。由于 100 万的数据划分到每个计算节点只有 33 万的数据，经测试 33 万数据构建的 MVP-tree 索引在静态线程池中线程数目为 3 时，系统吞吐量时间最小，所以图 4.8 实验中各个计算节点静态线程池的线程数目为 3。而图 4.9 实验中 TPT 的总线程数据设为 30，子线程查询数

目为 25。图 4.8 比较对象为 2 计算节点集群和 3 计算节点集群多机并行化的 MVP-tree 查询的时间相对于单节点静态线程池的吞吐量时间的加速比。横坐标是查询请求的个数纵坐标是总消耗时间的比值。当查询数目 600 时，多机并行的相似性搜索系统查询的最大加速比为 3.99 倍，而当查询数目为 500 时，2 计算节点集群的最大加速比为 2.60 倍。图 4.9 比较对象为 2 计算节点集群和 3 计算节点多机并行化的 MVP-tree 查询的时间相对于单节点 TPT 的总消耗时间的加速比。横轴为同时提交的查询请求的个数，纵轴为多机并行的 MVP-tree 查询时消耗时间与单机消耗时间的比值。查询请求的个数为 400 时，3 计算节点的最大加速比为 3.47，此时 2 节点集群也达到最大加速比，最大加速比值为 2.33 倍。

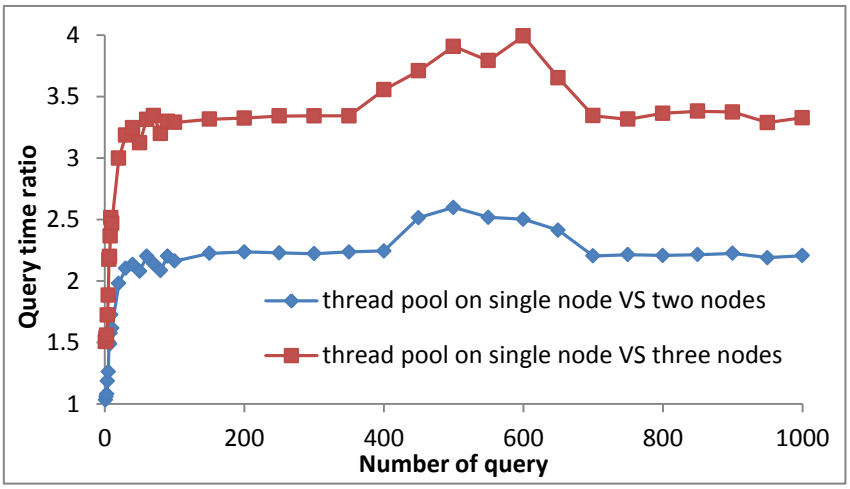

图 3.8 两个和三个计算节点集群上 MVP-tree 索引并行查询性能对比

Figure 3.8 Performance of search in MVP-tree with STP upon cluster with 2 server nodes vs 3 server nodes

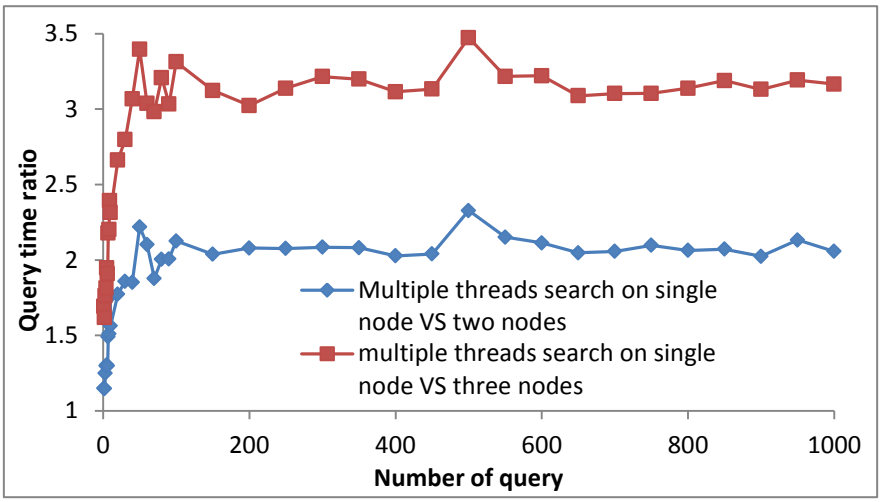

图 3.9 两个和三个计算节点集群上 MVP-tree 索引 TPT 性能对比

Figure 3.9 Performance of search in MVP-tree with TPT upon cluster with 2 server nodes vs 3 server nodes

从上面三组实验可以看出当集群为计算节点数目为 3 时，多机并行的加速比都大于 3，同理当集群有 2 个计算节点时，多机并行查询的加速比值大于 2，达到了超线性加速比效果。下面我们统计了集群中在查询过程中执行的距离计算次数。统计实验中 MVP-tree 索引基于的原始数据量为 100 万条 20 维浮点数向量，查询个数为 10，表中的距离计算次数为 10 次查询过程的平均距离计算次数。

表 3-1 多机并行的 MVP-tree 相似性检索系统各计算节点查询过程中距离计算次数统计

Table 3-1 Number of distance calculation during search process of every server in the multi-computer parallelized MVP-tree similarity searching system

集群服务器数目	1	2		3		
服务器编号	server 1	server 1	server 2	server 1	server 2	server 3
数据量	1,000,000	500,000	500,000	333,333	333,333	333,334
距离计算次数	999,998	444,444	500,000	296,296	296,297	296,296
总次数	999,998	944,444		888,889		

可以看出集群的结点的不同，每个计算节点得到的数据块不同，在构建索引过程中选择的支撑点数目也不同，这将使得在索引检索过程中距离的计算次数也不同，从统计结果中可以看出本章实验中，由两台和三台机器组成的集群在查询过程中计算距离的总次数小于单台计算机情况，而集群中多台机器的处理器和内存资源明显多于单台计算机，当集群中每个计算节点都采用并行查询算法时，集群创建的总查询线程大于单台机器情况，所有这些情况说明集群拥有的资源多但是执行的任务少于单台计算机情况下，所以出现了超线性加速比，尤其是在串行查询没有使用缓冲的情况下超线性加速比更加明显，这是因为每次查询过程中还需要从磁盘中获取目标结点信息，两台和三台机器组建的集群计算距离的次数少，索引在查询过程中排除效果好，从磁盘读取的结点总数目小于单台计算机情况下，再加上集群拥有的磁盘缓冲空间大于单台计算机且不同结点在访问磁盘读取不同的数据时磁盘缓冲命中率不同。

3.5 本章小结

本章设计和实现了一种多机并行的 MVP-tree 索引检索系统，系统用户可以通过客户端提交任务构建多机并行的 MVP-tree 索引和查找。同时，用户可以配置集群的节点

数目，给定和在串行的 GeDBIT 系统使用过程中相同的参数在集群上执行索引构建和查找任务。从本章的实验结果可以看出，多机并行的 MVP-tree 索引检索系统在原系统 GeDBIT 的基础上在索引构建和查询性能都明显提高。

第4章 总结与展望

本论文针对度量空间相似性检索模块 GeDBIT 的性能问题，对其核心索引结构 MVP-tree 进行并行化工作，工作中设计和实现的多种方法已经很好的解决了目前系统在处理相似性检索过程中存在的性能问题。下面对本论文工作内容进行归纳，同时指出了几点未来的工作方向。

4.1 论文工作总结

首先，实现了通用数据管理分析系统 UMAD，独立完成其相似性索引模块以及三个基础模块：支撑点模块、距离模块、数据对象模块。用目前最流行的 MVP-tree 作为相似性搜索系统模块 GeDBIT 的索引结构，并利用 GeDBIT 实现多种数据类型的 MVP-tree 索引构建和相似性检索。在本论文中，我们将 GeDBIT 作为 MVP-tree 索引结构的构建和查询并行化方法的实验测试平台。

其次，设计和实现了一种并行化构建 MVP-tree 索引结构的方法，借助多核计算机的多处理核的优势，为索引分支的构建过程分配单独的构建线程，使得多个任务之间并行执行。通过多种数据类型的实验测试证明此并行化方法对系统索引构建性能有明显的提高。

第三，提出多种基于 MVP-tree 相似性检索系统的并行检索方法。为了提高索引系统的吞吐量，给不同的检索请求分配不同的检索线程；为加快索引系统的响应速度，给索引的查询分支分配单独查询线程；为加快索引系统的整体检索性能，采用了内存缓冲节点的方法，减少各个独立平行执行的查询线程之间竞争磁盘的次数；通过实验测试确定出各个并行查询算法性能最佳时的参数值以及各个参数的搭配方式。

第四，设计和实现了一种基于 MVP-tree 的并行分布式局部索引架构的相似性检索检索系统，并在该系统上实现其多机并行化的 MVP-tree 索引构建和查询，从实验结果可以看出，不但基于原始串行的 GeDBIT 的索引构建和查找性能有所提高，而且在多线程并行化构建和查找的基础上提高了索引系统的性能。

4.2 展望

本论文实验部分结果可以证明本论文的工作很好的解决了我们在相似性索引过程中存在的性能问题，但在未来支持和应对更加复杂多变的应用环境时，需要从以下几个方面继续拓展：

首先，我们在并行化 MVP-tree 索引构建的结果中可以发现多种数据类型的索引构建的性能提升被锁定到 2.35 倍，其主要原因是根节点在构建过程中占用整个索引构建的很大一部分时间。未来可以研究实现一种索引结点内部并行化构建方法来减少构建根节点占用的时间，进一步提高整个索引的构建性能。

其次，本研究所使用集群由普通的机器组成，当集群扩大时，集群中的结点故障问题是常见的问题，为了在这种复杂情况下保证 MVP-tree 索引构建和检索的正确和可靠性，需要在容错方面作进一步的保证工作。

最后，调研目前一些已经成熟的分布式计算系统，这些系统提供了现成可靠的分布式负载均衡与容错机制，通过分析这些系统原理，结合 MVP-tree 索引构建和查找的特点，将度量空间索引和检索系统移植到这些分布式框架上，可以更好提高和保证相似性搜索的性能和可靠性。

参 考 文 献

[1] Batko Michal, Zezula Pavel. GHT*: Distributed Generalized Hyperplane Tree Structure [J]. 2008.

[2] 冯玉才，曹奎，曹忠升．一种支持快速相似检索的多维索引结构［J］．软件学报，2002，13(8)：8.

[3] Stevens Robert, Goble Carole, Baker Patricia, Brass Andy. A classification of tasks in bioinformatics [J]. Bioinformatics, 2001, 17(2): 180-188.

[4] Bayer R., McCreight E. Organization and maintenance of large ordered indices [J]. Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control, 1970, 107-141.

[5] 张军旗，周向东，王梅，施伯乐．基于聚类分解的高维度量空间索引 B+-Tree［J］．软件学报，2008，19(6)：1401-1412.

[6] Bayer Rudolf. Symmetric binary B-Trees: Data structure and maintenance algorithms [J]. Acta Informatica, 1972, 1(4): 290-306.

[7] Nievergelt J., Hinterberger Hans, Sevcik Kenneth C. The Grid File: An Adaptable, Symmetric Multikey File Structure [J]. ACM Trans Database Syst, 1984, 9(1): 38-71.

[8] Bentley Jon Louis. Multidimensional binary search trees used for associative searching [J]. Commun ACM, 1975, 18(9): 509-517.

[9] Guttman Antonin. R-trees: a dynamic index structure for spatial searching [J]. SIGMOD Rec, 1984, 14(2): 47-57.

[10] Lawder Jonathan K, King Peter JH. Using space-filling curves for multi-dimensional indexing [M]. Advances in Databases. Springer. 2000: 20-35.

[11] Blott Stephen, Weber Roger. A simple vector-approximation file for similarity search in high-dimensional vector spaces [J]. ESPRIT Technical Report TR19, ca, 1997,

[12] Abdi Hervé, Williams Lynne J. Principal component analysis [J]. Wiley Interdisciplinary Reviews: Computational Statistics, 2010, 2(4): 433-459.

[13] Needleman Saul B, Wunsch Christian D. A general method applicable to the search for similarities in the amino acid sequence of two proteins [J]. Journal of molecular biology, 1970, 48(3): 443-453.

[14] Chávez Edgar, Navarro Gonzalo, Baeza-Yates Ricardo, Marroquín José Luis. Searching in metric

spaces [J]. ACM computing surveys (CSUR), 2001, 33(3): 273-321.

[15] Hjaltason Gisli R, Samet Hanan. Index-driven similarity search in metric spaces (survey article) [J]. ACM Transactions on Database Systems (TODS), 2003, 28(4): 517-580.

[16] Samet Hanan. Foundations of Multidimensional and Metric Data Structures [M]. 1st ed.: Morgan Kaufmann, 2006.

[17] Bartoš Tomáš, Skopal Tomáš, Moško Juraj. Efficient indexing of similarity models with inequality symbolic regression [J]. Proceedings of the 15th annual conference on Genetic and evolutionary computation, 2013, 901-908.

[18] Steven Roman. Advanced Linear Algebra [M]. 3rd ed.: Springer-Verlag, 2008.

[19] Burkhard W. A., Keller R. M. Some approaches to best-match file searching [J]. Commun ACM, 1973, 16(4): 230-236.

[20] Uhlmann Jeffrey K. Satisfying general proximity/similarity queries with metric trees [J]. Information Processing Letters, 1991, 40(4): 175-179.

[21] Yianilos Peter N. Data structures and algorithms for nearest neighbor search in general metric spaces [J]. SODA, 1993, 93(194): 311-321.

[22] Bozkaya Tolga, Ozsoyoglu Meral. Indexing large metric spaces for similarity search queries [J]. ACM Trans Database Syst, 1999, 24(3): 361-404.

[23] Mao R., Weijia Xu, Ramakrishnan S., Nuckolls G., Miranker D. P. On optimizing distance-based similarity search for biological databases [J]. Computational Systems Bioinformatics Conference, 2005 Proceedings 2005 IEEE, 2005, 351-361.

[24] Papadopoulos Apostolos N, Manolopoulos Yannis. Distributed processing of similarity queries [J]. Distributed and Parallel Databases, 2001, 9(1): 67-92.

[25] Ciaccia Paolo, Patella Marco, Zezula Pavel. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces [J]. Proceedings of the 23rd International Conference on Very Large Data Bases, 1997, 426-435.

[26] Traina Agma J. M., Traina Agma, Faloutsos Christos, Seeger Bernhard. Fast indexing and visualization of metric data sets using slim-trees [J]. Knowledge and Data Engineering, IEEE Transactions on, 2002, 14(2): 244-260.

[27] Mao Rui, Xu Weijia, Singh Neha, Miranker Daniel P. An assessment of a metric space database index

to support sequence homology [J]. International Journal on Artificial Intelligence Tools, 2005, 14(05): 867-885.

[28] Carlo Caio César Mori, Pola Ives Rene Venturini, Ciferri Ricardo Rodrigues, Traina Agma Juci Machado, Traina Caetano, de Aguiar Ciferri Cristina Dutra. Slicing the metric space to provide quick indexing of complex data in the main memory [J]. Information Systems, 2011, 36(1): 79-98.

[29] Qiu Chu, Lu Yongquan, Gao Pengdong, Wang Jintao, Lv Rui. A Parallel Bulk loading Algorithm for M-tree on Multi-core CPUs [J]. Third International Joint Conference on Computational Science and Optimization (CSO), 2010, 300-303.

[30] Lokoc Jakub. Parallel Dynamic Batch Loading in the M-tree [J]. Proceedings of the 2009 Second International Workshop on Similarity Search and Applications, 2009, 117-123.

[31] Chu Qiu, Yongquan Lu, Pengdong Gao, Jintao Wang, Rui Lv. Parallel M-tree Based on Declustering Metric Objects using K-medoids Clustering [J]. Ninth International Symposium on Distributed Computing and Applications to Business Engineering and Science (DCABES), 2010, 206-210.

[32] Bryan Brent, Moore Andrew W, Snyder Andrew, Schneider Jeff. Using Distributed M-Trees for Answering K-Nearest Neighbor Queries [M]. Carnegie Mellon University, School of Computer Science, Machine Learning Department, 2007.

[33] 张兆功，李建中．基于广义超曲面树的相似性搜索算法 A [J]. Journal of Software, 2002, 13(10).

[34] 李建中，张兆功．超平面树：度量空间中相似性搜索的索引结构 [J]. 计算机研究与发展，2003，40(8)：1209-1215.

[35] Gil-Costa Veronica, Marin Mauricio, Reyes Nora. Parallel query processing on distributed clustering indexes [J]. Journal of Discrete Algorithms, 2009, 7(1): 3-17.

[36] Batko Michal, Gennaro Claudio, Zezula Pavel. GHT*: a Peer-to-Peer System for Metric Data [J]. 2004.

[37] 张兆功，李建中．度量空间中相似性搜索的并行算法 [J]. 第十七届全国数据库学术会议论文集（研究报告篇），2000.

[38] Yu Cui, Ooi Beng Chin, Tan Kian-Lee, Jagadish HV. Indexing the distance: An efficient method to knn processing [J]. VLDB, 2001, 421-430.

[39] Jagadish Hosagrahar V, Ooi Beng Chin, Tan Kian-Lee, Yu Cui, Zhang Rui. iDistance: An adaptive

B+-tree based indexing method for nearest neighbor search [J]. ACM Transactions on Database Systems (TODS), 2005, 30(2): 364-397.

[40] Traina Agma, Traina Agma JM, Faloutsos Christos. Similarity search without tears: the OMNI-family of all-purpose access methods [J]. 17th International Conference on Data Engineering, 2001, 623-630.

[41] Traina Jr Caetano, Santos Filho Roberto F, Traina Agma JM, Vieira Marcos R, Faloutsos Christos. The Omni-family of all-purpose access methods: a simple and effective way to make similarity search more efficient [J]. The VLDB Journal, 2007, 16(4): 483-505.

[42] Wang Haitao, Zhao Yanqiong, Han Jiaxin, Yue Pang. Building an Information Retrieval System: Global Indexing or Local Indexing? [J]. 2013.

[43] Vlachou Akrivi, Doulkeridis Christos, Kotidis Yannis. Peer-to-Peer Similarity Search Based on M-Tree Indexing [M]//Kitagawa Hiroyuki, Ishikawa Yoshiharu, Li Qing, Watanabe Chiemi. Database Systems for Advanced Applications. Springer Berlin Heidelberg. 2010: 269-275.

[44] Mao Rui, Miranker Willard L., Miranker Daniel P. Dimension reduction for distance-based indexing [M]. Proceedings of the Third International Conference on SImilarity Search and APplications. Istanbul, Turkey; ACM. 2010: 25-32.

[45] Mao R., Liu S., Xu H. L., Zhang D., Miranker D. P. On Data Partitioning in Tree Structure Metric-Space Indexes [J]. Database Systems for Advanced Applications, 2014, 141-155.

[46] Gil-Costa Veronica, Barrientos Ricardo, Marin Mauricio, Bonacic Carolina. Scheduling metric-space queries processing on multi-core processors [J]. 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2010, 187-194.

[47] Tang Chunqiang, Dwarkadas Sandhya. Hybrid Global-Local Indexing for Efficient Peer-to-Peer Information Retrieval [J]. NSDI, 2004, 16-16.

[48] Tomasic Anthony, Garcia-Molina Hector. Query processing and inverted indices in shared: nothing text document information retrieval systems [J]. The VLDB Journal—The International Journal on Very Large Data Bases, 1993, 2(3): 243-276.

[49] Abusukhon Ahmad, Oakes Michael P, Talib Mohammad, Abdalla Ayman M. Comparison between document-based, term-based and hybrid partitioning [J]. First International Conference on the Applications of Digital Information and Web Technologies, 2008, 90-95.

[50] Zhang Jiangong, Suel Torsten. Optimized inverted list assignment in distributed search engine

architectures [J]. IEEE International Parallel and Distributed Processing Symposium, 2007, 1-10.

[51] Bhagwat Deepavali, Eshghi Kave, Mehra Pankaj. Content-based document routing and index partitioning for scalable similarity-based searches in a large corpus [J]. Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, 2007, 105-112.

[52] Cambazoglu B Barla, Catal Aytul, Aykanat Cevdet. Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems [M]. Computer and Information Sciences–ISCIS 2006. Springer. 2006: 717-725.

[53] Abusukhon Ahmad, Talib Mohammad, Oakes Michael P. An investigation into improving the load balance for term-based partitioning [M]. Springer, 2008.

[54] Doulkeridis, Christos, Vlachou, Akrivi, Kotidis, Yannis, Vazirgiannis, Michalis. Peer-to-peer similarity search in metric spaces [M]. 2007: 986-997.

致　谢

三年的研究生生活只剩下最后几个月了，回想 2012 年金秋时节，带着行李满怀期待的登上从家乡到深圳的火车，在火车上那一段时间一直在想研究生三年将如何去努力奋斗让自己变得更有价值，而在这三年时间里也没有辜负当时的自己，虽然一直不知道自己最终想要什么，但是一直在努力着奋斗着。不管以后会怎样，这三年都是一段充实而且很有意义的回忆。

首先，我要特别感谢我的导师毛睿教授。三年的研究生生涯，他对我的影响很大。在他的耐心指导下，我懂得了学术研究的方法，感受到了学术研究的魅力。在生活方面，他给予我许多帮助，最有价值的东西还是他让我懂得了很多做人的道理，让我意识到自己的很多不足之处，让我明白如何去面对未知的困难，让我懂得与长辈，同学，朋友交流的方法和与别人合作的重要性，这笔无价的精神财富将使我受益终身。

其次，我要感谢广东省普及型高性能计算机重点实验室的其他老师：杨烜老师，罗秋明老师，陆克中老师，李荣华老师，王毅老师，廖好老师和刘刚老师，感谢他们对我在学术研究和生活上的一些帮助。

第三，我要感谢我的各位师兄们：许红龙师兄，岳磅师兄，张恒师兄和刘胜师兄，他们教会我许多学习和科研的方法，让我快速的进步。

最后，我要感谢我的父母，他们给了我很大的精神支持让我在研究生期间学习和科研道路上不断前进，是他们的支持让我有勇气去战胜各种困难，去迎接未来未知而又充满期待的人生。

攻读硕士学位期间的研究成果

[1] Lei, Fuli, Wu, Wenbo, Li, Qiaozhi, Zhang, He, Li, Ping, Luo, Qiuming, and Mao, Rui: ‘Speed Up Distance-Based Similarity Query Using Multiple Threads’, Sixth International Symposium on Parallel Architectures, Algorithms and Programming (PAAP), 13-15 July, 2014, pp. 215-219 (EI 会议)