doi:10.11887/j.cn.202401010 http://journal.nudt.edu.cn

面向众核处理器的阴阳 *K*-means 算法优化

周天阳[1,2],王庆林[1,2*],李荣春[1,2],梅松竹[1,2],尹尚飞[1,2],郝若晨[1,2],刘 杰[1,2]
(1. 国防科技大学 计算机学院,湖南 长沙 410073;
2. 国防科技大学 并行与分布计算全国重点实验室,湖南 长沙 410073)

摘 要:传统阴阳 *K*-means 算法处理大规模聚类问题时计算开销十分昂贵。针对典型众核处理器的体系结构特征,提出了一种阴阳 *K*-means 算法高效并行加速实现。该实现基于一种新内存数据布局,采用众核处理器中的向量单元来加速阴阳 *K*-means 中的距离计算,并面向非一致内存访问(non-unified memory access, NUMA)特性进行了针对性的访存优化。与阴阳 *K*-means 算法的开源多线程实现相比,该实现在 ARMv8 和 x86 众核平台上分别获得了最高约 5.6 与 8.7 的加速比。因此上述优化方法在众核处理器上成功实现了对阴阳 *K*-means 算法的加速。

关键词:*K*-means;非一致内存访问;向量化;众核处理器;性能优化
中图分类号:TP311.1 **文献标志码**:A **开放科学(资源服务)标识码(OSID)**:
文章编号:1001-2486(2024)01-093-10

Optimizing Yinyang *K*-means algorithm on many-core CPUs

ZHOU Tianyang[1,2], *WANG Qinglin*[1,2*], *LI Rongchun*[1,2], *MEI Songzhu*[1,2], *YIN Shangfei*[1,2], *HAO Ruochen*[1,2], *LIU Jie*[1,2]
(1. College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China;
2. National Key Laboratory of Parallel and Distributed Computing, National University of Defense Technology, Changsha 410073, China)

Abstract: Traditional Yinyang *K*-means algorithm is computationally expensive when dealing with large-scale clustering problems. An efficient parallel acceleration implementation of Yinyang *K*-means algorithm was proposed on the basis of the architectural characteristics of typical many-core CPUs. This implementation was based on a new memory data layout, used vector units in many-core CPUs to accelerate distance calculation in Yinyang *K*-means, and targeted memory access optimization for NUMA (non-uniform memory access) characteristics. Compared with the open source multi-threaded version of Yinyang *K*-means algorithm, this implementation can achieve the speedup of up to 5.6 and 8.7 approximately on ARMv8 and x86 many-core CPUs, respectively. Experiments show that the optimization successfully accelerate Yinyang *K*-means algorithm in many-core CPUs.

Keywords: *K*-means; NUMA; vectorization; many-core CPU; performance optimization

经典的 *K*-means 算法[1]是最重要的无监督学习算法之一,被广泛应用于数据挖掘、文档聚类、入侵检测等领域中。该算法包括两个步骤:样本点分配和中心点重选。两个步骤不断重复,直到每个中心点的位置不再改变或者迭代达到一定的次数。

当样本点数或中心点数较多时,经典的 *K*-means算法往往运行缓慢。因此,学术界提出了许多优化方法来加速算法本身,例如中心点的初始位置优化[2]、算法结构优化[3-4]和三角不等式优化[5-7]。阴阳 *K*-means[8]是基于三角形不等式优化的最流行实现之一,其通过三角形不等式有效地避免了大多数冗余的距离计算,从而性能得到大幅提升。

与此同时,许多学者在 *K*-means 算法的并行优化方面也开展了许多工作,以期通过并行计算的方式来减少其执行时间。例如:Wu 等[9]在 Intel MIC 架构上实现了 *K*-means 算法的细粒度的单指令多数据(single instruction/multiple data, SIMD)并行;Kwedlo 等[10]提出了 Drake、Elkan、Annulus 和阴阳四种 *K*-means 变种算法的 MPI/OpenMP 混合并行化;Zhao 等[11]提出了一种基于 MapReduce 的并行 *K*-means 算法;Kumar 等[12]实现了在多核超级计算机上使用大型数据集进行定

收稿日期:2022-09-06
基金项目:国家自然科学基金资助项目(62002365)
第一作者:周天阳(1995—),男,湖南湘潭人,硕士研究生,E-mail:zhoutianyang@nudt.edu.cn
***通信作者**:王庆林(1987—),男,贵州思南人,副研究员,博士,硕士生导师,E-mail:wangqinglin_thu@163.com

量生态区域划定的并行 *K*-means 聚类。此外，现有研究还尝试面向 GPU[13-15]、FPGA[16-17]等异构硬件平台进行 *K*-means 算法的并行优化。例如，Taylor 等[18]优化了阴阳 *K*-means 算法在 GPU 上的实现。因此，*K*-means 算法的并行优化是学术界关注的研究热点。

目前，众核处理器已经成为高性能计算领域中的主流平台。考虑功耗与性能的平衡，基于 SIMD 方式运行的向量单元几乎成为众核处理器的标配，如 x86 AVX-512[19]、ARMv8 Neon[20]，从而也使得向量单元成为众核处理器计算性能提升的主要来源之一。同时，众核处理器还通常采用非一致内存访问(non-unified memory access，NUMA)架构，以扩展片上计算核的数量以及访存带宽。单个处理器的所有核心组织成多个 NUMA 节点，如 AMD EPYC 系列[21]、飞腾 FT-2000+[22-24]；或者多个处理器以 NUMA 方式组织成为一个多 socket 节点。

尽管现有阴阳 *K*-means 算法的开源多线程实现可以直接运行于众核处理器上，但其性能通常不是最优的。其原因主要在于这些开源实现既没有利用众核处理器中的向量单元，也没有面向 NUMA 架构开展针对性的性能优化。

本文主要面向众核处理器上的阴阳 *K*-means 算法的高性能优化，通过向量化、NUMA 亲和性内存访问优化以及数据布局优化等手段实现阴阳 *K*-means 算法的高效并行，设计了多组实验来评估优化方法的性能，并将本文提出的阴阳*K*-means 实现与开源多线程阴阳 *K*-means[25]实现进行性能对比。

1 背景介绍

1.1 阴阳 *K*-means 算法

传统阴阳 *K*-means 算法的伪代码如算法 1 所示。该算法随机选择 k 个样本点作为中心点，并将它们分为 t 个中心点组，然后初始化所有 n 个样本点(第 1 ~ 3 行)。算法的核心是使用一个三级过滤器过滤不必要进行距离计算的样本点或中心点。具体来说，算法每轮迭代使用全局过滤器(第 9 ~ 11 行)过滤不必要进行距离计算的样本点，使用组过滤器(第 12 ~ 13 行)和局部过滤器(第 14 ~ 15 行)过滤不必要进行距离计算的中心点，最后剩下的每个样本点与所有中心点进行距离计算，根据计算结果为每个样本点找到最近的中心点，如此反复迭代，直到迭代达到一定的次数或者所有中心点的位置不再改变。

算法 1 传统阴阳 *K*-means 算法

Alg. 1 Traditional Yinyang *K*-means algorithm

输入：n 个样本点 Points$\{X_1, X_2, X_3, \cdots, X_n\}$

输出：k 个中心点 Centroids$\{C_1, C_2, C_3, \cdots, C_k\}$

```
1  随机选择 k 个中心点
2  将 k 个中心点分成 t 个组
3  初始化 n 个样本点
4  while 未收敛 do
5    更新 k 个中心点的位置
6    #omp parallel for schedule(static)
     //并行遍历所有 n 个样本点，每次循环的跨度为 1
7    for i = 1 : 1 : n do in parallel
8      更新 X_i 的 upper bound 和 group lower bound
9      if X_i 的全局过滤器条件 then
10       收紧 X_i 的 upper bound
11       if X_i的全局过滤器条件 then
           //遍历所有的中心点组 G_l
12         for l = 1 to t do
13           if G_l 的组过滤器条件 then
               //遍历每个中心点组中的所有中心点 C_m
14             for m = 1 to k/t do
15               if C_m 的局部过滤器条件 then
16                 d(X_i, C_m) = sqrt( sum_{j=1}^{d} (X_{i,j} - C_{m,j})^2 )
17                 将X_i分配给离它最近的C_m 所在的簇
```

样本点和中心点之间的距离使用欧几里得距离公式来计算：

$$d(p_i^d, c_i^d) = \sqrt{\sum_{j=1}^{d} (p_{i,j} - c_{i,j})^2} \tag{1}$$

式中，$d(p_i^d, c_i^d)$是样本点 p_i 和中心点 c_i 之间的距离，p_i 和 c_i 都有 d 个维度。阴阳 *K*-means 算法的整个执行过程如图 1 所示，图中 YYG 表示只使用全局过滤器的阴阳 *K*-means 算法，YYGG 则表示使用全局和组过滤器的阴阳 *K*-means 算法。

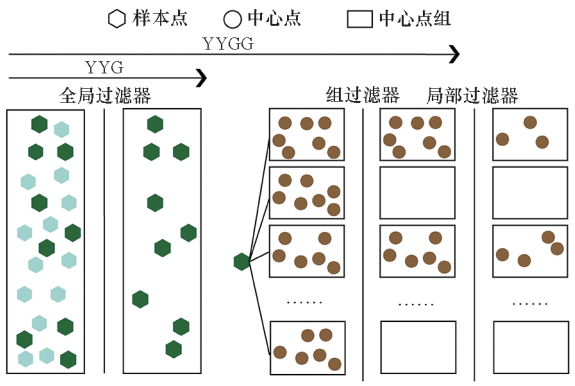

图 1 阴阳 *K*-means 算法执行过程

Fig. 1 Execution process of Yinyang *K*-means algorithm

值得一提的是,本文提出的阴阳 *K*-means 算法的并行实现中没有考虑局部过滤器,因为 Newling 等的研究[26] 表明局部过滤器会降低性能。

1.2　向量化技术

向量化也称为数据级并行化,它依赖于众核处理器中的向量处理单元以及对应的向量指令。以 Intel Xeon Gold 6240 为例,该处理器配备 512 位向量处理单元,在 AVX－512 指令集的支持下,可以利用向量乘加指令同时处理 8 个双精度浮点数或者 16 个单精度浮点数的乘加操作,大大提高了运算的并行性,从而可有效提高应用程序的性能。

1.3　非一致内存访问

NUMA 是一种由多个节点组成的内存架构。每个节点包含一个或多个处理器核心以及本地存储器,如图 2 所示。对于单个节点来说,每个节点内部的存储器称为本地内存,其他所有节点的存储器被称为远程内存,多个 NUMA 节点通过互联模块进行通信。在 NUMA 架构中,处理器核心访问内存的时间取决于内存相对于处理器核心的位置,任何处理器核心访问其本地内存的速度都比远程内存快得多。

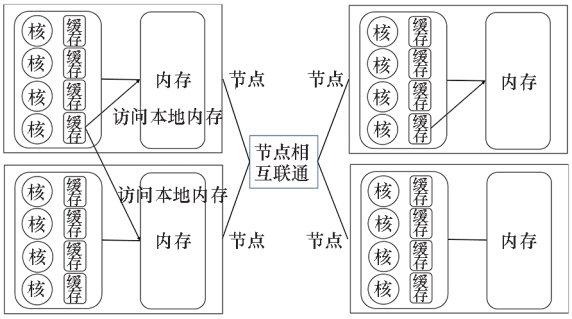

图 2　典型 NUMA 体系结构

Fig. 2　Classic NUMA architecture

2　问题分析

由于现有的研究主要使用多线程技术来对阴阳 *K*-means 算法进行并行优化,却没有利用众核处理器的向量处理单元来进一步提高算法的性能,所以对阴阳 *K*-means 算法进行向量化并行优化是非常值得研究的。向量化是并行计算中最常见的优化技术之一,对阴阳 *K*-means 算法进行向量化优化之前需要确定基于样本点数据还是中心点数据进行向量化。前者将所有样本点数据视为一个长向量,一次计算多个样本点到每个中心点的距离;后者则将所有中心点数据视为一个长向量,一次计算多个中心点到一个样本点的距离。Wu 等的研究[9] 表明,如果需要获得良好的性能,所构建的长向量必须足够长才能有效隐藏相关单元的延迟,而样本点数量要比中心点数量大得多,从而可以为 SIMD 指令流构建更长的向量。因此,决定基于样本点数据进行向量化优化。考虑到阴阳 *K*-means 算法的执行特性,不仅可以在距离计算阶段引入向量化,在其他任何可以批处理的地方都可以引入向量化。例如,可以将全局过滤器条件中的标量距离界限的比较转换为向量距离界限的比较,从而一次检查多个样本点的全局过滤器条件。但是必须调整数据内存布局,使其更适合向量化并有更好的数据局部性。

NUMA 架构良好的扩展性以及更高的访存带宽使其在超级计算机或服务器上得到了广泛应用。但是 NUMA 架构具有其天然的缺点,即远程内存访问成本高。因此,传统阴阳 *K*-means 算法的实现直接在 NUMA 架构处理器上运行时可能会因为数据没有合理分配而导致远程内存访问次数过多,从而可能会性能较差,当 NUMA 节点较多时性能下降尤其明显。所以如何减少远程内存访问量是提升算法性能的关键。这个问题可以通过线程绑定、数据分配、工作负载分配三个步骤来解决。

3　优化方法

根据上述分析,为了提升阴阳 *K*-means 算法的性能,可以利用众核处理器的向量单元进行向量化优化、针对众核处理器的 NUMA 架构进行 NUMA 亲和性内存访问优化。此外,为了支持这两种优化方法,还需优化内存数据布局,以获得更好的数据局部性,以及便于在各个 NUMA 节点进行数据分配。

3.1　向量化

传统阴阳 *K*-means 算法由于没有利用众核处理器中的向量处理单元,所以处理样本点时单周期最多只能处理一个样本点,而基于样本点数据进行向量化后则可以在单周期内同时处理 s(s 为向量长度,且大于 1)个样本点,进而算法的性能得到有效的提升。以 ARMv8 架构处理器为例,由于其配备了 128 位的向量处理单元,所以利用向量处理单元可在单周期内同时处理 2 个双精度存储的样本点(即向量长度 $s=2$)。算法 2 展示了同时采用向量化优化和 NUMA 亲和性优化的阴阳*K*-means算法的执行过程,其中多处借助了向量处理单元以提升性能。如样本点初始化阶段需要

算法 2　并行加速阴阳 *K*-means 算法

Alg. 2　Optimized Yinyang *K*-means algorithm

输入：n 个样本点 Points$\{X_1, X_2, X_3, \cdots, X_n\}$

输出：k 个中心点 Centroids$\{C_1, C_2, C_3, \cdots, C_k\}$

```
1 将每一个线程绑定到不同的核心
2 并行访问所有样本点数据以分配样本点数据到每一个节点的本地内存
3 随机选择 k 个中心点
4 将 k 个中心点分成 t 个组
5 初始化 n 个样本点(使用 SIMD 指令来计算样本点与中心点的距离)
6 while 未收敛 do
7    更新 k 个中心点的位置
8    # omp parallel for schedule(static)
     //并行遍历所有 n 个样本点,每次循环的跨度为 s
9    for i = 1 : s : n do in parallel
10      更新 X_i ~ X_{i+1} 的 upper bound 和 group lower bound
11      if X_i ~ X_{i+s-1} 的全局过滤器条件 then
12         收紧 X_i ~ X_{i+s-1} 的 upper bound
13         if X_i ~ X_{i+s-1} 的全局过滤器条件 then
              //遍历所有的中心点组 G_l
14            for l = 1 to t do
15               if G_l 的组过滤器条件 then
                    //遍历每个中心点组中的所有中心点 C_m
16                  for m = 1 to k/t do
17                     if C_m 的局部过滤器条件 then
18                        for o = 1 to d do
                             //将 X_i ~ X_{i+s-1} 的第 o 个维度数据放入向量 V_1 中
19                           V_1 = [X_{i,o}, ..., X_{i+s-1,o}]
                             //将 C_m 的第 o 个维度数据广播到向量 V_2 中
20                           V_2 = [C_{m,o}, ..., C_{m,o}]
21                           tmpVec = (V_1 - V_2)^2
22                           tmpDisVec + = tmpVec
23                        disVec = sqrt(tmpDisVec)
24                  将 X_i ~ X_{i+s-1} 分配给与其最近的 C_m 所在的簇
```

计算所有样本点与中心点之间的距离时(第 5 行),一次可以同时计算两个样本点到一个中心点的距离。具体做法是遍历样本点和中心点的所有 d 个维度,将两个相邻样本点的同一个维度数据放入一个向量中,并将单个中心点的对应维度广播到另一个向量,最后使用 SIMD 指令计算两个向量的欧几里得距离。得益于分组聚合内存数据布局所带来的良好的数据局部性,计算多个样本点与一个中心点的距离所需的内存访问都是连续的,所以非常高效。又如 for 循环开始后,由于单次 for 循环处理两个样本点,所以利用 SIMD 指令可同时检查两个样本点的全局过滤器条件(第 11 行)。如果两个样本点所在簇的中心点的位置在本次迭代都不会发生变化,则可以跳过这两个样本点;反之,如果两个样本点中至少有一个样本点通过全局过滤器,即样本点所在簇的中心点的位置会在本次迭代中发生改变,则继续收紧两个样本点的距离上限,并使用 SIMD 指令再次同时检查它们的全局过滤器条件(第 12 ~ 13 行)。与样本点初始化阶段的距离计算过程相同,第 18 ~ 23 行详细阐述了利用向量化后的样本点数据计算两个样本点与通过筛选后的中心点的距离,并为两个样本点中通过全局过滤器的样本点找到离它最近的中心点,从而在单次 for 循环中完成了两个样本点的分配,循环次数也因此减半,但是总的迭代次数不变。以上向量化操作由 ARMv8 架构提供的内部指令(ARM Neon Intrinsic)实现。

3.2　NUMA 亲和性优化

阴阳 *K*-means 算法在 NUMA 系统上运行时,所有样本点数据将被加载到单个 NUMA 节点的本地内存中,导致其他 NUMA 节点中的处理器核心对样本点数据的访问都是远程内存访问,过多的远程内存访问会明显降低算法的性能。阴阳 *K*-means 算法的执行特性表明单个 NUMA 节点中的各个处理器核心对样本点数据的处理是独立的,从而每个 NUMA 节点可以只持有部分样本点数据。理想的情况是每个 NUMA 节点需要的样本点数据都预先放在该节点本地内存中,从而每个 NUMA 节点都能快速且高效地访问自己所需的样本点数据[27]。此外,每个 NUMA 节点都需要持有所有中心点数据的一个副本。

根据对 NUMA 亲和性优化的分析,可以通过以下三个步骤实现具有 NUMA 亲和性的阴阳 *K*-means算法:

1)线程绑定:该步骤指定线程如何绑定到处理器核心。OpenMP [28] 4.0 或更高版本提供了两个环境变量——OMP_PLACES 和 OMP_PROC_BIND,这两个环境变量通常结合使用。OMP_PLACES 用于指定线程绑定到的机器上的位置;OMP_PROC_BIND 指定绑定策略(线程关联策略),它规定了线程如何分配到位置。在算法 2 的并行加速阴阳 *K*-means 算法实现中,指定线程绑定到处理器核心上,并且将每个线程分配到靠

近父线程所绑定处理器核心位置的处理器核心。为了充分利用处理器的并行能力,使用与处理器核心(core)数相同的 OpenMP 线程数。基于以上策略将每个 OpenMP 线程逐一绑定到不同的处理器核心(第 1 行)。

2)数据分配:线程绑定完成后,每个 NUMA 节点上运行的线程已经确定。因此,每个节点所需的样本点数据同样是明确的。OpenMP 在 NUMA 节点之间提供了一种名为 First Touch Policy 的数据分配方法,当线程第一次访问某个样本点数据时,会将该样本点数据所在的页分配到距离该线程最近的内存中。所以在阴阳 *K*-means算法开始之前,可以并行地对所有的样本点数据进行一次初始化(第 2 行),从而每个线程处理的样本点数据都存放在该线程所属节点的本地内存中。

3)工作负载分配:将所有的样本点处理任务均匀地分成 m 个部分,然后静态地分配给 m 个 OpenMP 线程(第 8 行),其中 m 大小等于众核处理器中计算核心的数量。

完成以上三个步骤后,每个线程都绑定了一个处理器核心,且每个线程需要的样本点数据都放在该处理器核心所属 NUMA 节点的本地内存中。因此消除了各个 NUMA 节点对样本点数据的远程内存访问,有效地降低了内存访问开销。

3.3　内存数据布局优化

假设有 n 个样本点,每个样本点有 d 个维度,这些样本点在内存中最常见的数据布局方式如图3所示,样本点与样本点以及单个样本点的每个维度之间都是顺序存储的。但是,当使用式(1)计算多个相邻样本点到一个中心点的欧几里得距离时,由于基于样本点数据进行向量化时需要访问多个样本点的相同维度来构造一个向量,经典内存数据布局将带来一个明显的缺点:相邻样本点的相同维度数据不连续,向量化存储访问开销较大。

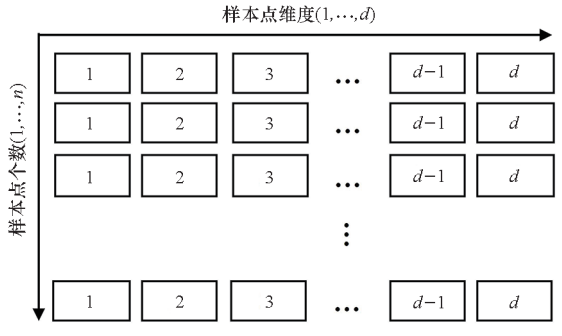

图 3　经典内存数据布局

Fig. 3　Classic memory data layout

如图 4 所示,为了使两个样本点的相同维度的内存访问连续,可以直观地将所有样本点的相同维度聚合在一起。但是,当访问样本点的下一个维度时,聚合内存数据布局将面临两次内存访问之间的跨度太大的问题,数据的空间局部性较差。并且,在进行 NUMA 亲和性优化时该内存数据布局不利于将样本点数据分配到各个 NUMA 节点。

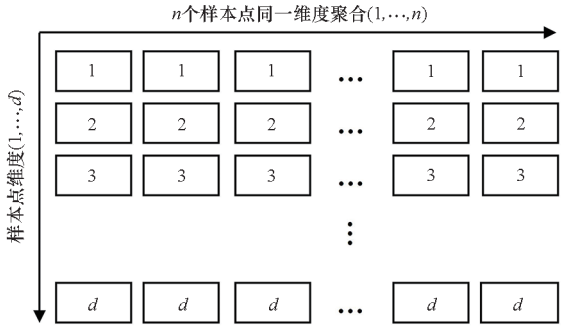

图 4　聚合内存数据布局

Fig. 4　Aggregated memory data layout

为了提高数据访问局部性以及便于在各个 NUMA 节点间分配样本点数据,提出了图 5 所示的分组聚合内存数据布局。以 ARMv8 架构处理器的 128 位向量处理单元为例,将两个样本点的数据划分为一组,并在组内使用聚合内存布局。至此,无论是访问两个样本点的相邻维度还是相邻的一组样本点,内存访问都是完全连续的。

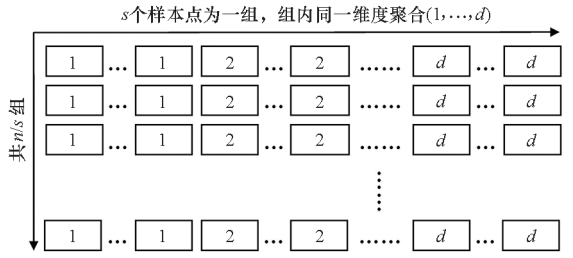

图 5　分组聚合内存数据布局

Fig. 5　Group aggregated memory data layout

4　实验

4.1　实验设置

为了证明向量化和 NUMA 亲和性内存访问优化对阴阳 *K*-means 算法的有效性,实验使用四个大型的真实世界数据集,其中两个来自 UCI 机器学习数据库[29],另外两个来自 Kaggle[30]。表 1 列出了四个数据集的参数。

表1 真实数据集参数

Tab. 1 Parameters of the real-world datasets

数据集	样本点数量 n	维度 d
YearPrediction(YP)	515 345	90
BotnetAttackDetection (BAD)	555 932	115
WifiWithPCA(WWP)	258 125	252
OGBNProducts (OP)	2 449 029	100

实验使用以下两种阴阳 *K*-means 算法的实现:

1)YYG:只使用全局过滤器的阴阳 *K*-means 算法。

2)YYGG:使用全局过滤器和组过滤器的阴阳 *K*-means 算法。

此外,还使用一些缩写来表示应用不同优化方法的 YYG(G)实现:

1)YYG(G)-SV:标量阴阳 *K*-means 算法在 CPU 上的多线程开源实现[25]。

2)YYG(G)-VV:仅采用向量化优化的多线程阴阳 *K*-means 算法。

3)YYG(G)-NUMA:同时采用了向量化优化和 NUMA 亲和性优化的多线程阴阳 *K*-means 算法。

所有代码都是用 C/C++编写,使用 GCC 编译器编译,并使用 -O3 级别的编译优化以及 Intrinsic 指令来实现向量化。算法的收敛条件是每个簇的中心点位置不再变化或迭代次数达到 1 000。在中心点数量 $k \in \{64,128,256,512\}$ 的条件下使用四个数据集进行了实验,每种情况下算法的性能测试重复三次,结果取平均值。所有数据都存储为双精度浮点值。实验使用以下三个平台:

1)Phytium FT-2000+平台:由一块 Phytium FT-2000+ARMv8 CPU 构成,有 8 个 NUMA 节点,64 个核心,核心的主频为 2.3 GHz,每个核心配备一个硬件线程、32 KB 私有 L1 指令缓存以及 32 KB私有 L1 数据缓存。每 4 个核心共享 2 MB 的 L2 缓存[22]。

2)Marvell ThunderX 平台:两块 Marvell ThunderX ARMv8 CPU 构成一个 NUMA 系统,每块 CPU 配备 48 个 2.0 GHz 的核心,每个核心配备一个硬件线程、78 KB 的 L1 指令缓存和 32 KB 的 L1 数据缓存。48 个核心共享 16 MB 的 L2 缓存[31]。

3)Intel Xeon Gold 6240 平台:两块 CPU 构成一个 NUMA 系统,每块 CPU 配备 18 个2.6 GHz的核心,每个核心配备两个硬件线程,并分别配备 576 KB 的 L1 指令缓存和数据缓存。18 个核心共享 18 MB 的 L2 缓存、24.75 MB 的 L3 缓存。支持 AVX-512 向量指令集[32]。

4.2 向量化性能

为了验证向量化的效果,对 YYG(G)-SV 和 YYG(G)-VV 这两个算法进行了多次比较实验。同时为了在 NUMA 节点上获得稳定的性能,执行 YYG(G)-SV 和 YYG(G)-VV 算法时使用了 Linux 命令 numactl--interleave = all 来使所有数据均匀分布到各个节点。Phytium、Marvell 和 Intel 三个平台的向量化优化实验结果所表现出的特征类似,分别最高获得了约 4.6、1.9 以及 8.5 的性能加速比。考虑到篇幅原因,本小节仅以 Phytium FT-2000+为例进行详细分析。

表2 展示了 FT-2000+上向量化阴阳*K*-means 算法相对于标量阴阳 *K*-means 算法的加速比。

表2 Phytium FT-2000+平台上向量化 YYG(G)算法相对于标量 YYG(G)算法的加速比

Tab. 2 Speedup of vectorized version of YYG(G) over scalar version on Phytium FT-2000+

k	算法	YP	BAD	WWP	OP
64	YYG-VV/YYG-SV	2.65	3.68	2.95	3.70
	YYGG-VV/YYGG-SV	0.60	1.04	1.04	1.00
128	YYG-VV/YYG-SV	2.67	4.22	3.24	3.66
	YYGG-VV/YYGG-SV	**0.57**	1.21	1.20	0.89
256	YYG-VV/YYG-SV	3.34	4.54	3.65	4.50
	YYGG-VV/YYGG-SV	0.61	1.31	1.25	1.03
512	YYG-VV/YYG-SV	3.47	**4.61**	3.90	3.93
	YYGG-VV/YYGG-SV	0.63	1.33	1.35	1.00

由表可概括出如下几个关键特征:

1)向量化优化效果明显:在 FT-2000+上,大多数情况下向量化算法的性能都优于标量算法,

向量化算法的加速比最高达到约 4.6。

2)向量化优化效果 YYG 算法明显强于 YYGG 算法:可以从两个方面来解释,一方面,前者只有全局过滤器,因此算法中的逻辑判断语句较少;另一方面,后者筛选了部分中心点,进一步减少了样本点与中心点的距离计算次数,导致计算量进一步降低,所以向量化之后的性能提升相对较小。

3)超线性加速比:当不存在 NUMA 架构的影响时,FT-2000 + 上 ARMv8 的 128 位向量单元对应的向量化理论加速比是 2。实验结果显示某些情况下向量化加速比远超过了 2,主要是原始标量算法与优化向量算法受到 NUMA 架构的影响不一样导致的。为了屏蔽 NUMA 架构对向量化优化效果的测量,本文又在 FT-2000 + 的单个 NUMA 节点上进行了实验,所有情况下向量化加速比都恢复到了小于 2 的正常范围。因此,超线性加速比表明向量化算法比标量算法受 NUMA 架构的影响更小。

4)低维度数据集实验效果不佳:在实验中观察到向量化算法在数据集 YP 上的性能比标量算法差。向量化在这些情况下不起作用的主要原因是该数据集的维度相对较低,从而计算量较小,指令流水线加载和排空的时间占总时间的比例较大,导致向量化性能不佳。因此,向量化更适合高维度的数据集。根据本文的测试结果显示,使用维度超过 100 的数据集的阴阳 *K*-means 算法在向量化后可以获得良好的性能。

5)加速比随 k 值的增大而增大:实验结果表明,随着 k 值的增大,加速比整体呈上升趋势。对于向量化算法来说,k 值增大意味着中心点数量增多,从而样本点与中心点之间的距离计算显著增多,程序并行部分运行时间占总时间的比例增大,处理器数量不变,根据 Amdahl 定律可知加速比增大。

4.3　NUMA 亲和性优化性能

为了评估 NUMA 亲和性内存访问优化带来的性能提升,将 YYG(G)-NUMA 算法与 YYG(G)-VV进行性能比较。实验结果显示,当 NUMA 节点数较少时,如 Intel 平台、Marvell 平台,以及仅使用两个 NUMA 节点的 Phytium 平台,NUMA 亲和性优化效果甚微。但是,随着 NUMA 节点数的增加,NUMA 亲和性优化性能显著增加。

表 3 和表 4 分别展示了同时使用向量化和 NUMA 亲和性优化的 YYG(G)算法相对于仅使用向量化优化的 YYG(G)算法在 8 节点和 4 节点的 Phytium 平台上的加速比。使用 8 节点的 FT-2000 + 处理器时,YYG(G)-NUMA 在四个数据集上的性能始终优于 YYG(G)-VV,加速比为 1.11 ~ 1.51。而对于使用 4 节点的 FT-2000 + 而言,NUMA 优化的加速比略低于 8 节点,最高只达到了 1.34。因此,在 NUMA 节点数较多时,通过 NUMA 亲和性内存访问优化策略,可以有效地避免大量远程内存访问,实现性能提升。此外,随着 k 值的增大,加速比整体呈下降趋势,k 值的增大意味着中心点数量增加,由于中心点只保存在主节点内存中,且中心点被所有 NUMA 节点共享,所以中心点数据增多导致其他 NUMA 节点访问主节点的内存访问开销占比增大,从而加速比下降。

表 3　Phytium FT-2000 + 平台上具有 NUMA 亲和性的向量化 YYG(G)相对于向量化 YYG(G)的加速比(8 节点)

Tab.3　Speedup of NUMA-aware vectorized version of YYG(G) over vectorized version on Phytium FT-2000 + (8 nodes)

k	算法	YP	BAD	WWP	OP
64	YYG-NUMA/YYG-VV	**1.51**	1.42	1.44	1.29
	YYGG-NUMA/YYGG-VV	1.38	1.44	1.47	1.24
128	YYG-NUMA/YYG-VV	1.51	1.28	1.38	1.20
	YYGG-NUMA/YYGG-VV	1.34	1.30	1.36	1.24
256	YYG-NUMA/YYG-VV	1.32	1.20	1.29	**1.11**
	YYGG-NUMA/YYGG-VV	1.34	1.20	1.29	1.14
512	YYG-NUMA/YYG-VV	1.27	1.21	1.22	1.13
	YYGG-NUMA/YYGG-VV	1.12	1.19	1.22	1.12

4.4　整体优化性能

结合向量化优化和 NUMA 亲和性优化,YYG(G)-NUMA 算法在 Phytium 平台、Marvell 平台以及 Intel 平台上相对于标量算法的整体加速比分别如表 5、表 6 和表 7 中所示。当数据集的维度比较高时,并行加速阴阳 *K*-means 算法相比传统阴阳 *K*-means 算法能实现较大的性能提升,

表4 Phytium FT-2000+平台上具有NUMA亲和性的向量化YYG(G)相对于向量化YYG(G)的加速比(4节点)

Tab.4 Speedup of NUMA-aware vectorized version of YYG(G) over vectorized version on Phytium FT-2000+(4 nodes)

k	算法	YP	BAD	WWP	OP
64	YYG-NUMA/YYG-VV	**1.34**	1.21	1.25	1.14
	YYGG-NUMA/YYGG-VV	1.30	1.21	1.27	1.11
128	YYG-NUMA/YYG-VV	1.22	1.21	1.24	1.10
	YYGG-NUMA/YYGG-VV	1.23	1.19	1.19	1.11
256	YYG-NUMA/YYG-VV	1.20	1.14	1.16	1.09
	YYGG-NUMA/YYGG-VV	1.19	1.15	1.14	1.10
512	YYG-NUMA/YYG-VV	1.14	1.10	1.12	**1.05**
	YYGG-NUMA/YYGG-VV	1.15	1.08	1.12	1.06

表5 Phytium FT-2000+平台上具有NUMA亲和性的向量化YYG(G)相对于标量YYG(G)的加速比

Tab.5 Speedup of NUMA-aware vectorized version of YYG(G) over scalar version on Phytium FT-2000+

k	算法	YP	BAD	WWP	OP
64	YYG-NUMA/YYG-SV	3.99	5.23	4.24	4.78
	YYGG-NUMA/YYGG-SV	0.82	1.50	1.53	1.24
128	YYG-NUMA/YYG-SV	4.02	5.38	4.49	4.41
	YYGG-NUMA/YYGG-SV	0.76	1.56	1.63	1.11
256	YYG-NUMA/YYG-SV	4.42	5.43	4.72	5.00
	YYGG-NUMA/YYGG-SV	0.81	1.58	1.62	1.17
512	YYG-NUMA/YYG-SV	4.38	**5.59**	4.75	4.46
	YYGG-NUMA/YYGG-SV	**0.71**	1.58	1.64	1.11

表6 Marvell ThunderX平台上具有NUMA亲和性的向量化YYG(G)相对于标量YYG(G)的加速比

Tab.6 Speedup of NUMA-aware vectorized version of YYG(G) over scalar version on Marvell ThunderX

k	算法	YP	BAD	WWP	OP
64	YYG-NUMA/YYG-SV	1.20	1.88	1.50	1.33
	YYGG-NUMA/YYGG-SV	1.00	1.79	1.45	1.24
128	YYG-NUMA/YYG-SV	1.08	1.94	1.52	1.20
	YYGG-NUMA/YYGG-SV	1.00	1.86	1.50	1.14
256	YYG-NUMA/YYG-SV	1.19	1.97	1.55	1.36
	YYGG-NUMA/YYGG-SV	0.96	1.88	1.50	1.20
512	YYG-NUMA/YYG-SV	1.17	**2.00**	1.54	1.18
	YYGG-NUMA/YYGG-SV	**0.95**	1.87	1.51	1.14

表7 Intel Xeon平台上具有NUMA亲和性的向量化YYG(G)相对于标量YYG(G)的加速比

Tab.7 Speedup of NUMA-aware vectorized version of YYG(G) over scalar version on Intel Xeon

k	算法	YP	BAD	WWP	OP
64	YYG-NUMA/YYG-SV	4.27	6.90	4.47	4.60
	YYGG-NUMA/YYGG-SV	0.87	2.96	2.46	1.65
128	YYG-NUMA/YYG-SV	3.94	7.77	5.15	4.95
	YYGG-NUMA/YYGG-SV	**0.82**	3.40	3.08	1.58
256	YYG-NUMA/YYG-SV	5.81	7.95	5.65	6.66
	YYGG-NUMA/YYGG-SV	0.99	3.41	3.16	1.92
512	YYG-NUMA/YYG-SV	5.43	**8.68**	5.85	4.87
	YYGG-NUMA/YYGG-SV	1.01	3.39	3.26	1.73

这表明本文的优化方法可以有效地提高阴阳 *K*-means算法在众核处理器上的性能,在 Phytium 平台、Marvell 平台以及 Intel 平台上最高可分别获得约 5.6、2.0 和 8.7 的加速比。

对比测试结果可知,本文算法在 Intel x86 平台上的收益显著高于 Phytium 和 Marvell 两个 ARMv8 平台,主要原因有两点:一是 Intel Xeon 处理器配备了 512 位向量处理单元,相对于 ARMv8 架构 128 位向量处理单元,其向量宽度更宽,因此向量化优化收益更大;二是 Intel 平台每个核心配备了更大的 L1 缓存、L2 缓存以及独有的 L3 缓存,使得本文数据布局优化的效果更加明显,从而提升了整体优化方案的效果。

4.5　扩展性

除上述实验外,在 Phytium 平台上以数据集 WWP 为例设计了一组实验来研究阴阳 *K*-means 算法并行加速实现的强扩展性。分别测试了并行加速阴阳 *K*-means 算法和传统阴阳 *K*-means 算法从单个 NUMA 节点扩展到 8 个 NUMA 节点时的强扩展效率,实验结果如图 6 和图 7 所示。图 6 表明向量化和 NUMA 亲和性优化都提高了 YYG 的扩展效率,YYG-SV 的扩展效率经过向量化后从 30% 提升到 60% 以上,经过 NUMA 亲和性优化后进一步提升到 85% 以上。图 7 中传统的 YYGG-SV 算法的扩展效率已经高达 90% 以上,

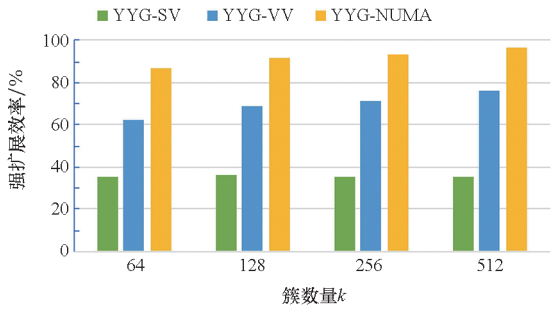

图 6　YYG 算法的扩展性

Fig. 6　Strong scaling of YYG algorithm

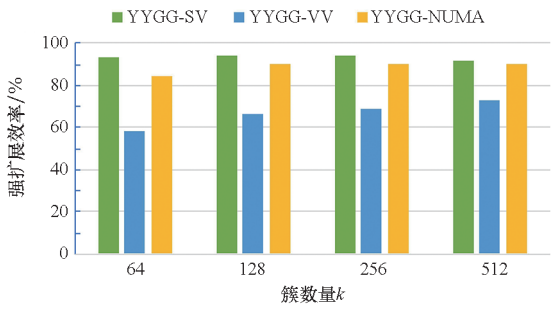

图 7　YYGG 算法的扩展性

Fig. 7　Strong scaling of YYGG algorithm

向量化大幅度提升了算法在单 NUMA 节点上的性能,从而计算出的扩展效率下降,但经过 NUMA 亲和性优化后,扩展效率恢复到仅使用多线程技术实现的水平。因此,并行加速阴阳 *K*-means 算法依然具有较好的扩展性。

5　结论

本文使用向量化和 NUMA 亲和性内存访问优化对众核处理器上的阴阳 *K*-means 算法进行了优化,同时为了获得更好的数据局部性,还提出了一种新的内存数据布局来支持上述优化。实验结果表明,使用高维数据集时,向量化可以有效地提高阴阳 *K*-means 算法在众核处理器上的性能;NUMA 亲和性内存访问优化在具有较多 NUMA 节点的 NUMA 系统上可以进一步提升阴阳 *K*-means算法的性能。在 ARMv8 和 x86 众核平台上,本文提出的阴阳 *K*-means 算法的并行加速实现最终分别获得了最高约 5.6 倍与 8.7 倍的加速。

未来的研究拟将本文的阴阳 *K*-means 并行加速实现扩展到分布式系统中,以实现更大规模的并行。

参考文献(References)

[1] LLOYD S. Least squares quantization in PCM[J]. IEEE Transactions on Information Theory, 1982, 28(2): 129 – 137.

[2] ARTHUR D, VASSILVITSKII S. k-means++: the advantages of careful seeding[C]//Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 2007.

[3] KANUNGO T, MOUNT D M, NETANYAHU N S, et al. An efficient *k*-means clustering algorithm: analysis and implementation[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2002, 24(7): 881 – 892.

[4] XIA S Y, PENG D W, MENG D Y, et al. Ball k-means: fast adaptive clustering with no bounds[J]. IEEE transactions on pattern analysis and machine intelligence, 2022, 44(1): 87 – 99.

[5] DRAKE J, HAMERLY G. Accelerated *k*-means with adaptive distance bounds[C]//Proceedings of 5th NIPS Workshop on Optimization for Machine Learning, 2012, 8: 1 – 4.

[6] HAMERLY G. Making *k*-means even faster[C]//Proceedings of the 2010 SIAM International Conference on Data Mining, 2010.

[7] MILANOV D V, MILANOVA Y V, KHOLSHEVNIKOV K V. Relaxed triangle inequality for the orbital similarity criterion by Southworth and Hawkins and its variants[J]. Celestial Mechanics and Dynamical Astronomy, 2019, 131: 5.

[8] DING Y F, ZHAO Y, SHEN X P, et al. Yinyang K-means: a drop-in replacement of the classic K-means with consistent speedup[C]//Proceedings of the 32nd International

Conference on Machine Learning, 2015.

[9] WU F H, WU Q B, TAN Y S, et al. A vectorized K-means algorithm for intel many integrated core architecture[M]// Lecture Notes in Computer Science. Berlin: Springer Berlin Heidelberg, 2013: 277 – 294.

[10] KWEDLO W, CZOCHANSKI P J. A hybrid MPI/OpenMP parallelization of *K*-means algorithms accelerated using the triangle inequality[J]. IEEE Access, 2019, 7: 42280 – 42297.

[11] ZHAO W Z, MA H F, HE Q. Parallel *K*-means clustering based on MapReduce [M]//Lecture Notes in Computer Science. Berlin: Springer Berlin Heidelberg, 2009: 674 – 679.

[12] KUMAR J, MILLS R T, HOFFMAN F M, et al. Parallel *k*-means clustering for quantitative ecoregion delineation using large data sets[J]. Procedia Computer Science, 2011, 4: 1602 – 1611.

[13] BHIMANI J, LEESER M, MI N F. Accelerating K-means clustering with parallel implementations and GPU computing[C]// Proceedings of IEEE High Performance Extreme Computing Conference (HPEC), 2015.

[14] ZECHNER M, GRANITZER M. Accelerating k-means on the graphics processor via CUDA [C]//Proceedings of First International Conference on Intensive Applications and Services, 2009.

[15] FARIVAR R, REBOLLEDO D, CHAN E, et al. A parallel implementation of K-means clustering on GPUs [C]// Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, 2008.

[16] HUSSAIN H M, BENKRID K, SEKER H, et al. FPGA implementation of K-means algorithm for bioinformatics application: an accelerated approach to clustering Microarray data [C]//Proceedings of NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2011.

[17] DIAS L A, FERREIRA J C, FERNANDES M A C. Parallel implementation of K-means algorithm on FPGA[J]. IEEE Access, 2020, 8: 41071 – 41084.

[18] TAYLOR C, GOWANLOCK M. Accelerating the Yinyang *k*-means algorithm using the GPU[C]//Proceedings of IEEE 37th International Conference on Data Engineering (ICDE), 2021.

[19] Intel. Accelerate your compute-intensive workloads: Intel® advanced vector extensions 512 (Intel® AVX – 512)[EB/OL]. [2022 – 07 – 14]. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.

[20] ARM. Neon [EB/OL]. [2022 – 07 – 14]. https://developer.arm.com/Architectures/Neon.

[21] AMD. Meet the AMD EPYC™ server processor family[EB/OL]. [2022 – 07 – 14]. https://www.amd.com/en/processors/epyc-server-cpu-family.

[22] 飞腾. FT-2000 +/64 高性能服务器 CPU[EB/OL]. [2022 – 07 – 14]. https://www.phytium.com.cn/homepage/production/3/.
Phytium. FT-2000 +/64 high performance server CPU[EB/OL]. [2022 – 07 – 14]. https://www.phytium.com.cn/homepage/production/3/. (in Chinese)

[23] 王庆林, 李东升, 梅松竹, 等. 面向飞腾多核处理器的 Winograd 快速卷积算法优化[J]. 计算机研究与发展, 2020, 57(6): 1140 – 1151.
WANG Q L, LI D S, MEI S Z, et al. Optimizing Winograd-based fast convolution algorithm on Phytium multi-core CPUs[J]. Journal of Computer Research and Development, 2020, 57(6): 1140 – 1151. (in Chinese)

[24] HUANG X D, WANG Q L, LU S Y, et al. Evaluating FFT-based algorithms for strided convolutions on ARMv8 architectures[J]. Performance Evaluation, 2021, 152: 102248.

[25] Taylor. Yinyang k-means in CUDA[EB/OL]. [2022 – 07 – 14]. https://github.com/ctaylor389/k_means_yinyang_gpu.

[26] NEWLING J, FLEURET F. Fast *k*-means with accurate bounds [C]//Proceedings of the 33rd International Conference on International Conference on Machine Learning, 2016.

[27] WANG Q L, LI D S, HUANG X D, et al. Optimizing FFT-based convolution on ARMv8 multi-core CPUs[M]//Euro-Par 2020: Parallel Processing. Cham: Springer International Publishing, 2020: 248 – 262.

[28] OpenMP. Reference guides[EB/OL]. [2022 – 07 – 14]. https://www.openmp.org/resources/refguides/.

[29] UC Irvine. Welcome to the UC Irvine machine learning repository[EB/OL]. [2022 – 07 – 14]. http://archive.ics.uci.edu/ml.

[30] Kaggle. Datasets[EB/OL]. [2022 – 07 – 14]. https://www.kaggle.com/datasets.

[31] WikiChip. ThunderX-Cavium[EB/OL]. [2022 – 07 – 14]. https://en.wikichip.org/wiki/cavium/thunderx.

[32] Intel. Intel© Xeon© Gold 6240 processor[EB/OL]. [2022 – 07 – 14]. https://www.intel.com/content/www/us/en/products/sku/192443/intel-xeon-gold-6240-processor-24-75m-cache-2-60-ghz/specifications.html

(编辑: 熊立桃, 杨琴)