ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.21, No.6, June 2010, pp.1220−1232
doi: 10.3724/SP.J.1001.2010.03575

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于共享 Cache 多核处理器的 Hash 连接优化*

邓亚丹+, 景 宁, 熊 伟

(国防科学技术大学 电子科学与工程学院,湖南 长沙 410073)

Hash Join Query Optimization Based on Shared-Cache Chip Multi-Processor

DENG Ya-Dan+, JING Ning, XIONG Wei

(College of Electronic Science and Engineering, National University of Defense Technology, Changsha 410073, China)

+ Corresponding author: E-mail: dengyadan2008@yahoo.com.cn

Deng YD, Jing N, Xiong W. Hash join query optimization based on shared-cache chip multi-processor. *Journal of Software*, 2010,21(6):1220−1232. http://www.jos.org.cn/1000-9825/3575.htm

Abstract: This paper presents hash join optimization based on shared cache CMP (chip multi-processor). Firstly, it proposes a multithreaded execution framework of hash join based on Radix-Join algorithm, and then analyzes the factors which affect the performance of multithreaded Radix-Join algorithm through two instances. Based on the analysis, the performance of various threads and their shared-cache access behaviors in the hash join multithreaded execution framework were optimized, and optimize memory access of hash join in cluster join phase. It then analyzes the speedup of multithreaded cluster partition in theory was analyzed. All of the algorithms are implemented in the INGRES and EaseDB. In the experiments, the performance of the multithreaded execution framework of hash join is tested, and the results show that the proposed algorithm could effectively resolve the cache access conflict and load balance of CMP cores in multithreaded environment and hash join performance is improved.

Key words: hash join; Radix-Join; chip multi-processor; shared cache; multithread performance analysis

摘 要: 针对目前主流的多核处理器,研究了基于共享缓存多核处理器环境下的数据库 Hash 连接优化.首先提出基于 Radix-Join 算法的 Hash 连接多线程执行框架,通过实例分析了影响多线程 Radix-Join 算法性能的因素.在此基础上,优化了 Hash 连接多线程执行框架中的各种线程及其访问共享 Cache 的性能,优化了聚集连接时 Hash 连接算法的内存访问,并分析了多线程聚集划分的加速比.基于开源数据库 INGRES 和 EaseDB,实现了所提出的连接多线程执行框架,在实验中测试了多线程 Hash 连接框架的性能.实验结果表明,该算法可以有效解决 Hash 连接执行时共享 Cache 在多线程条件下的访问冲突和处理器负载均衡问题,极大地提高了 Hash 连接性能.

关键词: Hash 连接;Radix-Join;多核处理器; 共享 Cache;多线程性能分析
中图法分类号: TP316 文献标识码: A

* Supported by the National Natural Science Foundation of China under Grant No.40801160 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant Nos.2007AA120400, 2007AA12Z208, 2006AA12Z205 (国家高技术研究发展计划(863))

Received 2008-09-01; Revised 2008-11-27; Accepted 2009-01-15

随着处理器晶体管数量的不断增加,处理器频率已达到现有条件下的极限,且能耗越来越大.所以,处理器的发展趋势正从高主频的单核处理器转向片上多核处理器(chip multi-processor,简称 CMP)[1],从依靠指令级的并行转向线程级并行.目前,2 核和 4 核的处理器正成为市场的主流,而且内核数量仍在不断增加,极大地提高了单处理器的并行计算性能.相比较多核处理器普及之前,并行计算正成为主要的程序运行模式,并日益受到重视.与传统的 SMP(symmetrical multi-processing)系统相比,CMP 不但从硬件成本上更易被采纳,而且由于芯片内的通信速度大大高于芯片间,CMP 并行效率要高于 SMP 系统.对数据库而言,共享 Cache 可以减少数据重复,提高 Cache 利用率.但也正由于 CMP 一般共享 L2-Cache,造成了多线程之间比较容易发生共享 Cache 访问冲突,造成性能下降.所以,CMP 的出现给数据库系统既带来了机遇也带来了挑战[2].很多学者[2−5]展开了基于 CMP 的数据库优化研究.相对于查询之间的并行优化,查询内部的并行性还有待进一步研究[5],而且随着处理器核心数量的不断增加,特别是在多处理器系统中,查询内部的并行性所能带来的性能提升日益增大.

与此同时,目前的计算机体系结构中一般采用多缓存体系结构减少数据访问延迟.越来越多的研究表明,在多缓存体系结构中,随着内存价格不断下降和内存容量的不断增大,大容量内存数据库服务器正成为主流配置,磁盘 I/O 延迟逐渐得到缓解[1],Cache 访问性能成为影响数据库性能的重要因素.而内存访问速度相对于 Cache 和寄存器差距明显,并且仍在不断扩大.所以,处理器等待 Cache 从上一级存储器取数据造成的延迟正成为数据库系统的主要瓶颈[2,6−11],例如 L2-Cache 和 L1-DataCache.

目前,CMP 一般共享 L2-Cache,可以有效地减少数据重复,提高 Cache 利用率.但由于争用共享资源,对多线程执行会造成一些负面影响,主要为 Cache 访问冲突[12].即多个内核可以同时访问共享 Cache,Core 1 可能将其他 Core 所需的 Cache Line 交换出 Cache,造成其他 Core 访问这些 Cache Line 时出现 Cache 不命中,甚至出现 Cache Thrashing 现象.因此,要充分利用共享 Cache 多核处理器的性能优势,就必须有效解决因 Cache 访问冲突造成的性能瓶颈,同时需要使得处理器各个核心尽量达到负载均衡.

本文第 2 节介绍基于 CMP 的 Hash 连接多线程框架.第 3 节给出 Hash 连接多线程框架中各种线程和内存访问的优化.第 4 节是实验与分析.最后为结论与展望.

1 相关研究工作

目前,数据库通过优化 Cache 访问和线程级并行达到减少延迟或隐藏 Cache 延迟的目的.针对 Hash 连接,已有不少学者提出优化算法,分为两类:

(1) 优化 Cache 访问性能.此类方法主要分为预取[9]和划分[13,14].文献[9]利用软件预取指令隐藏访问延迟,提出了两种预取技术:组预取和流水线技术.文献[13]提出了 Radix-Join 算法,通过在聚集过程中采用多路划分减少聚集划分过程中的 Cache 访问缺失,被作为基本的 Hash 连接算法用于 Hash 连接的优化研究[10,14],本文亦基于 Radix-Join 展开研究工作.文献[14]给出了一种 Cache 优化技术,通过运行时获取最佳 Cache 参数,在某些情况下可以获得比采用 Cache 参数更好的优化效果.此外,文献[15]研究了多线程处理器环境下 Hash 连接的性能,通过优化内存拷贝提高 L1-Cache 的性能;

(2) 线程级并行优化.在片上多线程处理器之前,许多学者针对多处理器系统,比如 SMP,利用并行多线程技术优化 Hash 连接的性能.文献[16]提出了基于对称多处理器系统的多连接执行优化,该优化算法基于动态编程和贪婪策略来确定多连接在 SMP 中的最佳执行方式.文献[17]则通过优化处理器资源分配和 Hash 过滤器来改善并行 Hash 连接的性能.文献[8]基于共享内存的多处理器系统,比较了 3 类常见 Hash 连接算法的并行化后的性能,并分析了代价模型.处理器进入片上多线程时代后,许多学者纷纷展开基于其上的 Hash 连接优化.文献[10]在 MTA-2 多核处理器中实现了 Radix-Join 算法,在大量线程并发执行时,Hash 连接的加速比和吞吐量都有一定提高.文献[11]提出在 SMT(simultaneous multithreading)处理器中将元组分为奇偶两类在两个线程之间划分,实现并行执行数据库操作;还提出利用主线程和 Helper 线程模式优化 Data Cache.实验结果表明,Hash 连接的性能有一定程度的提高.文献[5]将 Hash 连接分成若干操作,操作之间按照流水线的方式执行,每个操作由多个线程同时执行.

上述研究工作中,文献[16,17]都是在多核处理器问世或普及之前,基于单核多处理器系统展开,因此不存在线程之间的 Cache 访问冲突问题.而且,当时内存和处理器之间的速度差异还未有目前严重,Cache 访问性能问题并不突出.文献[9,14]的 Cache 优化未针对多核处理器,也未考虑多线程的情况.文献[10,15]侧重于验证 Hash 连接在多线程环境下的性能,未研究如何高效利用共享 Cache.文献[5]也未研究如何充分利用共享 L2-Cache.文献[11]虽然优化了 Data Cache 访问,但 SMT 处理器一般只能同时执行两个线程的指令,而且部分处理器执行部件也是共享使用的,并非真正的同时执行多个线程.所以,文献[11]只设置了两个可并行执行的线程,这种方式不能充分发挥 CMP 处理器的优势;而且,CMP 正逐渐取代 SMT 处理器,成为片上多线程处理器技术发展的方向.

综上所述,虽然 Hash 连接优化的研究成果很多,但目前尚未有在多核处理器硬件条件下,特别是充分发挥共享 L2-Cache 优势、优化 Hash 连接的研究工作.而且本文基于多核处理器优化了 Radix-Join 算法,目前尚未有类似的研究工作.

2 Hash 连接多线程框架

本文主要研究共享 L2-Cache 多核处理器条件下,优化 Hash 连接的多线程和多线程的 L2-Cache 访问性能,所以更适用于内存数据库,或者基于磁盘的数据库其内存较大、页面缓存能够缓存足够多的数据页面极大地减少了 I/O 延迟,内存访问成为其主要代价.下面介绍 Hash 连接多线程框架的细节,它以 Radix-Join 算法[13]为基础.Radix-Join 算法从最左端位开始,处理 Hash 值的 $D=\sum_1^P D_p$ 位数据,通过 P 次划分将表分成 $H=\prod_1^P H_p$ 个聚集,每一次聚集划分生成 $H_i=2^{D_i}$ 个新聚集,而两个表的聚集之间采用简单 Hash 连接或嵌套循环连接算法,本文则只针对采用简单 Hash 连接的情况.Radix-Join 算法分为聚集划分和聚集连接两个阶段.聚集划分阶段的多线程执行框架如图 1 所示.

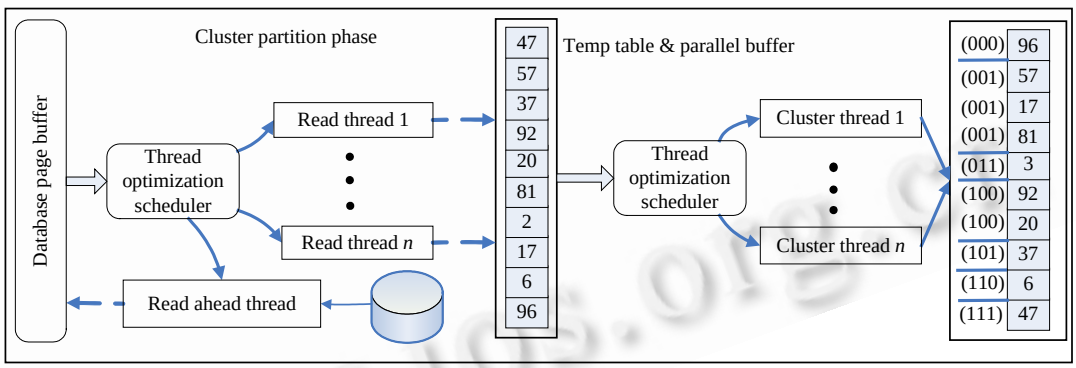

Fig.1 Cluster partition multithreaded framework

图 1 聚集划分多线程执行框架

图 1 的临时表采用文献[3]提出的 Parallel Buffer 进行内存访问和管理,如图 2 所示.整块内存分成若干个 Chunk,每个线程以 Chunk 为单位读/写内存,可以极大地降低互斥访问代价.其中,Thread Optimization Scheduler 模块对应第 3.2 节的页面读取线程和聚集划分线程优化.图 1 中分为两类线程:

(1) 页面处理线程.在页面处理线程执行前,根据表的页面号范围,从页面缓存模块获取两个页面号集合:已在页面缓存中的页面集合和未在页面缓存中的页面集合.Prefech Thread(预取线程)从磁盘读取该表在页面缓存中不存在的页面,只需在基于磁盘的数据库中才需要使用.预取分为组读取(读取页面号连续的页面,*grNum* 表示一次组读取的页面数)和单页面读取两种模式.由于 INGRES 中每个表对应一个或多个文件,连续的页面号一般对应磁盘中连续的存储位置,所以 INGRES 中组读取相比较单文件多表存储结构的数据库,可以更好地降低磁盘 I/O 次数.预取线程先读取组读取页面,再读取单页面读取页面,将已读取的页面号放入一个全局队列,供 Read Thread(页面读取线程)处理.页面读取线程执行投影运算,将满足条件的元组生成 Hash Cell 存入临时表中.

Hash Cell 结构为⟨*HashValue*,*KeyValue*,*TID*⟩,其中,*HashValue* 为 Hash 值,*KeyValue* 为该表连接列的值,*TID* 为元组 ID 号.页面读取线程先处理由预取线程读取的页面,再处理 Hash 连接前就已经位于缓存中的页面.

(2) 聚集划分线程.表中所有页面处理完,Cluster Thread(聚集划分线程)才处理临时表,生成 Radix 聚集.聚集划分线程需根据情况决定在哪次聚集划分完成后启动线程,每个聚集划分线程分配一定数量的聚集.比如,聚集划分线程启动时已有 H_j 个聚集,启动 m 个聚集划分线程,在线程间平均分配这 H_j 个聚集.对于每个聚集,聚集划分线程完成余下的 $P–j$ 次聚集划分(详细分析见第 3.2.3 节).

聚集连接阶段的多线程执行框架如图 3 所示.其中,Thread Workload Allocation Scheduler 模块对应于第 3.2.4 节中的聚集连接线程优化.聚集划分生成了如图 3 所示的 *L* 和 *R* 两个聚集表.由 Join Thread(连接线程)同时对多个聚集执行连接运算,聚集连接线程之间需要减少 Cache 访问冲突,并达到处理器核心间的负载均衡.Output Buffer 为结果输出缓存,采用与临时表相同的访问和管理方式.

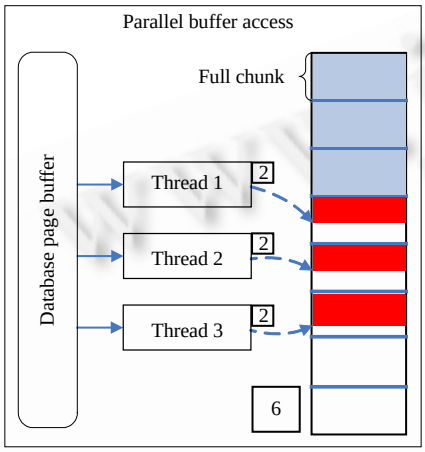

Fig.2 Parallel buffer structure

图 2 Parallel Buffer 结构

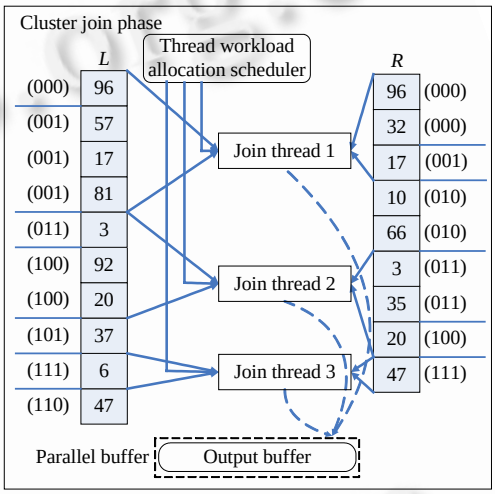

Fig.3 Cluster join multithreaded framework

图 3 聚集连接多线程执行框架

3 Hash 连接多线程框架优化

3.1 Radix-Join多线程性能比较及分析

下面通过两个实例的性能比较,分析共享 L2-Cache 多核处理器中影响多线程 Radix-Join 算法性能的因素,为以下的 Hash 连接多线程执行框架的优化提供依据.硬件为 1M 共享 Cache 双核处理器.数据库为 EaseDB,*L* 和 *R* 的元组数为 2 727 100 和 2 729 600,*Cellsize*=12B,数据在聚集间分布得比较均匀.设置 Radix-Join 参数 *P* 为(7,6,5,4),对应的 *ClusterSize* 值为(256kB,512kB,1024kB,2048kB).

例 1:聚集连接阶段执行时间比较,采用不用数量的聚集连接线程执行连接运算.

例 2:聚集划分和连接阶段执行时间比较,即聚集线程和连接线程的执行时间,采用不同数量线程执行聚集划分和聚集连接.

从多线程角度分析,CMP 中单线程程序的处理器利用率非常低,图 4 中,*ClusterSize*=1024KB 时,即使单线程的 Cache 访问冲突要大大低于多线程,但是执行时间却远大于多线程.而由于双核处理器只能同时运行两个线程,对同一个任务而言,比如聚集连接,程序的性能与线程数量不存在正比关系,当线程数大大超过处理器核心后,性能甚至会有所下降.但由于现代操作系统的线程执行片段一般为 10ms~20ms,DDR2 内存与片上 L2-Cache 之间的带宽一般可达 25G/s,即使考虑 Cache 命中和不命中的延迟代价,在一个线程执行片段内,线程所能处理的数据量也大大超过 L2-Cache 的容量.所以,线程数超过处理器核心数后,也不会出现严重 Cache 访问冲突导致

性能较大下降.图 4、图 5 中,线程数大于 N 且为奇数时,由于处理器核心之间负载不均衡,导致性能下降;但随着线程数增加,单个聚集数据量变小,负载不均衡的程度也随之降低,性能有所回升.

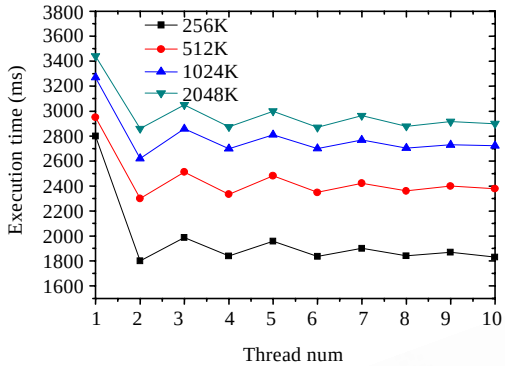

Fig.4 Cluster join performance comparison

图 4 聚集连接性能比较

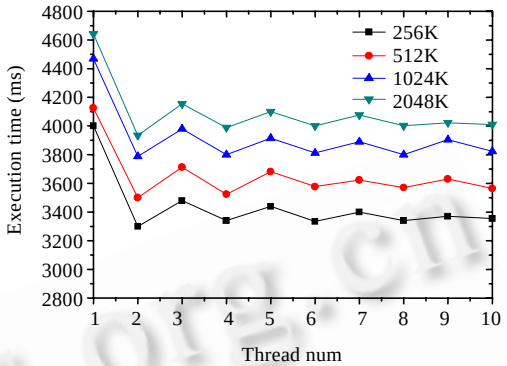

Fig.5 Cluster partition and join performance comparison

图 5 聚集划分和连接性能比较

从 Cache 角度分析,图 4 中 *ClusterSize*=256KB 时,两个连接线程对各自的聚集执行连接时,所需处理的数据大小接近 C,Cache 访问冲突较少;而当 *ClusterSize*=512KB 时,所需处理的数据大于 C,导致 Hash Table 对应的 Cache Line 被替换出 L2-Cache 的可能性大大增加,使得两者性能相差较大,表明 Cache 访问冲突对多线程性能影响很大.而 *ClusterSize*=1024KB 和 2048KB 时,虽然后者的 Cache 访问冲突更加严重,但性能却不会出现更严重的下降,表明当需要处理的数据大大超过 Cache 容量后,性能下降与数据量没有严格的正比关系.图 5 中,*ClusterSize*=256KB 时 P=7,虽然比 *ClusterSize*=512KB 时多一次临时表访问,但在 *ClusterSize*=512KB 时的 Cache 访问冲突更加严重,临时表访问上的优势被抵消;但当 *ClusterSize*=1024KB 和 2048KB 时,相对于 *ClusterSize*=512KB 时, Cache 访问冲突更加严重,即使它们的聚集划分次数更少,但总执行时间却更多.

3.2 Hash连接多线程执行框架优化

在共享 L2-Cache 多核处理器中,需要根据第 3.1 节中的性能比较和分析的结论.考虑共享 L2-Cache 对多线程执行的影响和各种线程的数据访问特点,在运行时决定线程执行的各种参数,比如线程数、线程启动时机、线程工作集分配等.本文提出的多线程框架中,页面读取、聚集划分和聚集连接采用多线程,具体分析如下.

3.2.1 Radix-Join 算法参数优化

文献[13]提出了聚集连接阶段采用 Hash 连接的参数 D 设置策略,分别针对 L1-Cache,L2-Cache 和 TLB 进行优化.由于本文主要针对共享 L2-Cache 的优化,而且其他两种设置在多核处理器中并未受到影响,所以我们关注 L2-Cache 的优化设置.文献[13]提出 $D=\log_2(L.tuple\times CellSize/C)$,未考虑多核处理器中多线程访问共享 L2-Cache 的冲突问题,而由第 3.1 节的分析可知会造成比较严重的 Cache 访问冲突.因此,需要加入聚集连接线程个数和处理器核心数的限制,$D=\log_2(2N\times L.tuple\times CellSize/C)$.使得聚集连接时,尽可能地使得连接线程访问的数据量小于等于 C.

3.2.2 页面读取线程优化

线程启动个数:从第 3.1 节的分析可知,页面读取线程处理预取线程读取的页面时,如果执行单个数据库操作的线程超过处理器核心数后,性能不但不会提高,甚至还可能有轻微下降.所以,Group Read 时线程数为 N–1 即可,Single Read 时为 N,其中,预取线程因读磁盘需要较多的运算时间,一般会耗尽每个时间片段.所以组读取时,为页面读取线程数为 N–1,避免与预取线程争抢处理器,Single Read 则无须考虑.同样,处理连接执行前已在缓存中的页面,启动 N 个页面读取线程即可.

线程启动时机设置:从磁盘读取页面时,整个页面被读入 L2-Cache,所以页面读取线程先处理预取线程读取

的页面,充分利用页面被完全读入 L2-Cache 的特点.

组读取优化:由于 INGRES 支持一次读取 8 个页面的组读取模式,所以 $grNum$=8.为了减少页面读取线程的启动次数,可令组读取或单页面读取的页面达到一定数量时再启动页面读取线程.因此,为了减少因访问临时表将预取的数据页面替换出 L2-Cache 造成的 Cache 访问冲突,需要设置页面读取线程最佳启动时机.令预取的数据页面达到 Max$Page$ 时启动页面读取线程,可得对应的页面读取线程需要访问的临时表数据量为 Max$page$×$M.tuple$×$CellSize$.因此,该次页面处理线程启动所需要访问的全部数据量 $Totalsize$=Max$page$×$M.tuple$×$CellSize$+Max$page$×$M.size$.令 $Totalsize$≤C,可得 $\text{Max}Page \le \frac{C}{M.size + M.tuple \times CellSize}$.

3.2.3 聚集划分线程优化

文献[13]提出每次聚集划分时的 D_i 一般大于 1,这需要通过交换表实现 Hash Cell 交换,数据量很大时可减少聚集划分次数,提高 Radix-Join 算法的整体性能.但由于需要额外的交换表,导致多线程之间的 Cache 访问冲突加剧.从图 5 的性能比较分析可知,严重的 Cache 访问冲突会抵消聚集次数上的优势.因此,我们提出了一种新的策略,在运行时判断临时表大小,根据 D 的阈值 Max$Bits$ 决定划分策略:D≥Max$Bits$ 时,数据量较大,为了减少聚集划分的代价,仍采用文献[13]提出的聚集划分策略;D<Max$Bits$ 时,聚集划分次数较少,因此令 D_i=1,即每次聚集划分处理 1 位 Hash Value,这样可以避免使用交换表.Max$Bits$ 的取值在第 4.2.2 节有详细说明.由第 3.1 节的分析可推断,如果过早启动聚集线程会造成聚集过大,导致比较严重的 Cache 访问冲突,抵消多线程执行的优势;但过晚启动聚集线程会造成串行执行占整个聚集划分的比例过高,因此需要在两者之间取得平衡.分析如下:

1. D<Max$Bits$.令 $H_j=\prod_{i=1}^{j} H_i$ 为第 j 次聚集划分后生成的聚集数,根据每次聚集划分后生成的聚集数分为 3 种情况:

(A) H_j≤N 且 $TTSize$≤C,表明数据量很少,第 1 次聚集划分完成后,即可完成聚集划分进入连接阶段;

(B) H_j≤N 且 $TTSize$>C 时,在每次聚集划分完成后,判断 $\frac{i \times TTSize}{H_j} \le C$ 是否满足,其中,2≤i≤H.如果存在 i 满足该条件,启动 Max(i)个聚集划分线程(Max(i)表示满足条件的 i 中最大的 i,下同),此时不会出现较严重的 Cache 访问冲突;如果不存在 i 满足该条件,则进入情况(C);

(C) 此时,H_j>N,在每次聚集划分完成后,判断 $\frac{i \times TTSize}{H_j} \le C$ 是否满足,其中,2≤i≤N.如果存在满足该条件的 i 且 Max(i)<N,此时启动 Max(i)个线程可能会降低处理器利用率,是否可以推迟启动多线程,直到 $\frac{i \times TTSize}{H_j} \le C$ 成立,此时可启动 N 个聚集划分线程.哪种策略的整体性能更好,第 3.3 节的理论分析和实验 6 的分析都表明,在 Cache 访问冲突很小的情况下,越早启动多线程聚集划分,串行执行的数据量越小,整体性能就越好,因此无须推迟启动.

在上述情况(B)和情况(C)中,启动聚集线程的时机 $j=\log_2 \frac{\text{Max}(i)}{C} \times TTSize$,如果 j 接近 P,则失去了启动多线程的意义.比如,由 $TTSize = \frac{2^D \times C}{2N}, H_{P-1} = 2^{\frac{D}{P}-1}$ 可知, $\frac{N \times TTSize}{H_{P-1}} > C$ 不成立,表明虽然可在第 P−1 次聚集划分后启动 N 个聚集划分线程,但此时 j=P−1,显然启动多线程不会提高性能.所以需要在聚集划分开始前,先由 $j=\log_2 \frac{i}{C} \times TTSize$ (其中,i 取值为 2≤i≤N)计算出(i,j)值,再以(i,j)为参数,根据第 3.3 节的加速比公式 S(i,P,j)计算出加速比 s,选择最大 s 对应的(i,j).如果此 j 值大于 Max$J$ 则无须 $\frac{i \times TTSize}{H_j} \le C$ 满足,在 H_j≥N 时即可启动 N 个聚集划分线程;否则,可按照情况(B)、情况(C)中的策略启动聚集划分线程.Max$J$ 的取值,实验 3 有详细分析.

2. D≥Max$Bits$.文献[13]中,每次聚集划分的 D_i 值较大,因此 P 值较小,所以需要在减少 Cache 访问冲突的前

提下尽可能早地启动多线程聚集.由于聚集划分过程需要访问交换表,在第 P–1 次聚集划分完成后,每个线程平均需要处理的数据量为 $\frac{2N\times TTSize}{H_{P-1}}=C\times 2^{\frac{D}{P}}$,$N$ 个聚集划分线程访问的数据量 $N\times C\times 2^{\frac{D}{P}}>>C$;而如果在第 P–1 次聚集划分完成后只启动两个线程,每个线程平均需要处理的数据为 $\frac{4\times TTSize}{H_{P-1}}=C\times\left(2^{\frac{D}{P}+1}/N\right)$,在数据量较大、处理器核心较少时,$C\times\left(2^{\frac{D}{P}+1}/N\right)$ 一般也是大于 C 的.所以,需要在聚集划分前判断 $C\times\left(2^{\frac{D}{P}+1}/N\right)$ 是否大于 C.如果大于 C,表明数据量较大,即使在 P–1 次聚集划分后,在只启动 2 个聚集划分线程的情况下也会发生比较严重的 Cache 访问冲突,故可在第 1 次聚集划分完成后启动 Min(H_1,N)个聚集划分线程;否则,在每次聚集划分完成后判断 $\frac{2i\times TTSize}{H_j}\le C$ 是否成立(2≤i≤N),如果成立,则可启动 Max(i)个聚集划分线程.而在 j=⌊P/2⌋时,如果仍不存在满足条件的 i,则可立即启动 N 个聚集划分线程.

3.2.4 聚集连接线程优化

聚集连接线程访问聚集划分后的临时表和结果输出缓存.聚集个数一般远远大于处理器核心数,而且第 3.1 节的分析表明,线程数大于 N 后,性能也不会有较大提升,所以最大连接线程个数 $JTNum=N$ 即可.由于连接过程存在对 Hash 表的重复访问,出现 Cache 访问冲突的可能性很大.聚集划分后,虽然一般情况下聚集之间数据分布比较均匀,但在数据严重不均匀分布的情况下,仍会造成聚集间数据分布不均匀,从而造成如下问题:(1) 不能较精确地控制连接线程的启动个数,在某些情况下可能造成比较严重的 Cache 访问冲突.比如,假设表 L 和 R 对应的聚集集合分别为 $ClusterSetL=(LCL_1,LCL_2,\ldots,LCL_A)$和 $ClusterSetR=(RCL_1,RCL_2,\ldots,RCL_B)$,如果只是简单地在聚集连接线程间平均分配聚集,则无法精确地控制连接线程启动数量,只能启动 N 个连接线程,但很可能这 N 个连接线程同时处理的数据量 $\sum_{i=1}^{N}Sizeof(LCL_i+RCL_i)>C$,导致比较严重的 Cache 访问冲突;(2) 如果线程间工作集大小相差较大,则会导致处理器核心间的负载不均衡,影响聚集连接整体性能.所以,为了减少连接线程之间的 Cache 访问冲突和处理器核心间的负载不均衡,本文提出了基于聚集大小分类的多线程聚集连接执行策略.聚集分类算法如下:

算法 **1**. Cluster Classification Algorithm.

Input: *ClusterSetL*[0…*A*–1], *ClusterSetR*[0…*B*–1]; **Output**: *WorkSet*[0…*JTNum*].

Procedure: ***WSallocation***(*ClusterSetL*,*ClusterSetR*)

```
1.  i=0; j=0; k=0;
2.  While (i<A and j<B)
3.  {
4.    While (ClusterSetL[i].radixvalue!=ClusterSetR[j].radixvalue)
5.      j++;
6.    TotalClusteSize=Add(ClusterSetL[i].size,ClusterSetL[j].size);
7.    for (k=1; k<=JTNum–1; k++)
8.      if (C/(k+1)<TotalClusteSize<=C/k)
9.      {
10.       add (i,j) to WorkSet[k];
11.       break;
12.     }
13.   if (0<TotalClusteSize<=C/JTNum)
```

14. {add (*i*,*j*) to *WorkSet*[*JTNum*];}
15. **if** (*TotalClusteSize*>*C*)
16. {add (*i*,*j*) to *WorkSet*[0];}
17. *i*++;
18. }

除了 *WorkSet*[0],在处理 *WorkSet*[*i*]中的聚集时,根据启动连接线程前 $2i \le N$ 是否满足分为两种情况:

(1) 如果 $2i \le N$ 满足,将连接线程分为 *i* 组,每组由 $1+\left\lfloor \frac{N-i}{i} \right\rfloor$ 个连接线程组成,执行一个聚集对的连接运算,比如 LCL_i 和 RCL_j.每组线程之间平均分配该聚集对中 Probe 表的元组,同时指定某个线程在 Probe 操作执行前构建 Hash 表,将 Hash 桶加入 Hash 表中,组内其他线程需等待 Hash 表构建完成;

(2) 如果 $2i > N$,启动 *i* 个连接线程,每个连接线程均匀分配聚集对即可.

聚集连接执行时,首先处理 *WorkSet*[1]中的聚集对,接着处理 *WorkSet*[2],以此类推,最后处理 *WorkSet*[0]中的聚集对.虽然它包括大于 *C* 的聚集对,会产生比较严重的 Cache 访问冲突,但从图 4 的性能分析可知,多线程连接的性能在任何情况下都要优于单线程连接,故启动 *N* 个连接线程.连接线程访问一定数量的数据后,才会访问一次输出缓存.一般情况下,输出缓存对连接线程的 Cache 访问性能影响较小,所以上述分析中未考虑输出缓存.

3.2.5 内存访问优化

本节优化了聚集连接阶段 Hash 连接的内存访问.具体如下:在聚集划分阶段,生成临时表的同时构建数据量较大的表的 Hash 表数组 *H*[*i*],同时记录映射到每个 *H*[*i*]的元组个数 *H*[*i*].*BucketNum*,避免 Hash 桶插入 Hash 表时浪费内存空间.聚集连接阶段生成 Hash 桶时,如果有数据映射到 *H*[*i*],则一次性地从已分配好的内存中获取能容纳 *H*[*i*].*BucketNum* 个 Hash 桶的内存块.该存储结构减少了 Hash 表所需内存量,从而降低了 Cache 访问冲突的可能性,因为 Hash 桶以数组存储,无须指针即可遍历 *H*[*i*]所有 Hash 桶.假设 *L* 表有 *n* 个 Hash 桶,节省的内存空间最少为 4*n* 个字节.特别地,当 Hash 列为数字型时,可节省一半的 Hash 桶存储空间.虽然构建 Hash 表数组时页面读取线程会增加一定的互斥访问开销,但实验 5 证实,Hash 连接的整体性能得到较大提高.

本文中启动的线程多以处理器核心数为上界,适用于目前处理器核心数较少的情况.当处理器核心较多,比如 20 甚至 80 个处理器核心时,则无须启动 *N* 个线程.因此,本文将每种操作启动的线程上界定为 8,当更多核心的处理器出现后,可再通过实验确定更为合适的线程上界.目前,共享 L2-Cache 由数据与指令共用,由于 Hash 连接为典型的大数据量运算,指令数据对本文的分析影响很小,所以本文未将其纳入分析范围.

3.3 多线程聚集划分性能分析

根据 Amdahl 定律,在多核处理器中,加速比为 $S(A,f)=\frac{A}{1+(A-1)f}$,其中,*A* 为执行运算的物理线程个数,*f* 表示串行化执行的数据占整个工作量的比重.在共享 L2-Cache 的多核处理器中,虽然由于 Cache 访问冲突而不能片面追求高加速比,但其仍是多线程程序性能的重要衡量指标,也可用于分析聚集划分阶段的优化策略.以下为聚集划分阶段加速比的命题和证明.

命题 **1**. 聚集划分阶段,加速比 $s=S(A_i,P,j)=\frac{PA_i}{P+j(A_i-1)}$,并且 *f* 与聚集划分线程的启动时机成正比.其中:A_i 为聚集划分线程数;*j* 为在 *P* 次聚集划分中,聚集划分线程的启动时机.

证明:假设数据在聚集中均匀分布.当 *D*<Max*Bits* 时,假设一次聚集划分串行执行的时间为 *t*,那么串行执行完所有的聚集划分所需时间为 *P*×*t*.如果在第 *j* 次聚集划分后启动 A_i 个聚集划分线程,则已经串行执行的时间为 *j*×*t*,产生 H_j 个聚集.一个线程处理一个聚集的时间为 $\frac{t}{H_j}(P-j)$,可得剩余的聚集划分处理时间为 $\frac{t}{H_j}(P-j)\times\frac{H_j}{A_i}\to\frac{t}{A_i}(P-j)$.加速后整个聚集划分时间为 $\frac{t}{A_i}(P-j)+j\times t$,而全部串行执行所需时间为 *P*×*t*,由加速比最基

本的定义可知 $s=S(A_i,P,j)=Pt\Big/\left(\frac{t}{A_i}(P-j)+jt\right)\to PA_i/P+j(A_i-1)$.而由 $S(A_i,P,j)=S(A_i,f)$,可得 $\frac{PA_i}{P+j(A_i-1)}=\frac{A_i}{1+(A_i-1)f}\to f=j/P$,与 j 存在正比关系.$D\geq$Max*Bits* 时证明过程类似,且同样成立.证毕. □

现以实例说明加速比计算,并分析第 3.2.3 节聚集划分线程的优化策略.在 4M 缓存的四核处理器中,假设有表 *L.tuple*=670000,*CellSize*=12B,可得 $D=\log_2(2N\times L.tuple\times CellSize/C)=4$,$D<$Max*Bits*.$j=2$ 时,满足 $\frac{i\times TTSize}{2^2}\leq C$ 的最大 i 为 2,此时启动 2 个聚集线程,s_1=1.33;若推迟启动多线程到 j=3,$\frac{4\times TTSize}{2^3}<C$ 成立,可启动 4 个聚集划分线程,s_2=1.23.由 $s_1>s_2$ 可知,在 Cache 访问冲突很小的前提下,越早启动多线程加速比越大;而且 f 越小,串行执行时间也就越少.实验 6 的结果也证实了上述推断.加速比计算公式未将 Cache 访问冲突的影响考虑在内,所以只有在 Cache 访问冲突很小的情况下,才能比较客观地反映程序性能的提高程度.

4 实验结果与分析

4.1 实验设置

本文的研究和实验基于共享 L2-Cache 多核处理器平台展开.实验数据库平台有两个:INGRES 和 EaseDB[7].实验硬件平台为 Intel 双核和四核处理器,3 种 L2-Cache,1M,3M 和 4M,2G 内存.在以下的实验中,大部分测试只使用了多核处理器的入门级配置,即 1M 缓存双核处理器,以突出本算法的优势.测试时,执行两个表的等值连接.除实验 1 以外,数据设置与文献[14]类似,两个表中的列都为整型,数据由随机函数生成.

4.2 实验内容及结果分析

4.2.1 预取和页面读取线程性能分析

本节基于 INGRES 实现了 Hash 连接多线程执行框架,验证预取和页面读取线程的有效性.硬件条件为 1M 缓存双核处理器.

实验 1. 测试页面读取线程读取页面生成临时表的性能.分别执行 3 种数据读取模式:(A) 无预取线程,单页面读取线程;(B) 无预取线程,多页面读取线程;(C) 启动预取线程,多页面读取线程.如图 6 所示,对不同大小的表按照上述 3 种模式执行数据读取,每次执行前数据库缓存中与访问表相关的页面的数量相同.由性能比较可知,预取和页面读取线程可以较大辐度提高将磁盘或内存中的页面数据转换成临时表的性能,不支持组读取的数据库可以增加组读取功能,能够提高顺序访问表页面的性能.

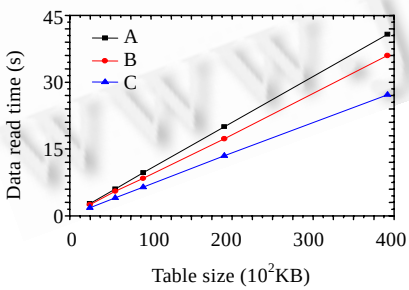

Fig.6 Page access performance (I/O and CPU time)

图 6 页面访问性能(I/O 和 CPU 时间)

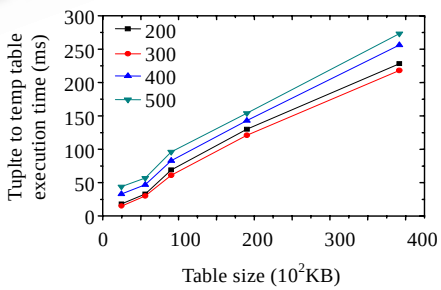

Fig.7 Page access performance (CPU time)

图 7 页面访问性能(CPU 时间)

接下来分析 Max*Page* 对页面处理线程的影响,测试了页面读取时的非 I/O 执行时间,数据与图 7 相同.由 *M.size*=2K,*M.tuple*=120,*CellSize*=12B 可得 Max*Page*=300,如图 7 所示.Max*Page*=200 时,因启动页面读取线程次数过多导致性能下降,但比较小;而 Max*Page*>300 后,由于数据页面和需要访问的临时表大于 *C*,Cache 访问冲突

加剧,性能有所下降.

4.2.2 聚集划分和聚集连接线程性能分析

由于 EaseDB 内核更加简洁,而且可以排除磁盘 I/O 的影响,对聚集划分和聚集连接的性能测试更准确.所以,本节基于 EaseDB,实现了除预取线程外的多线程框架,用于聚集划分和聚集连接线程的性能分析.

实验 2. 测试 Max*Bits* 参数取值.处理器为 1M 缓存双核处理器,对于大小不同的表,分别采用聚集划分策略 D_i=D/P 和 D_i=1,测试聚集划分执行时间,测试表的 *L.tuple* 分别为(3.5e+10^5,7.0e+10^5,1.4e+10^6,2.8e+10^6,5.57e+10^6,1.1185e+10^7),*CellSize*=12B,对应 D 值为(4,5,6,7,8,9).如图 8(a)所示,由 D_i=D/P 和 D_i=1 时多线程聚集划分性能比较可知,在 D=6 之前,虽然 D_i=D/P 在聚集划分次数上占有优势,但每次执行聚集划分处理时需要访问交换表,加剧了 L2-Cache 访问冲突,因此 D_i=1 的性能优于前者.在 D=4 时,D_i=1 只需要 D_i=D/P 执行时间的 57%,D=5 时只需 74%.但随着数据量的增加,D_i=1 时需要执行的聚集划分次数不断增加,而 D_i=D/P 的聚集划分次数则不会明显增加,一般维持在 2 或 3,所以 D_i=1 时的聚集划分执行时间不但受到 *L.tuple* 的影响,还会由于 P 值的增加导致整个聚集划分需要处理的数据量比 D_i=D/P 增加得快.比如,假设需要处理的表的数据为 *TTSize*,在 D=4 时,D_i=1 时聚集划分访问的数据量 *TotalSize*=4×*TTSize*,而 D_i=D/P 需 2 次聚集划分,由于还需要访问交换表,其 *TotalSize*=4×*TTSize*,但其每次聚集划分的 Cache 访问冲突比 D_i=1 时严重,所以 D_i=1 时性能更优.同理,D=5 时也一样.而 D=6 时,D_i=D/P 时 *TotalSize* 同样等于 4×*TTSize*,而 D_i=1 时 *TotalSize*=6×*TTSize*.显然,Cache 访问性能优势已被抵消,所以在 D=6 时,两者性能已相差不大.当 D=7 以后,D_i=D/P 的性能已明显优于后者,因此可得 Max*Bits*=7.在大多数情况下,此设置可以保证整体性能是最好的.

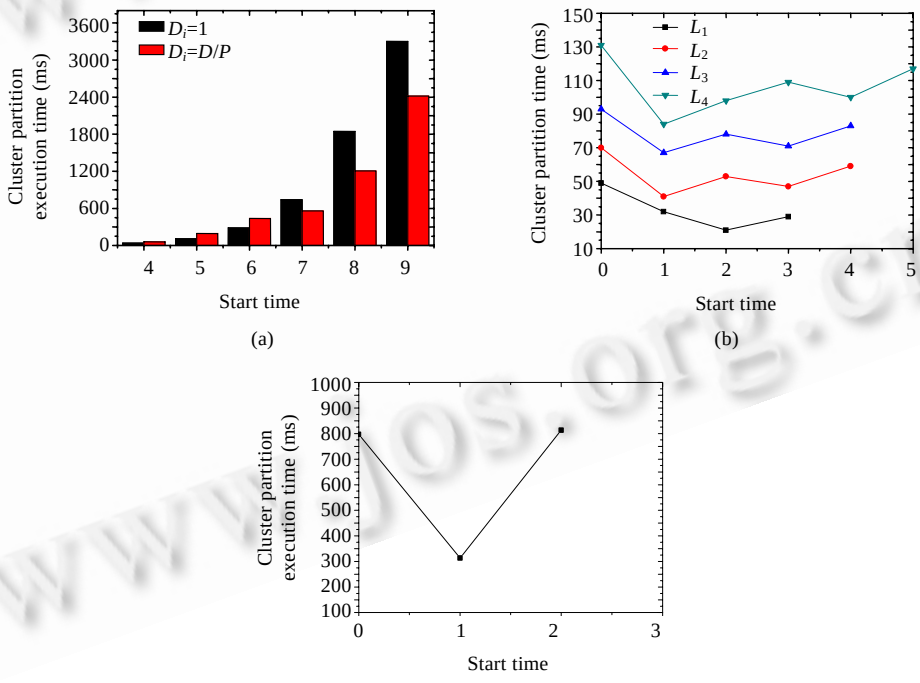

Fig.8 Cluster partition performance comparison

图 8 聚集划分性能比较

实验 3. 测试聚集划分线程的性能.分为两种情况:D<Max*Bits* 时,多线程聚集划分 L 表,测试 4M 缓存四核处理器条件下,在不同时机启动聚集划分线程的性能.L_i.*tuple*(i=1,2,3,4)分别为(6.67e+10^5,1.0e+10^6,1.33e+10^6,

1.67e+10^6),*D*=(4,5,5,6),*CellSize*=12B,满足 $\frac{i\times TTSize}{H_j}\le C$ 的 j 和 Max(i)分别为((2,2),(3,2),(3,2),(4,3)).如图 8(b)所示,*Start Time* 表示聚集划分线程在哪次聚集划分后启动,其中,*Start Time*=0 表示在聚集划分开始时就启动聚集划分线程.由于此时只有一个聚集,即临时表,所以只能启动一个聚集线程.L_1 在第 2 次聚集划分完成后启动 2 个聚集划分线程,基本不会出现 Cache 访问冲突,所以性能有明显提高.其他表由于数据量较大,即使在满足条件后,由于串行执行计算过多,虽然不会出现 Cache 访问冲突,但性能也不是所有 *Start Time* 中最优的.因此可以推断,当 $\log_2\frac{i}{C}\times TTSize\le\frac{1}{2}D$ 时,可按照第 3.2.3 节中情况 1(B)和情况 1(C)的策略启动聚集线程,否则,在 $H_j\ge N$ 时即可启动 N 个聚集划分线程;$D\ge$Max*Bits* 时,多线程聚集划分 R 表,1M 缓存双核处理器,*R.tuple*=7.744e+10^6,*CellSize*=12B,*D*=9,*P*=3.如图 8(c)所示,因为 $C\times\left(2^{\frac{D}{P}+1}/N\right)>C$,所以在第 1 次聚集划分后启动多线程性能最好.

实验 4. 测试聚集连接线程的性能.处理器为 1M 缓存双核处理器,分别执行 3 种模式:(A) 按照聚集大小分类后分配线程工作集;(B) 平均分配线程工作集;(C) 单线程.连接表 *L*,*R* 的元组数分别为 *a*(1.0e+10^5,1.76e+10^5),*b*(2.5e+10^5,3.666e+10^5),*c*(5.2667e+10^5,7.0e+10^5),*d*(1.167e+10^6,1.4e+10^6),*e*(1.867e+10^6,2.8e+10^6),*f*(4.0e+10^6,5.6e+10^6),*g*(7.433e+10^6,1.12e+10^7),*CellSize*=12B,如图 9 所示,随着数据量增加,模式(A)的优势不断增加,*Start Time*=*e* 时,聚集中数据分布比较均匀导致性能差距不大,而单线程模式在双核处理器中不能充分利用处理器资源,导致其性能与其他两种模式性能相差较大.

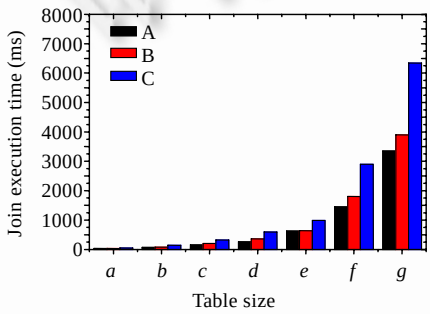

Fig.9 Cluster join performance comparison

图 9 聚集连接性能比较

实验 5. 比较 Simple Hash Join(SHJ),Radix-Join(RJ)[13],多线程 Radix-Join(MRJ),Cache-Oblivious Hash Join(COHJ)[7,14],本文提出了基于 CMP 优化的多线程 Hash 连接(CMPHJ)但未经内存访问优化和经内存访问优化的 CMPHJ(MOCMPHJ)的性能.图 10(a)为 MOCMPHJ 与 MRJ 的性能比较,采用的处理器为 3M 和 4M L2-Cache 四核处理器.如果 Radix-Join 算法只是进行多线程优化,而未考虑共享 L2-Cache CMP 的特点,线程间严重的 Cache 访问冲突会降低程序的性能,而且数据量越大,Cache 访问冲突影响越严重.L2-Cache 容量较大的处理器,连接执行时的 Cache 命中率较高,因此性能有所提高.图 10(b)为 MOCMPHJ 与基于 Cache 优化 Hash 连接算法和 CMPHJ 的性能比较,采用的处理器和实验数据同实验 4.在多核处理器中,如果单线程执行,甚至 MRJ 的性能也要优于 Cache 优化后的 Hash 连接.所以,在共享 Cache 的多核处理器中,需要综合考虑多线程和 Cache 优化才能充分发挥多核处理器的性能,而且本文提出的内存优化方法由于减少了内存访问量和 Cache 访问冲突,相对于 CMPHJ,性能有所提高.

实验 6. 双核和四核处理器性能测试.双核处理器的 L2-Cache 为 1M 和 4M,四核处理器为 4M.首先测试了这两种处理器条件下,经内存访问优化的多线程 Hash 连接的性能,实验数据与实验 4 相同.如图 11(a)所示,同为双核处理器,L2-Cache 较大的一方不但可以更早地启动聚集划分线程,Cache 访问冲突相对较少,而且由于 L2-Cache 增大,每个聚集的大小增加了,导致聚集划分次数也减少了,所以性能上有所提升.四核处理器相对于相同 L2-Cache 的双核处理器,可并行执行的物理线程数增加了 1~2 个,而且我们提出的优化策略为尽可能早地启动多线程聚集划分,所以虽然 N 值增加了,但由于 L2-Cache 较大,多线程聚集划分的启动时机基本没有受到影响,比如 *L.tuple*=1200000 时,在双核处理器时,聚集线程启动时机 j=2,启动 2 个线程;而四核处理器时,同样也是在 j=2 时启动线程,但可以启动 3 个线程,而且聚集连接性能也随着处理器核心数的增加而提高性能,因此总体性能要大大优于双核处理器.