阵列众核结构上的一种多层分区 Hash 连接算法

石 嵩 宁永波 李宏亮 郑 方
(江南计算技术研究所 无锡 214083)

摘 要 连接是数据查询处理中最耗时、使用最频繁的操作之一，对提高连接操作的速率具有重要意义。阵列众核处理器是一类重要的众核处理器，具有强大的并行能力，可用来加速并行计算。基于阵列众核处理器的结构，设计和优化了一种高效的多层分区 Hash 连接算法。该算法通过多层划分的策略大大降低了主存访问次数，通过分区重排方法有效消除了数据倾斜的影响，获得了很高的性能。在异构融合阵列众核处理器 DFMC(Deeply-Fused Many Core)原型系统上的实验结果表明，DFMC 上多层分区 Hash 连接算法的性能是 CPU-GPU 耦合结构上最快的连接算法的 8.0 倍，表明利用阵列众核处理器加速数据查询应用具有优势。

关键词 阵列众核，Hash 连接，数据倾斜，并行算法

中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2016.3.003

Multi-layer Partition Hash Join Algorithm on Array-based Manycore Architecture

SHI Song NING Yong-bo LI Hong-liang ZHENG Fang
(Jiangnan Institute of Computing Technology, Wuxi 214083, China)

Abstract Join is one of the most time-consuming and most frequently used operations in database query processing, so it has great importance to improve the speed of join operation. The array-based manycore processor is one of the most important classes of manycore processors and has great parallelism, and can be used to accelerate parallel computing. In this paper, we designed and optimized an efficient algorithm called multi-layer partition hash join algorithm on array-based manycore architecture. Our algorithm achieves high performance by multi-layer partitioning which reduces the times of main memory accessing, and by partition rearrangement which eliminates the influence of data skew. Experimental results show that the performance of multi-layer partition hash join algorithm on DFMC(Deeply-Fused Many Core) processors is 8.0x times higher than that of the fastest hash join on CPU-GPU coupled-architecture, which demonstrates that array-based manycore processors have advantages in data query processing.

Keywords Array-based manycore, Hash join, Data skew, Parallel algorithm

1 引言

阵列众核处理器是指一类计算核心以阵列方式组织的众核处理器，包括 Tilera[1]、PACS-G[2]、epiphany[3]、MPPA[4]、Godson-T[5] 和 DFMC[6] 等。与 NVIDIA 的 GPU、Intel 的 MIC 这些非阵列结构的众核处理器相比，阵列众核处理器具有可扩展性好、功耗低、实现代价小的优点，是众核处理器发展的重要方向。

众核处理器具有高峰值性能、高存储带宽和强大的并行计算能力，可以用来加速大规模的并行任务，已经在高性能计算领域中得到成功应用。近年来，许多研究表明众核处理器如 GPU、MIC 等可以大幅度提高数据库查询处理的速度[7-9]，但是阵列众核处理器上的研究尚较少，本文对阵列众核处理器上 Hash 连接算法的设计和优化进行了研究。

连接是数据查询操作中最耗时、使用最频繁的操作之一。绝大部分的复杂查询都要使用连接操作，而且在当前流行的列式数据库系统中，关系表的各个属性分开存储，更需频繁的连接操作来连接多个属性，因此提升连接操作的效率具有重要意义。在传统关系数据库中，主要有 3 种连接算法：嵌套循环连接、Sort-Merge 连接和 Hash 连接，其中 Hash 连接是最为高效的一种连接算法，本文选取 Hash 连接进行研究，设计了一种适合阵列众核结构的多层分区 Hash 连接算法。

尽管阵列众核具有强大的计算潜力，但若要充分发挥其性能，需要用户利用好处理器提供的底层硬件特征。特别是对于连接这种数据密集型的任务，其具有计算简单而访存频繁的特点，而阵列众核处理器由于核心数众多，单个核心所分得的带宽资源有限，容易出现访存瓶颈，因此需要设计出访存友好的连接算法，以尽量减少访存次数。本文根据阵列众核的结构特点，通过多层划分的办法，将关系表划分成多个其 Hash 表可以放置在局部存储器的子表，然后对子表进行 Hash 连接，从而有效降低了主存访问次数。最后，为了避免数据倾斜带来的负载均衡问题，本文设计了一种有效消除数

到稿日期：2015-01-05 返修日期：2015-05-13

石 嵩(1990－)，男，硕士生，主要研究领域为高性能微处理器设计，E-mail：shisong1990@163.com；宁永波(1986－)，男，硕士，助理工程师，主要研究领域为高性能微处理器设计；李宏亮(1975－)，男，博士，副研究员，主要研究领域为高性能微处理器设计；郑 方(1985－)，男，硕士，助理研究员，主要研究领域为高性能微处理器设计。

据倾斜干扰的方法，使得算法在有数据倾斜的关系表的连接中仍能高效工作。

本文第2节介绍众核处理器上连接算法的相关研究工作；第3节介绍阵列众核结构的模型；第4节详细介绍多层分区Hash连接算法；第5节对算法性能进行理论分析；第6节是实验结果；最后总结全文。

2 相关工作

连接算法由于其重要性得到了广泛的研究，本文只介绍众核处理器上连接算法的研究情况。

在GPU方面，He等人[10]最早研究了GPU上连接算法的设计和实现，他们利用GPU提供的底层硬件支持，实现了Map、Prefix scan、Scatter、Gather、Split和Sort等原语，并使用这些原语设计了GPU上的NLJ、SMJ和HJ算法，文章声称与当时CPU上的连接算法相比，这些算法最大可以达到7倍加速。但是他们的算法假定待连接的数据已经放置在GPU上的显存中，因此没有考虑CPU与GPU之间的PCI-E的传输时间。随后，Kaldewey等人[11]研究了UVA(Unified Virtual Addressing)存储构架下GPU上的连接算法，他们的算法假定数据放置在主存中，利用UVA技术，PCI-E带宽的利用率能达到6.1GB/s。此外，He等人[12]研究了CPU-GPU耦合结构(如AMD的APU)上的Hash连接算法，CPU-GPU耦合结构结合了CPU和GPU的优点，同时消除了PCI-E总线的影响，因此获得了更好的性能。

在Intel系列的众核处理器上，Petrides等人[13]研究了Intel SCC架构上决策支持系统查询的性能，其中使用了数据预取的方法来优化连接操作。文章提到利用Intel SCC可以开发出巨大的并行性。在阵列众核处理器上，由于结构新颖，还尚未有关于连接算法的研究，本文针对阵列众核结构，对连接算法进行设计和优化研究。

3 阵列众核处理器结构

阵列众核结构最主要的特征是多个计算核心以阵列方式(Mesh)组织，可以分为同构阵列众核和异构阵列众核两类，其中同构阵列众核全部由计算核心阵列构成，异构阵列众核在阵列核心外还有额外的管理核心，后者可视为前者的一般化情形。本文抽象的阵列众核结构采用后者，但是所设计的算法对于两种结构均适用。

阵列众核处理器的整体结构如图1所示，包括管理核心MPE(Management Processing Element)、计算核心阵列CPA(Computing Processing Array)、存控和系统接口，它们通过片上网络NoC互连。

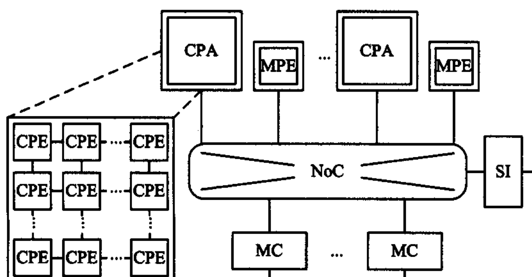


图1 阵列众核结构

其中MPE是功能完整的通用核心，负责提供一些标准服务如IO代理和消息代理等，也适合运行程序的串行部分。CPA由若干计算核心CPE(Computing Processing Elements)以阵列方式组织构成，阵列内的CPE可通过高速的显式或者隐式的片上通信(如消息)交换数据。CPE是功能简单的运算核心，支持SIMD指令开发细粒度的并行性，提高峰值性能，从而加速大规模的并行任务。

阵列众核处理器中CPA的数据存储构架如图2所示。CPE内具有片上存储器SPM(Scratched Pad Memory)，或CPA中共享片上存储器(本文叙述采用前者)。SPM可配置为数据Cache结构，也可配置为由软件显式管理的局部存储器，配置为Cache可以减轻软件编程的负担，但由于需要保持众多核心之间Cache的一致性，其效率比配置为局部存储器的低。本文将SPM视为局部存储器，SPM与主存之间通过软件显式管理的DMA传输数据。

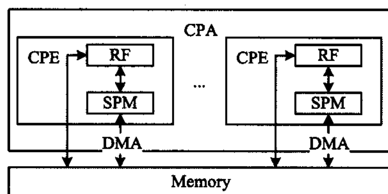


图2 CPA存储构架

从阵列众核的结构可以看出，阵列众核提供了多个层次的并行性，若要充分发挥性能，则需最大限度利用不同层次的并行性，且要考虑好多个核心并行时的负载均衡问题。另一方面，主存的访问延迟比SPM高一到两个数量级，充分利用好SPM是提高访存效率的关键。

4 多层分区Hash连接算法

本文用R和S表示两个关系表，每个表中包含两个属性(key,idx)，key表示待连接的关键字，idx表示元组的标识或行号，key和idx均是32位的整型数据。R的元组个数不大于S中的元组个数，即$|R|\leqslant|S|$。连接是指将两个表R和S(或多个表)中符合特定条件的元组连接起来，形成一个新表。本文考虑最常用的等值连接情形，即连接条件为r.key=s.key。Hash连接通常包含3个步骤：划分、建立Hash表和探测Hash表，其中划分过程是可选的，根据划分与否可以分为简单Hash连接算法和分区Hash连接算法。分区Hash连接算法可以增加数据局部性，从而较好地适应具有Cache或者局部存储器结构的处理器，因此阵列众核处理器中更适合采用分区Hash连接。本节将详细介绍如何在阵列众核结构上实现这3个步骤。

4.1 划分

传统的分区Hash连接算法[14]直接将一个大表分成若干小表(小表的Hash表可以容纳在SPM中)，通常分区数目非常多，如具有16M($1M=2^{20}$)个元组的表，SPM中可支持的缓冲大小为1024个元组，则需要16M/1024=16384个分区。若使用一次划分将整个表分成16384个分区，将导致非常严重的访存问题。因为对于分区数较少的情形，可以在SPM中开辟缓冲区来存储划分到各个分区的数据，当缓冲区满的时候使用DMA将缓冲区的内容写回到主存。当分区数增多时，SPM缓冲区的数目会增加，而每个SPM缓冲区的容量将

减小，当SPM缓冲区容量小于某个临界值时，可能会导致DMA的效率急剧下降（对于上例中16384个分区的情形，SPM中无法开辟出这么多的分区数，只能直接访问主存），从而急剧降低访存效率。

因此，本文在传统划分过程的基础上进行改进，采用分区的思想，但不是通过一次划分直接将大表划分成最终数目的子表，而是通过迭代划分的方法将关系表R和S划分成若干足够小的子表$R_1,R_2,\cdots,R_B$和$S_1,S_2,\cdots,S_B$（R_i生成的Hash表可以放置在SPM中）。算法使用key的最右边的b个比特进行划分，得到$B=2^b$个子表。整个分区过程主要分为核间划分（图4中步骤②、③以及步骤⑦、⑧）和核内划分（图4中步骤④和步骤⑨）两个部分。

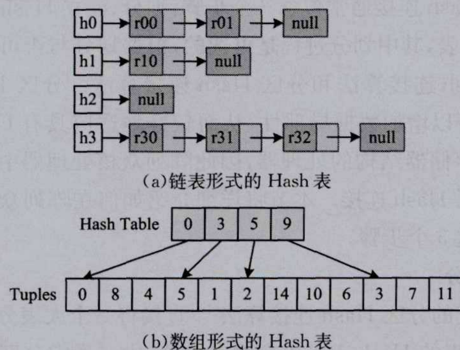
在核间划分过程中，各CPE（如CPE_i）首先利用key值最右端的$\log(P)$个比特将关系表R（以R为例）划分成P（核心数目）个子表$R_i(1),R_i(2),\cdots,R_i(P)$，并得到一个直方图$H_i$，$H_i[j]$表示$CPE_i$中第$j$个子表的元组数；接着在CPA内并行计算出前缀和$PrefixSUM_i$，其中，$PrefixSUM_i[j]=\sum_{y=1}^{j-1}\sum_{x=1}^{p}H_x[y]+\sum_{x=1}^{i-1}H_x[j]$表示$CPE_i$中第$j$个子表写回主存的起始位置，然后各CPE根据$PrefixSUM_i$将自己划分的各个子表$R_i(j)$散播到相应位置$PrefixSUM_i[j]$，得到$P$个全局子表，将子表$R(i)$划分给$CPE_i$。

核内划分过程中各CPE继续利用剩下的$b-\log(P)$个比特将自己所持的子表划分成B/P个更小的子表（如图4中步骤④所示）。由于B/P通常很大而SPM容量有限，该过程需要分多次迭代完成，如每次利用d比特划分，则需要$(b-\log(P))/d$次迭代。

整个划分过程结束后，各CPE_i均得到B/P个R的子表$R_i(1),R_i(2),\cdots,R_i(B/P)$和$B/P$个$S$的子表$S_i(1),S_i(2),\cdots,S_i(B/P)$，只需要连接对应的子表$R_i(j)$和$S_i(j)$即可。

4.2 构建Hash表

在对R进行分区之后，需要为各R的子表创建Hash表（图4中步骤⑤）。Hash表中存储关系表中的各个元组，以供高速查找，其实现方式主要有两种，一种是链表形式的Hash表，一种是数组形式的Hash表，如图3所示。



(a)链表形式的Hash表

(b)数组形式的Hash表

图3 两种Hash表

链表形式的Hash表的优点是创建过程简单，只需要扫描一遍关系表中的元组，然而链表形式的Hash桶所耗费的存储空间大而且要不断为新结点分配空间；同时，链表形式的Hash桶的遍历过程中，访存单元是离散的。与之相比，数组形式的Hash表克服了链表形式Hash表的缺点，即各Hash桶只存储了元组数据，而且Hash桶的遍历过程中，访存单元是连续的，可以加快访存操作。因此，本文采用数组形式的Hash表。Hash表的创建过程需要扫描两次关系表，第一次用于统计各Hash桶中的元组数目，然后计算各Hash桶的起始位置；第二次扫描将各元组散播到各Hash桶对应的位置。

该算法在实现中需要考虑的一个问题是Hash桶数目的选择，如果Hash桶数目太少，则各Hash桶中的平均元组数目会很多，在探测Hash表阶段的查找时间会增长，会降低整个Hash连接的效率；另一方面，如果Hash桶的数目很多，则需要耗费更多的SPM空间，且Hash表在SPM与主存之间的传输时间会变长。折中考虑，本文Hash桶的数目选择为R子表的平均大小即$|R|/B$。

4.3 探测Hash表

在为表R构建好Hash表以及对表S进行分区之后，最后一个步骤是依次遍历各S子表中的元组（图4中步骤⑩），对其进行Hash计算，然后到Hash表中查找是否有匹配值。由于已经进行了分区，各CPE只需对自己所持分区中的元组进行连接操作即可，CPE之间无需通信和同步操作。

4.4 算法整体流程

多层分区Hash连接算法通过多层划分的方法将数据表划分成多个子表，使其构建的Hash表可放置于SPM，然后依次处理各个子表。算法流程如图4所示，主要分为3个步骤：

步骤1 划分。将两个数据表均匀分配给P个CPE（步骤①和⑥），各CPE对两个待连接的数据表进行1次核间划分（步骤②、③与⑦、⑧）和$(b-\log(P))/d$次核内划分（步骤④和⑨），共得到B个分区，每个CPE持有B/P个分区。

步骤2 构建Hash表。各CPE依次处理各个分区，对于每一个分区，利用其中的子表R_i构建一个Hash表，并将它放置在SPM缓冲中。

步骤3 探测Hash表。对于该分区中的另一个子表S_i，依次计算S_i中各个元组的Hash值，根据Hash值在由R_i构建的Hash表中查找是否有匹配值，如果满足匹配条件，则将两个元组加入到连接结果中。

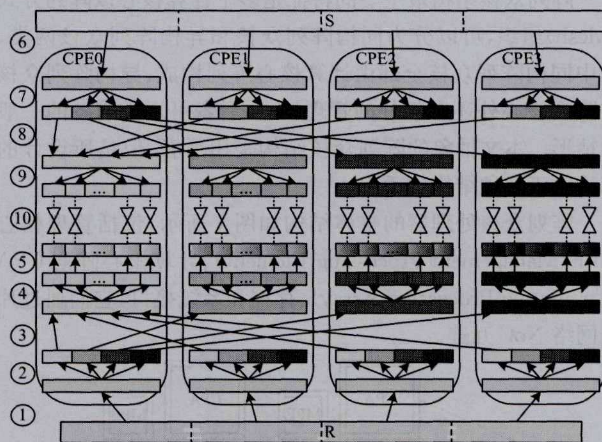


图4 CPA内多层分区Hash连接示意图

5 数据倾斜处理

数据倾斜主要是指非均匀的数据分布，如高斯分布、Zipf分布[15]等。数据倾斜在并行算法中会导致各计算单元的负载出现不均衡的现象。在连接操作中，通常是一个表S的某个属性和另一个表R的主键（称为S表的外键）连接。表的

主键是唯一的,无需考虑存在数据倾斜的现象。但是 S 表中待连接属性可能存在数据倾斜的问题,所以本文着重解决 S 表的数据倾斜问题。

在 4.1 节的核间划分过程中,各 CPE 通过一次划分将数据划分成 P(P 为 CPE 数目)个分区后就将这 P 个分区分给各 CPE(每个 CPE 负责一个分区)。该划分方法的优点是实现简单、核间通信少,但是当存在数据倾斜而导致各分区中的数据量出现不均衡时,各 CPE 会出现负载不均衡问题。解决该问题的一个方法是增大核间划分过程中的分区数,即将整体数据分成 $a*P(a>1)$ 个分区,然后为每个 CPE 分配 a 个分区。因为增加了分区数,所以在将分区分配给 CPE 的过程中算法可以决定分配给某个 CPE 的 a 个分区的组合方式,从而可选取一种使各 CPE 所分得数据量比较均衡的组合方式。

本文采用分区重排分配的方式来实现这个目标。即首先根据各分区的数据量对所有分区进行排序,然后将排序后的分区按照连续划分的方式进行分组,共分成 a 组,每组 P 个分区,最后将每组中的 P 个分区随机地分给 P 个 CPE,如图 5 所示。

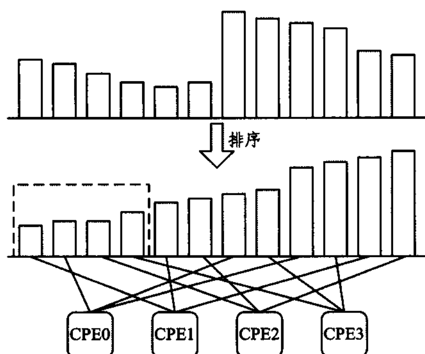


图 5　分区重排分配

由于经过了排序过程,使得相邻分区的数据量相差较小,即一个分组内的各分区的数据量的倾斜度相对较小。同时,一个 CPE 从某个分组内获得的分区在分组内的相对位置是随机的,如图 5 中 CPE0 从分组 1 中获取第 2 个分区,从分组 2 中获取第 3 个分区,从分组 3 中获取第 1 个分区,这样可以克服采用固定位置分配所带来的系统误差,即各 CPE 获得的数据量都接近于 3 个分组的平均值之和。

在实现中需要确定核间划分过程中的分区数目或者 a 的取值,如果 a 太小,则可供消除数据倾斜的分区组合方式不够多,使用分区重排分配方法的效果不明显(最极端的例子是 $a=1$,则无法消除分区中存在的数据倾斜问题)。反之,如果 a 太大,则分区的复杂度会上升。因此,本文在核间划分过程中选取 $a=8$,即 $8\times64=512$ 个分区。

6　实验结果与分析

实验在异构融合阵列众核处理器 DFMC 上进行,使用其中的 1 个 MPE 和 1 个 CPA 共 64 个 CPE,主频 1.0GHz,支持 128 位 SIMD 指令,每个 CPE 拥有 32kB 局存,局存访问延迟 3 周期,局存与主存间支持 DMA 多种模式传输,CPA 内支持片上通信,单次主存访问延迟约 100 周期,主存带宽为 25.6GB/s,32 位整数峰值性能为 256GOps。软件配置是 MPE 运行在内核为 2.6.28 版本的 Linux 系统上,CPE 的系统是一个轻量级的定制系统,编程环境是定制的提供并行编程的 C 语言。目前 DFMC 原型系统采用 FPGA 构建,时钟频率为 2.6MHz[6]。实验在该原型系统上进行,所得的实验结果根据真实的时钟频率进行校准。

6.1　分区数目对划分时间的影响

在第 4 节中提到分区数目过多时会由于访存等原因导致划分过程的效率很低,但是若分区数过少,需要的划分次数则会增多,因此要选取合适的分区数。本文在 DFMC 上研究了一次划分的时间随分区数目的变化关系。使用的表的大小为 16M,服从均匀分布,分区数目从 4 扩展到 128,实验结果如图 6 所示。

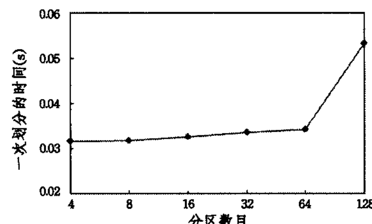


图 6　一次划分过程中执行时间随分区数目的变化关系

从图 6 可以看出,在分区数目从 4 增加到 64 的过程中,一次划分的时间增长极其缓慢,这是因为划分过程都是两次读写操作,与分区数目关系不大。然而当分区数目增长到 128 时,划分时间急剧上升,与分区数为 64 相比,增加了 55%。这是因为在 DFMC 处理器中,要最大限度发挥出存储带宽的利用率,需要 DMA 最低传输 256B 的数据。在实验中 SPM 配置的写缓冲区大小为 16kB(这是 SPM 可用以配置写缓冲区的最大容量),当分区数为 64 时,单个缓冲的容量为 256B,正好满足最低传输数据量要求。而当分区数为 128 时,单次 DMA 能传输的数据量只有 128B,带宽利用率急剧下降,从而导致划分时间也急剧增长。可见,分区数设置为 64 比较合适,在实际中可根据具体情况取 16～64。

6.2　算法执行时间

本文测试了算法的执行时间。其中表 R 和表 S 的键值服从均匀分布,表 R 的大小为 $|R|=16$M 个元组,表 S 的大小 $|S|$ 从 16M 增长到 128M。实验中核间划分过程中分区数设置为 64(CPE 数目),核内划分过程采用两层划分,两层划分的分区数目分别为 64 和 32。算法各个阶段的执行时间如图 7 所示。

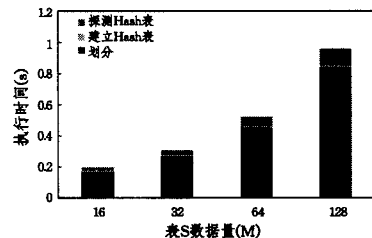


图 7　多层分区 Hash 连接算法各阶段的执行时间

从图 7 可以看出,算法的执行时间随表 S 的数据量呈线性增长(图中横坐标相当于对数坐标),有很高的效率,如表 S 数据量为 64M 时,总执行时间为 0.52s,连接速率达 $(64+16)/0.52=154$M (元组/s)。从各阶段的执行时间可以得知,算法的主要执行时间为划分时间,今后进一步的优化可以从划分过程着手。

6.3　数据倾斜的影响

本文使用 Zipf 分布的数据测试了分区重排方法消除数

据倾斜影响的效果。Zipf 分布的概率密度函数如下：

$$p(k)=\sum_{i=1}^{k} i^{-\theta}/\sum_{i=1}^{N} i^{-\theta} \quad (1)$$

表 R 和 S 的大小分别为 16M 和 64M，算法中核间划分过程的分区总数为 8×64＝512 个。多层分区 Hash 连接算法的执行时间随 Zipf 分布中 θ 值的变化关系如图 8 所示，θ＝0.7 时表示数据倾斜已经非常严重。从图中可以看出，Zipf 分布中的数据倾斜现象对连接算法的性能影响很小，可见分区重排方法能有效消除数据倾斜对算法性能的影响。

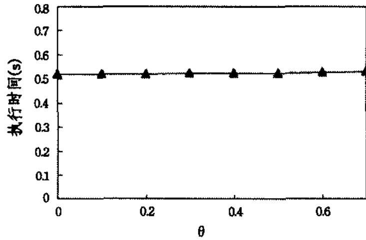


图 8 算法执行时间随 Zipf 分布中 θ 值的变化关系

6.4 与 GPU 众核上连接算法的比较

本文选取了最新的也是性能最高的 CPU-GPU 混合结构上的散列连接算法[12]与 DFMC 上的多层分区 Hash 连接算法进行了对比，文献[12]在 AMD A8-3870K 上实现了简单散列连接(SHJ)和分区散列连接两种算法(PHJ)。其中 AMD A8-3870K 和 DFMC 的参数的比较如表 1 所列。

表 1 DFMC 与 A8-3870K 参数比较

处理器	核心数	32 位整数峰值性能	存储带宽
DFMC	1MPE+64CPE	4GOps+128GOps	25.6GB/s
A8-3870K	4CPU+400GPU	48GOps+240GOps	29.9GB/s

文献[12]在实验中使用的关系表同样由一个 32 位的键值和 32 位的 ID 组成，$|R|$ 和 $|S|$ 均为 16M，实验中 SHJ 和 PHJ 的最小执行时间如图 9 所示(数据来源于文献[12])。从图中可知，尽管 DFMC 的峰值性能和存储带宽均小于 AMD A8-3870K 处理器，但是本文设计的多层分区 Hash 连接算法的性能比 GPU-GPU 上的 SHJ 和 PSJ 的性能分别高 8.5 和 8.0 倍。这主要是由于本文算法很好地结合了阵列众核的结构特征，极大地发挥了阵列众核的峰值性能和存储带宽。一方面，本文算法通过对 SPM 的精细管理利用，大大降低了访存延迟，提高了访存效率，通过负载均衡的处理，充分利用了各个 SPE 的计算资源；另一方面，文献[12]中的算法由于硬件结构的限制(如无法动态分配空间等)，采用了大量的原子操作，限制了其算法效率。

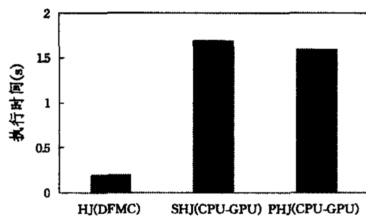


图 9 与 CPU-GPU 处理器上连接算法的比较

结束语 阵列众核处理器是众核处理器发展的重要方向，在高性能计算领域发挥着重要作用，将其用来加速数据库、数据处理等数据密集型应用具有广阔的前景和重要的现实意义。本文基于阵列众核的结构特点，设计和优化了一种高效的多层分区 Hash 连接算法。算法通过多层划分的策略，将关系表划分成可以在 SPM 中处理的子表，最大限度地降低了主存的访问次数，提高了算法效率。同时，设计了分区重排的方法来消除数据倾斜的影响，有效地维护了计算核心之间的负载均衡。同 GPU 上的连接算法相比，DFMC 上多层分区 Hash 连接算法的性能是它们的 8.0 倍，表明阵列众核处理器在加速数据查询处理中更有优势，将在数据查询应用中发挥重要作用。

参考文献

[1] Ramey C. Tile-gx100 manycore processor: Acceleration interfaces and architecture[R]. Tilera Corporation, 2011

[2] Sato M. Feasibility study on future HPC infrastructure[R]. 2014

[3] Gwennap L. Adapteva: More flops, less watts[J]. Microprocessor Report, 2011, 6(13): 11-12

[4] Dinechin B I T D, Massas P G D, Lager G, et al. A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor[J]. Procedia Computer Science, 2013, 18: 1654-1663

[5] Fan D, Yuan N, Zhang J, et al. Godson-T: An efficient many-core architecture for parallel program executions[J]. Journal of Computer Science and Technology, 2009, 24(6): 1061-1073

[6] Zheng Fang, Zhang Kun, Wu Gui-ming, et al. Architecture techniques of many-core processor for energy-efficient in high performance computing[J]. Chinese Journal of Computers, 2014, 37(10): 2176-2186(in Chinese)
郑方，张昆，邬贵明，等. 面向高性能计算的众核处理器结构级高能效技术[J]. 计算机学报，2014，37(10)：2176-2186

[7] He B, Lu M, Yang K, et al. Relational query coprocessing on graphics processors[J]. ACM Transactions on Database Systems (TODS), 2009, 34(4): 21

[8] Mostak T. An Overview of MapD (Massively Parallel Database) [R]. Massachusetts Institute of Technology, 2013

[9] Scherger M. Design of an in-memory database engine using Intel Xeon Phi coprocessors[C]//PDPTA'14. 2014

[10] He B, Yang K, Fang R, et al. Relational joins on graphics processors[C]//Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. ACM, 2008: 511-524

[11] Kaldewey T, Lohman G, Mueller R, et al. GPU join processing revisited[C]// Proceedings of the Eighth International Workshop on Data Management on New Hardware. ACM, 2012: 55-62

[12] He J, Lu M, He B. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture[J]. Proceedings of the VLDB Endowment, 2013, 6(10): 889-900

[13] Petrides P, Diavastos A, Christofi C, et al. Scalability and Efficiency of Database Queries on Future Many-Core Systems[C]// 2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). IEEE, 2013: 24-28

[14] Kim C, Sedlar E, Chhugani J, et al. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs[J]. PVLDB, 2009, 2(2): 1378-1389

[15] Gray J, Sundaresan P, Englert S, et al. Quickly Generating Billion-Record Synthetic Databases[J]. ACM Sigmod Record, 1994, 23(2): 243-253