一种基于GPU集群的深度优先并行算法设计与实现

余 莹[1,2] 李肯立[2] 郑光勇[1]
(衡阳师范学院计算机科学系 衡阳 421002)[1] (湖南大学信息科学与工程学院 长沙 410082)[2]

摘 要 深度优先搜索算法在GPU集群中大型图上的简单执行,会导致线程间的负载不平衡和无法合并内存访问的情况,这使得算法的性能较低。为了明显提高算法在单个GPU和多个GPU环境下的性能,在处理数据之前通过采取一系列有效的操作来进行重新编排。提出了构造线程和数据之间映射的新技术,通过利用前缀求和及二分查找操作来达到完美的负载平衡。为了降低通信开销,对DFS各分支中需要进行交换的边集执行修剪操作。实验结果表明,算法在单个GPU上可以尽可能地实现最佳的并行性,在多GPU环境下可以最小化通信开销。在一个GPU集群中,它可以对含有数十亿节点的图有效地执行分布式DFS。

关键词 GPU,深度优先搜索(DFS),分布式算法,CUDA,MPI
中图法分类号 TP391.9 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.1.019

Implementation of Depth First Search Parallel Algorithm on Cluster of GPUs

YU Ying[1,2] LI Ken-li[2] ZHENG Guang-yong[1]
(Department of Compuper Science, Hengyang Normal University, Hengyang 421002, China)[1]
(College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China)[2]

Abstract Straightforward implementation of depth first search algorithm for large graph on GPU cluster, may lead to load imbalance between threads and un-coalesced memory accesses, giving rise to the low performance of the algorithm. In order to achieve improvement of the performance in a single GPU and multi-GPUs environment, a series of effective operations were used to reschedule before processing the data. A novel strategy for mapping between threads and data was proposed, and by using the prefix sum and binary search operations, load balancing was achieved perfectly. In order to reduce the communication overhead, we performed pruning operation on the set of edges which needs to be exchanged at all branches of DFS. Experimental results show that the algorithm can achieve its best parallelism available on a single GPU and minimize communication overhead among GPUs. GPU cluster can effectively perform the distributed DFS on graphs which contain billions of nodes.

Keywords GPU, DFS, Distributed algorithm, CUDA, MPI

1 引言

图形处理单元(Graphics Processing Unit, GPU)是解决数值问题一个非常高效的平台,它在内存中有着定期访问模式和高算术强度[1]。另一方面,许多图算法如深度优先搜索(DFS)存在不规则内存访问模式和低运算强度[2,3]。单一来源最短路径(Single Source Shortest Path, SSSP)和强连通分量(Strongly Connected Component, SCC)都很少在多GPU系统中表现出很好的性能。文献[4-6]表明,一个单一的GPU实现可以通过优化线程的使用,超越高端多核CPU系统的性能。由于GPU内存大小的限制,这些应用程序可以访问含有百万计以内顶点数的图。目前,对含有以百万计节点大图的研究是热点。

本文研究目的是证明一个GPU集群可以在含有数十亿节点的图上高效执行分布式DFS。DFS算法虽然简单,但它具有数据密集型应用程序的大多数特性[7,8]。使用支持计算统一架构(Compute Unified Device Architecture, CUDA)的多GPU来完成实现。设备之间的通信是通过消息传递接口(Message Passing Interface, MPI)来实现,保证了在任何互连技术上的可移植性和高效性。

本文第2节提出了分布式DFS算法;第3节首先描述多GPU系统中DFS的简单实现,然后在此基础上提出了一种优化算法,并将算法在不同体系结构中实现的性能水平进行了比较;最后总结全文。

2 分布式DFS算法

图算法具有较低的运算强度,即在整个执行过程中,用于计算的时间只占全部开销的一小部分。在分布式环境中,数

到稿日期:2013-12-26 返修日期:2014-03-15 本文受国家自然科学基金项目(61370095,61370098,61070057,90715029),湖南省教育厅科学研究一般项目(13C074),衡阳市科技局科技发展计划项目(2011KJ22)资助。

余 莹(1982-),女,硕士,讲师,主要研究方向为并行计算,E-mail:yuying_kszx@126.com;李肯立(1971-),男,教授,博士生导师,主要研究方向为并行处理、网格计算、DNA计算以及实时与混成嵌入式系统;郑光勇(1972-),男,硕士,副教授,主要研究方向为网格计算。

据存放在远程存储器中，因此大部分的时间都用来发送和接收数据。此外，包含在图算法中的通信模式是不规则的。消息的大小、发送集合和接收集合在执行过程中都是变化的[9]。因此，通信问题是分布式 DFS 发展的瓶颈。

任务间通信的优化，对分布式体系结构中算法的高效性而言至关重要[10]。在 GPU 系统环境中，需要考虑计算节点的具体特点。目标是访问一个图，该图的节点数可以多到无法存储在单个节点内存的一个全局矩阵中。每个节点只保存整个图的一个子集。

Harish[11]和 Hong[4]使用了静态映射。Hong，Agarwal，Merrill 和 Beamer[4,5,12]使用一个全局位掩码矩阵标记访问的节点，位掩码缩减了全局矩阵的尺寸。然而，该解决方案是不可扩展的，图的最大尺寸受限于适合存储装置的矩阵的最大尺寸。例如，图 500 基准提供的最大尺寸为 2^{40} 个节点，即使每个元素分配一位也至少需要 128GB 的内存，远远超过目前支持的 GPU 的大小[9]。

虽然所有共享存储器的优化加速了本地节点的访问，然而分布式系统中用于执行操作的时间只占总运行时间的一小部分，时间开销主要是用在算法中处理非本地节点的部分。

算法 1 显示了一个分布式 DFS 算法的伪代码。根节点是随机选择的，DFS 搜索从负责根节点的本地任务开始，然后延伸到整个图中其他下一分支边界集（Next Branch Frontier Set，NBFS）中节点的任务。NBFS 中节点的任务执行本地节点边界扩展，与任务所有者交换其他节点的信息，并在需要时更新父节点。另外还建立了一个矩阵（第 17 行）来发送通信（第 20 行）和过滤接收到的节点（第 21－25 行）。

算法 1 分布式 DFS

```
CS(the current branch stack)为当前分支堆栈
NS(the next branch stack)为下一分支堆栈
设 s 为随机选择的起始节点
1. CS←Ø,NS←Ø
2. d[u] ← −1, ∀u∈V
3. p[u] ← −1, ∀u∈V
4. if s is local then
5. d[s]=0,p[s]=s,push(CS,s)
6. end if
7. totlen← 1
8. while totlen> 0 do
9. u ← pop(CS)
10. sendarry ← [],recvarry ← []
11. for each v∈Adj[u] do
12. if v is local then
13. if p[v] ==−1 then
14. p[v]=u,push(NS,v)
15. end if
16. else
17. sendarry. append((u,v))
18. end if
19. end for
20. SEND(sendarry),RECV(recvarry)
21. for each (z,w) in recvarry do
22. if p[w] ==−1 then
23. p[w]=z,push(NS,w)
24. end if
25. end for
26. CS←NS,NS←Ø
27. totlen=allreduce(size(CS))
28. end while
```

3 多 GPU 体系结构中的 DFS

为便于理解，首先设计了分布式深度优先搜索问题的一个简单的实现。设计遵循图 500 基准的规范，利用 R-MAT 发生器提前产生一个边的列表。实际的基准测试包括两个部分：(1)kernel1，对应表示图数据结构的生成；(2)kernel2，对应图的搜索[9]。

利用 MPI 实现节点之间的通信。计算和通信通过使用固定大小的信息缓冲区进行重叠。通过对缓冲区的大小调整，使得在下一块非本地节点交换时，每个任务都有足够的节点来进行本地处理。

3.1 图的生成和数据结构

利用 RMAT 生成器[13]生成一个合成图。图中每个节点用一个 64 位的整数表示。对 GPU 而言，内存是有限的资源，这一要求必须严格限制。图必须是无向的，将边数乘以 2（自循环是不可复制的）。生成 $N=2^{scale}$ 个节点和 $M=16\times N$ 条边，每个边连接两个节点，所以元素的总数是 32×2^{scale}。目前一个 Nvidia GPU 能达到 6G 字节的主存，理论上 *scale* 的最大规模为 24。然而，进行 DFS 还需要额外的数据结构。因此 *scale* 无法取到最大值。此外，数据结构一旦创建就不能被修改。最后能够运行在每个设备上的 *scale* 最大值为 21。

通过一个简单的规则给边分配任务：$edge(U_i,V_j)\in Pk if\ U_i\%\#P==k$，其中 $\#P$ 为任务数，%为模运算符。数据结构直接创建在 GPU 上。使用压缩稀疏行（Compressed sparse line，CSR）数据结构来表示图，因为其简单且降低了存储要求。CSR 数据结构由两个矩阵组成：偏移矩阵（Offset Array）和包含图中所有节点的邻接表（Adjacency Lists）。

3.2 简单的实现

执行分布式 DFS 的简单实现使用了基于堆栈的方法与基本操作算法 1。DFS 输出的是父节点矩阵。寻找父节点矩阵比计算距离矩阵的开销更大，因为父节点矩阵的信息在 DFS 的每一个分支都必须存储和交换。当前分支的堆栈（Current Branch Stack，CS）和下一分支堆栈（Next Branch Stack，NS）作为两个单独的矩阵，还有两个额外的矩阵 *sendarry* 和 *recvarry* 存储发送和接收的节点。在 DFS 的每一个分支中，下一个分支集合的所有节点都在内存中被访问，不需要额外的矩阵。

堆栈中的每个节点分配给一个 CUDA 线程，每一个线程访问其节点的邻节点。首先验证邻节点是否在本地。如果邻节点都不在本地，则它们会和父节点一起发送到各自的所有者。对于本地邻节点，其父节点矩阵会被检查是否已经被访问过。未被访问过的节点会添加到 *NS*。为了保持堆栈的一致性，入栈机制依赖于基本操作算法 1。每个任务都发送和接收边；对于每条接收到的边，如果需要会进行检查并使边的第一个节点入栈。

但该简单算法存在如下问题：第一，线程之间的工作量不均衡。如线程 1 访问 $L1$ 个元素的邻接表，而线程 2 访问 $L2$ 个元素的邻接表。在真正的图中，两个邻接列表 $L1$ 和 $L2$ 可能相差几个数量级。此外，存储器访问模式通常都不规则，而且属于同一个块的线程可能需要访问不连续的或彼此相隔很

远的内存区域。存储器访问的数目取决于 NBFS 元素的数目，远远大于线程数。第二，存在通信问题。在 CSR 数据结构中，像任何图的简单表示，邻接表包含了相同节点的多个副本。因此，NBFS 可能含有相同节点的多份副本。这些副本会直接发送给它们的所有者，从而产生了无用的通信负担。第三，非常依赖基本操作算法 1，而这对 GPU 而言是非常大的开销。

3.3 多 GPU 平台上优化的 DFS

对算法进行优化，需要解决两个问题：1)线程间的负载不平衡；2)重复数据的通信。

在 DFS 堆栈中，算法通过简单的映射将线程分配给节点，将会直接导致工作量的失衡，这个问题在大图中后果更加严重。在大图中，堆栈中的元素的数量和要被访问的每个 DFS 分支的数量，可能相差几个数量级。文献[11]中描述了一个复杂的方法，其通过在合作线程矩阵(Cooperative Thread Array，CTA)层构造线程和数据之间的映射，减少了负荷不平衡。然而，优化主要依赖于位图的使用。

通过充分利用 GPU 的并行性来直接解决问题。作为 NBFS 中的元素，应该有许多活动线程，每个线程只负责一个节点，整个 NBFS 可以进行并行处理。线程到 NBFS 元素的映射，通过利用前缀求各运算和二分查找功能，来达到完美的负载平衡。映射允许建立、并行、邻接矩阵来表示 NBFS。在 3.2 节中指出，NBFS 可能含有相同节点的多份副本。这些副本直接发送给它们的所有者，因此制造了一个无用的通信开销。多个副本可以通过进行修剪程序来删除，即利用排序和去重操作来删除 NBFS 矩阵中所有的复本。这个策略有两大优势：第一，减少了交换元素的数目和处理节点的数量。第二，减少了需要将本地节点入栈的基本操作次数。实际上，通过在整个 NBFS 中执行修剪程序，删除了本地节点的副本，从而减少了在本地入栈阶段需要处理的元素数量。

3.4 算法综述

该算法是基于堆栈的，返回值为父节点矩阵。从堆栈开始，建立一个偏移矩阵，计算 NBFS 中的元素总数 m。然后开始 m 个线程。每个线程计算它将处理的元素的 CSR 索引，并行读取 NBFS，将相同节点的多份副本删除。最后与其他任务交换节点，访问新的节点，并更新父节点矩阵。当所有的堆栈为空时，该算法停止运行。在 DFS 算法的每一次循环中执行以下步骤。

(1)建立一个偏移矩阵 $offset$，计算 NBFS 中元素的总数 m。对当前分支堆栈中的每一个元素启动一个线程。建立矩阵 Q_{degree}，用节点的度的值(即 degree 值)来替换每个节点。然后，通过对 Q_{degree} 执行前缀求和操作，来建立新偏移矩阵 $NewOffset$。新偏移矩阵中的最后一个元素为元素的总数，在 NBFS 中为 m。建立另一个矩阵 Q_{offset}，在 CSR 数据结构中用它的起始偏移量来替换每个节点。

(2)建立线程与 NBFS 项之间的映射。建立 NBFS 的连续矩阵。在这一步使用 m 个线程。每个线程在新偏移矩阵 $NewOffset$ 中执行二分查找，通过利用存储在 Q_{offset} 中的老偏移量来计算 CSR 的邻接表矩阵中项的索引，如下：

$i=binsearch(NewOffset, thread\_id, nelements)$；

$t\_off=thread\_id-NewOffset[i]$；

$index=Q_{offset}[i]+t\_off$；

其中，$thread\_id$ 是全局线程标识的索引，$nelements$ 是新偏移矩阵 $NewOffset$ 中元素的个数。二分查找函数返回的是新偏移矩阵 $NewOffset$ 中项的索引，其值小于等于 $thread\_id$ 的值。

然后，每个线程读取邻接表的元素，对应于索引写入一个新的矩阵：下一分支边界矩阵(the Next Branch Frontier array，NBFA)，有 m 个元素。最后给定堆栈中的所有邻节点一个相邻矩阵。

(3)修剪 NBFA。使用 m 个线程来对 NBFA 进行排序。当矩阵排好序后，就很容易简化成 n 个唯一的元素。矩阵修剪前后元素数量之间的比例有可能很小。因为建立一个父节点矩阵，需要有 NBFA 中每个节点的父节点的信息，所以要对节点进行排序，以保持父节点的有效负载。

(4)节点与其他任务进行交换，更新父节点矩阵。将所有被发送的边存储在矩阵 $EdgesToSend$ 中，并按照每条边的第一个节点的所有者进行排序。如果它的所有者为任务 j，任务 i 就发送给矩阵 $EdgesToSend$ 中的任务 j。然后，任务就等待，直到它接收到来自其他任务的所有边，并把他们收集在矩阵 $EdgesRecv$ 中。矩阵 $EdgesRecv$ 和矩阵 $EdgesToSend$ 的本地部分显然都只包含本地节点。最后，那些未被访问过的本地节点被添加到下一分支堆栈。发送和接收的元素数目与不同 DFS 分支中的任务到任务是不同的。为了处理这种情况，在每个 DFS 分支都使用 MPI 来了解要被接收的节点的数量。

简单的经典模型(如 PRAM)不能为现代的并行体系结构提供性能的真实界限[8,14]。然而，为了解决算法的复杂性，对上述的 4 个阶段算法的复杂度进行评价，假定由一个单一的任务执行。前两个步骤中开销最大的操作是二分查找，在最坏的情况下的执行性能为 $O(|M|\log(|V|))$。在第三步排序操作中，通过一个基数排序来实现，最坏情况下的复杂度约为 $O(64\times M)$，因为每个元素编码为一个 64 位的整数。最后第四步的复杂性度约为 $O(M+N)$，因为 $\log(|V|)$ 总是小于 64，整个算法的复杂度不超过 $O(M+N)$。

3.5 实验结果

在 64 个 NvidiaGPU 环境下，将排序-去重算法与简单实现算法进行比较(见图 1)，排序-去重 DFS 比简单算法的速度要快近 4 倍。64 个 GPU 每秒可以遍历约 1 亿条边。

图 2(c)显示了 32 个任务的时间分解算法。很明显，所有的 CUDA 内核总和花费了大部分时间。图 2(b)显示了任务之间的计算和通信是平衡的(这里得到的是 16 个任务的结果，但当任务数更多时这种状况没有改变)。图 2(a)显示了 32 个 GPU 环境下一个任务运行时计算和通信的时间。修剪程序(排序和去重)不仅是最耗时的一部分，而且它实际上控制着运行的时间。点对点通信是该算法第二个最耗时的部分，而二分查找相对耗时要少一些。数据到线程的映射已被证明是非常有效的，而且也可以使用在其他负载不平衡和不规则内存访问的情况。

表 1 显示的是算法计算和通信部分的时间。第一列表示图的大小：$|V|=2^{SCALE}$；第二列表示 GPU 的数量；第三列和第四列分别表示所有 DFS 分支在计算(CUDA 内核)和通信(MPI)上的执行时间。从表中可以看出，当 GPU 的数量增加时，计算时间几乎不变，但通信时间会增加。原因在于，分配给每个 GPU 的子图大小近似一个常量即 $|V|\simeq 2^{21}$(当 GPU 数量为 1 时，取等于值)。

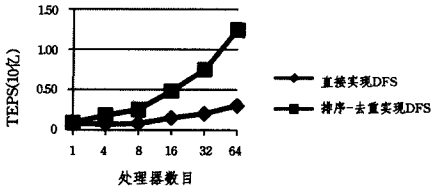
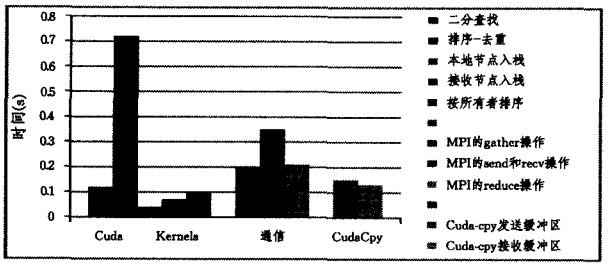
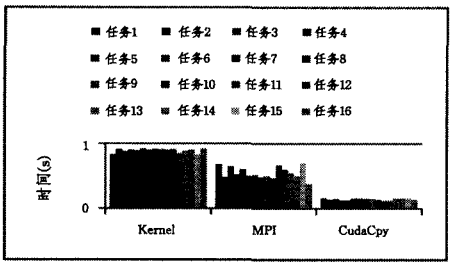


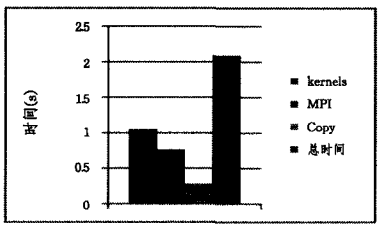
图 1　不同 DFS 算法的实验结果比较



(a)32 个 GPU 环境下,算法不同部分的运行时间



(b)所有内核和所有 DFS 分支通信运行时间之和,每一柱形代表一个不同的任务,16 个任务



(c)算法不同部分的运行时间之和

图 2

表 1　算法计算和通信部分的时间

SCALE	GPU 数量	计算时间(秒)	通信时间(秒)
21	1	0.78	0.0
22	2	0.95	0.1
23	4	0.95	0.4
24	8	0.95	0.6
25	16	1.0	0.7
26	32	1.05	0.8

结束语　为了改善 DFS 实现性能特别是最小化通信负担,尽可能多地减少冗余是必要的。因此,所提算法在每个 DFS 分支,对需要访问的每个 NBFS 节点集都执行了修剪操作。这一步减少了新节点入栈所需的工作量和在不同任务间进行交换的消息大小。

另外,为了充分利用 GPU 的计算能力,如何使用足够大数量的线程和平衡工作负载是需要解决的主要问题。出于这个目的,我们设计了一个数据元素到二进制 CUDA 线程的新映射,使用了二分查找功能。这种映射允许处理 NBFS 的所有边,通过使用集合中每个元素的一个 CUDA 线程,使可用的并行性得到了最好的利用,从而达到完美的负载平衡。新的映射允许对每个 DFS 分支全局矩阵进行有效填充,该矩阵包含堆栈中所有节点的邻节点,并要求执行修剪过程(排序和去重操作)。通过基数排序来提高排序和去重操作的效率。

在执行 DFS 操作期间,NBFS 的修剪花费了很多的执行时间。此外,整个计算时间大于通信时间。实验表明,计算时间和通信时间的比率可以通过增加任务的数量得以减少。当图的大小增加时,相应也使用了更多的 GPU,且增加了任务间进行消息交换的数量。为保持使用上千个 GPU 时的良好可扩展性,需要进一步提高通信机制。为了实现这个目标,许多研究提出一个 2D 分区图以减少参与通信的处理器数量,这样的分区原则上可以在代码中实现,它将会是我们未来工作的研究方向。

参 考 文 献

[1] 王海峰,陈庆奎. 图形处理器通用计算关键技术研究综述[J]. 计算机学报,2013,36(4):757-772

[2] Daniel B, Maxim L, Carlos Linares L, et al. Anytime AND/OR depth-first search for combinatorial optimization[J]. AI Communications, 2012, 25(3): 211-227

[3] Hera J-H, Ramakrishna R S. An external-memorydepth-first search algorithm for general grid graphs[J]. Theoretical Computer Science, 2007, 374(1-3): 170-180

[4] Hong S, Oguntebi T, Olukotun K. Efficientparallel graph exploration on multi-core cpu and gpu[C]// 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), 2011: 78-88

[5] Grimshaw A, Merrill D, Garland M. High Performance and Scalable GPU Graph Traversal[R]. Technical Report, Nvidia, 2011

[6] 吴鸿伟,汤伟宾,李晓潮,等. GPU 编程原理及其在网络安全领域的应用算法分析[J]. 计算机科学,2012,39(11):24-27

[7] AlfonsLaarman, van de Pol J. Variations on Multi-Core Nested Depth-First Search[J]. Electronic Proceedings in Theoretical Computer Science, 2011, 72: 13-28

[8] Chao Li-wen, Sen Yan-hong. Depth-first search algorithm based on special properties of PFSP[J]. Control Decis, 2009, 24(8): 1203-1208

[9] Mastrostefano E, Bernaschi M. Efficient breadth first search on multi-GPU systems[J]. Journal of Parallel and Distributed Computing, 2013, 73(9): 1292-1305

[10] 张庆科,杨波,王琳,等. 基于 GPU 的现代并行优化算法[J]. 计算机科学,2012,39(4):304-311

[11] Harish P, Narayanan P. Accelerating large graph algorithms on the gpu using cuda[C]// Aluru S, Parashar M, Badrinath R, et al., eds. High Performance Computing-HiPC2007. Heidelberg: Springer, Berlin, 2007: 197-208

[12] Agarwal V, Petrini F, Pasetto D, et al. Scalable graph exploration on multicore processors[C]// 2010 International Conference for High Performance Computing, Networking, Storage and Analysis. SC: 2010: 1-11

[13] Leskovec J, Chakrabarti D, Kleinberg J, et al. Kronecker graphs: an approach to modeling networks[J]. Journal of Machine Learning Research, 2010, 11: 985-1042

[14] 卢风顺,宋君强,银福康,等. CPU/GPU 协同并行计算研究综述[J]. 计算机科学,2011,38(3):5-9,46