多核 CPU-GPU 协同的并行深度优先算法*

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

摘 要：针对多核 CPU 和 GPU 环境下图的深度优先搜索问题，提出多核 CPU 中实现并行 DFS 的新算法，通过有效利用内存带宽来提高性能，且当图增大时优势越明显。在此基础上提出一种混合方法，为 DFS 每一分支动态地选择最佳的实现：顺序执行；两种不同算法的多核执行；GPU 执行。混合算法为每种大小的图提供相对更好的性能，且能避免高直径图上的最坏情况。通过比较多 CPU 和 GPU 系统，分析底层架构对 DFS 性能的影响。实验结果表明，一个高端 single-socket GPU 系统的 DFS 执行性能相当于一个高端 4-socket CPU 系统。

关键词：多核 CPU；GPU；深度优先搜索；并行；异构
中图分类号：TP391.9 **文献标志码**：A **文章编号**：1001-3695(2014)10-2982-04
doi:10.3969/j.issn.1001-3695.2014.10.023

Parallel depth first search algorithm on multicore-CPU and GPU

YU Ying[1], LI Ken-li[2]
(1. *Dept. of Compuper Science, Henyang Normal University, Henyang Hunan* 421002, *China*; 2. *College of Information Science & Engineering, Hunan University, Changsha* 410082, *China*)

Abstract: In order to solve the depth first search on multi-core CPU and GPU environment, this paper put forward a kind of parallel DFS algorithm on multicore CPU . Through effective utilization of memory bandwidth to improve performance, and enhanced its advantage as the size of the graph increased. Then the paper proposed a hybrid method which offered dynamical choices from a sequential execution, two different algorithms of multi-core execution, and a GPU execution, for each branch of DFS best implementation. Such hybrid method could provide the best performance for each size of the graph, and avoided the worst-case performance on high-diameter graphs. Finally, the paper compared the multiple CPU and GPU systems to analyse the influence of the underlying architecture on DFS. Experimental results show that a high-end GPU system on DFS perform as well as a quad-socket high-end CPU system.

Key words: multi-core CPU; GPU; depth first search(DFS); parallel; heterogeneous

当今多核 CPU 已广泛应用到高性能计算和消费者电子产品领域，将 GPU 用于通用计算也成为了目前的热点问题[1]。并行(CPU 和 GPU 上的多个线程)和异构(同步使用 CPU 和 GPU)的扩展，已经大大提高了许多传统计算密集型工作的性能[2]。然而，实际上存在一些有效并行计算或异构实现尚未被确定而需要快速计算的问题，图搜索就是一个重要的例子。图是一个基本的数据表示，广泛应用于很多领域，如情报分析[3]、机器人[4]、社会网络分析[5]和计算生物学[6]等。由于这些应用程序的数据集非常大，因此使用传统方法处理需要很长的时间。并行不能缓解该问题，因为这些应用程序的并行加速严重受限于内存访问模式的随机性，而这是一个图处理算法的基本属性[7~9]。

本文主要思想来自于前期的相关研究工作[10,11]，通过利用异构系统中的 CPU 和 GPU，将它们合并到一个通用的解决方案中。

1 并行 DFS 的实现

1.1 多核 CPU 的一种新方法

Hong 等人[10]和 Agarwal 等人[11]提出了多核 CPU 中的一种实现，应用了一系列的优化技术，本文在此基础上提出了一种类似的实现方法，伪代码如算法 1。代码中 Bitmap V(第 3 行)对访问的节点集进行编码。因为该集合在算法中是最经常访问的数据，使用位图数据结构能最小化它的大小，允许缓存来保存该集合最大可能的部分。通过使用内在原子操作和 test and test-and-set 操作来最小化位图并行访问的开销。下一分支节点首先存储在每个线程的本地堆栈(LS)中，直到它们批量插入到全局堆栈。这种批量插入可以通过使用一个单一的原子操作来有效实现，首先增加堆栈索引，然后对前面的索引执行正常的内存拷贝。这是有可能的，因为在这个阶段，对任何线程堆栈都只进不出。本文将算法 1 中描述的方法称为基于堆栈算法。此算法的最终优化是使用一个巧妙的堆栈实现，虽然这种优化提供了性能收益，当输入数据集变得非常大且性能主要由容量失效控制时，性能收益就变得不明显了。

算法 1 基于堆栈的 DFS 实现

```
DFS_Stack(G,w){
  Stack N,C,LS[threads];
  Bitmap V;//访问节点集
  N.push(w);V.set(w.id);
```

收稿日期：2013-10-12；**修回日期**：2013-12-02 **基金项目**：国家自然科学基金资助项目(61370095, 61370098, 61070057, 90715029)；湖南省教育厅科学研究项目(13C074)；衡阳市科技局科技发展计划项目(2011KJ22)；湖南省教育科学“十二五”规划课题(XJK014CGD006)

作者简介：余莹(1982-)，女，讲师，硕士，主要研究方向为并行计算(yuying_kszx@126.com)；李肯立(1971-)，男，教授，博导，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.

```
    Int branch = 1; w. br = branch;
    While( N. size( ) > 0) {
    swap( N, C); N. clear( ); //将下一分支与当前分支进行交换
    fork;
      Foreach( c; C. partition( tid) ) {
        Foreach( n; c. nbrs) {
        if ( ! V. isSet( n. id) ) {//test and test-and-set 操作
          if( V. atomicSet( n. id) ) {
            n. br = branch + 1;
            LS[ tid]. push( n); //线程 tid 的当前堆栈
            if( LS[ tid]. size( ) = THRESHOLD) {
              N. safeBulkPush( LS[ tid]); //全局堆栈
              LS[ tid]. clear( );
    } } } } }
    if( LS[ tid]. size( ) > 0) {
      N. safeBulkPush( LS[ tid]);
      LS. clear( );
    }
  join;
  branch + + ;
} }
```

因此,接下来采取另一种方法,专注于高效使用内存带宽。由于 GPU 体系结构的特点,该方法不使用共享堆栈,而是用一个一维数组 $O(N)$ 来表示一个节点是属于当前分支集合、下一层分支集合还是已访问集合。该数组在每一分支迭代时不断地被访问,充分利用了 GPU 的内存带宽。伪代码如算法 2。与基于堆栈算法类似,同样把已访问集合作为一个位图,通过相同的原子更新操作来访问它。与基于堆栈算法不同的是,在 GPU 实现中下一分支集合和当前分支集合作为一个一维数组 $O(N)$ 被同时执行。本文称这种方法为基于读取算法。

算法 2　基于读取的 DFS 实现

```
DFS_Read( G, w) {
  Bitmap V;
  Bool fin[ threads];
  V. set( w. id);
  Int branch = 0; w. br = branch;
  While( ! finished) {
    fork;
      fin[ tid] = true;
      Foreach( c; G. Nodes. partition( tid) ) {
        if ( c. br! = branch) continue;
        Foreach( n; c. nbrs) {
          if( ! V. isSet( n. id) ) {//test and test-and-set 操作
        if( V. atomicSet( n. id) ) {
        n. br = branch + 1;
        fin[ tid] = false;
    } } } }
  join;
  finished = logicalAnd( fin, threads);
  branch + + ;
} }
```

基于读取算法主要有两个优点:a)它完全不需要堆栈的开销,不仅删除了之前用于堆栈操作的原子指令,而且还保存在缓存和内存带宽中;b)它的内存访问模式更有序,在分支和邻接表的搜索过程中,最大化了顺序读取的概率。

表 1 中显示了四种机器顺序和随机读取带宽的测量结果。从表 1 中可以看出,多核 CPU 系统中顺序和随机读取的带宽相差 9 倍,进一步突出了基于读取算法中顺序内存访问模式的重要性。然而基于读取算法并不能完全消除随机读取访问。该算法仍然需要从邻接表到目标节点额外的间接方法,这是一个固有的随机操作。基于堆栈算法也对位图执行了相同的随机访问。基于读取算法的主要缺点是它在每一分支的迭代都会读取整个分支数组,即使只有少数节点属于这一分支。然而,当对最坏情况的数据输入使用基于读取算法时,这个缺点会导致不好的性能。

表 1　四种机器内存读写带宽

机器类型	顺序读取/GBps	随机读取/GBps
Nehalem CPU	8.6	0.98
Core CPU	3.0	0.25
Fermi GPU	76.8	2.71
Tesla GPU	72.5	3.15

1.2　混合方法

为解决这个问题,本文提出一个混合方案,可以在处理每一分支时动态确定应用的方法。基本想法很简单:如果当前分支只包含几个节点,则使用基于堆栈算法(算法 1),否则使用基于读取算法(算法 2)。

该混合方法可以作为一个状态机,如图 1 所示。混合方法始于顺序状态,即从第 1 分支开始顺序处理。之后,如果下一个分支的大小大于 T_1,就转向堆栈状态,且该分支以基于堆栈算法并行处理。类似地,当下一个分支的大小大于 T_2 时退出堆栈状态,用基于读取算法处理(算法 2)。在 test and test-and-set 操作之后,下一分支的大小可以使用私有计数器、在同步合并时很容易计算。还有一个额外的堆栈和读取之间的过渡状态。在该过渡状态,当前分支集合是从当前分支堆栈中读取,但下一分支的节点不是写回给堆栈,而是给分支数组。如果下一分支被检测的节点数呈指数增长且大于 T_3 时,就退出堆栈。这是因为到达读取状态需要通过一个额外的过渡状态,且后续分支的节点数量增长非常迅速,当最终到达读取状态之前,当前分支中将有足够的节点。

从读取到堆栈的过渡状态也是类似的。当前分支集合变小或节点数停止呈指数增长时,退出读取状态(只有当通过指数增长到达读取状态时,才会检查第二个条件)。这种方法称为读取和堆栈的混合方法,如图 1 所示。该方法通过对节点较少的分支应用基于堆栈算法,避免了基于读取算法最坏情况下的低效率问题。设

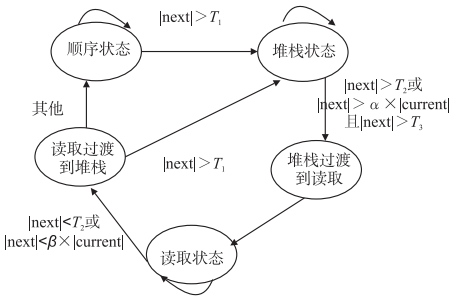

图1　读取和堆栈混合方法的CPU执行状态
(next为下一分支节点数, current为当前分支节点数)

$$(T_1, T_2, T_3) = (64, \max(2^{18}, N \times 0.01), 2048)$$

$$\text{and}(\alpha, \beta) = (2.0, 2.0)$$

混合方法的思想可以应用到 GPU 中 DFS 的实现[12]。图 2

显示了在 GPU 上实现读取和堆栈的混合方法。从根本上讲，当前分支的大小大于 T_4 时（默认值为 16384），前几个分支在迁移到 GPU 之前是在 CPU-side 上执行。GPU 只有在每一分支节点的数量呈指数增长时才启动执行，否则就退回到图 1 的堆栈状态。这是因为在高直径图（如二维网格）的每一分支没有足够的并行性使 GPU 大规模并行硬件达到饱和。同时，一旦 GPU 启动执行，直到完成前都不会返回到 CPU。因为移动回到 CPU 的会带来如下开销：a）需要在 GPU 执行过程中计算下一分支集合的大小，用来确定何时返回 CPU；b）在 CPU-side 需要重建被访问集合的位图。

在图 2 中的初始化 GPU 阶段，整个分支数组不是简单地复制到 GPU。相反，在 CPU-side 处理期间需要保存当前分支堆栈的所有内容，并复制这些堆栈到 GPU。使用这些堆栈，GPU 可以重建整个分支数组，这个操作比从 CPU 复制整个分支数组 $O(N)$ 更快。这个方案称为 CPU 和 GPU 的混合方法（图 2）。

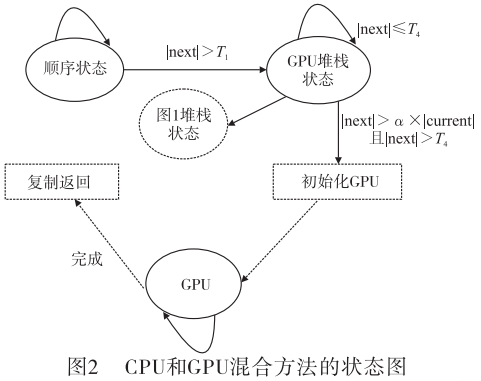

图2 CPU和GPU混合方法的状态图
（next为下一分支节点数，current为当前分支节点数）

本章提出了 DFS 实现的一系列新方法。基于读取算法很简单且对大型图可以利用有效的内存带宽；混合方法可以防止在处理小图或大直径图时出现最坏情况的执行模式。

2 实验方法

在实验中，通过各种机器和不同的图例测量了第 1 章中提到的 DFS 实现。对于输入的数据，使用 RMAT 模型图形生成器，生成给定节点数（N）和边数（M）的图例。生成器来自 SNAP[13] 图库，并使用 SNAP 库的默认值：$(a,b,c)=(0.45, 0.25, 0.15)$。

使用 CSR（压缩稀疏行）格式表示图，将所有节点的邻接列表合并到一个 $O(M)$ 大小的一维数组，将每个节点的邻接表的开始位置存储在一个 $O(N)$ 大小的一维数组中。由于低内存要求且简单，这个数据结构被广泛使用在很多之前的文献中[12,14,15]。每个节点的 DFS 分支是存储在一个单独的 $O(N)$ 字节的数组中。代码利用 NVCC 3.2 来编译。主要实验是用 Nehalem CPU 和 Fermi GPU 两个机器来进行的。

实验有两个主要目标：a）评估新方法的有效性；b）研究多类 GPU 和 CPU 环境下体系结构对 DFS 性能的影响。

通过执行 DFS10 次，随机选择 10 个不同的根节点来测量性能。然而，所有的根节点都属于相同的连通分量，大小为 $O(N)$。结果取多个测量的平均值。当测量 GPU 性能时，不包括在 GPU 内存设置图数据结构（即节点和边）的时间，因为这个图在执行期间不会发生突变，这一步类似于 CPU 从文件系统加载图到主存，但是包括在 GPU 内存初始化 DFS 分支数组的时间和将最终数值复制回 CPU 内存的时间，因为这些步骤每次 DFS 执行时都会重复。

3 实验结果

1）验证第 1 章提出的实现方法的有效性

使用 RMAT 图例，含有 3 200 万个节点和 2.56 亿条边，在 Nehalem CPU 机器上测量了这些方法的性能，结果显示在图 3 中，y 轴是测量的性能（每秒处理的边数），x 轴是所用线程的数量。

从图 3 中可以看出，基于读取算法的性能要优于基于堆栈算法。性能差异的主要原因是基于读取算法节约了内存带宽和更多的顺序数据访问模式。另外，堆栈和读取的混合方法比基于读取算法的性能也稍有改善。

2）测量相同的图例在 Fermi GPU 机器上 DFS 的性能

使用了纯 GPU 执行及 CPU 和 GPU 混合方法。为了比较，图 4 中还显示了堆栈和读取混合方法在 CPU 实现的结果。从图中可以得知，即使最好的 CPU 来实现，GPU 的执行仍然优于 CPU。CPU 和 GPU 混合方法的性能比纯 GPU 稍有改善。

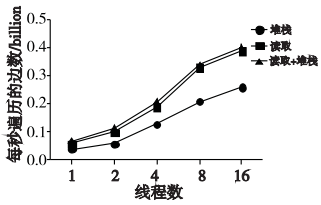

图3 不同DFS执行性能比较（Nehalem CPU）

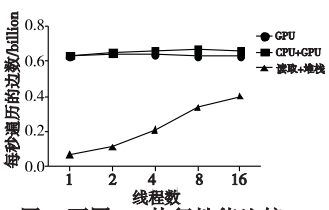

图4 不同DFS执行性能比较（Fermi GPU）

3）分析图大小对 DFS 性能的影响

通过使用 RMAT 图形生成器来生成不同大小的图，并测量其在 CPU 和 GPU 上的性能。在图 5 中，（a）改变节点的数量（从 100 万到 6 400 万），同时保持图的平均度为 8，即 $M=8\times N$。为了表示更大的图，（b）变化图的边数（从 2.56 亿到 20 亿），而节点数固定在 3 200 万。为简便起见，图 5 中省略了纯 GPU 和基于读取算法的结果，因为它们在图 3 和 4 中显示了类似的相对性能。

图 5（a）表明，对于大量的节点，GPU 执行的性能水平基本保持一个常量，而基于 CPU 方法的性能往往会随节点数的增加而有所下降。这是因为随着图尺寸的增加，更多的内存请求离开缓存而被 DRAM 服务。

在图 5（b）中也可以观察到类似的趋势，增加边数时，随着图规模的增长，新方法和之前算法的性能差距扩大。当边的数量超过 5.12 亿时，图不再适应 GPU 内存。

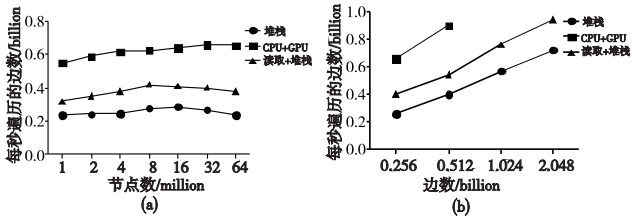

图5 图的大小对性能的影响

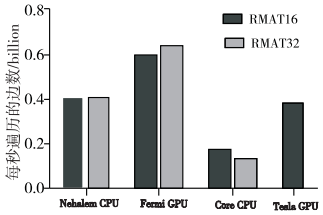

图6 不同机器上DFS执行的性能比较

4)研究机器体系结构对 DFS 执行的影响

在 Nehalem CPU、Fermi GPU、Core CPU 和 Tesla GPU 上执行最佳 DFS 实现:对 CPU 使用读取和堆栈的混合方法;对 GPU 使用纯 GPU 执行。使用了两个大小不同的 RMAT 图(1 600 万个节点与 128 百万条边,3 200 万节点与 2.56 亿条边)。实验结果显示在图 6 中。

首先比较不同 CPU 之间的区别:Nehalem CPU 和 Core CPU。Core CPU 和 Nehalem CPU 之间的性能差异很大,超过 2 倍的差异。可以将这个性能差异归因于这些机器的内存带宽(表 1),而不是缓存大小。

接下来比较两个不同 GPU 之间的区别:Tesla GPU 和 Fermi GPU。Tesla GPU 不能容纳大图实例。对于较小的图例,在相同内存带宽情况下,Fermi GPU 的执行性能要比 Tesla GPU 高出大约 60%。这个改进可以归因于新应用在 Fermi GPU 中的共享最后层(L2)缓存[11]。

总的来说,对于 CPU 和 GPU 系统之间的最后判定并不是唯一的,因为它还需考虑许多其他因素,如功耗、系统的开销和图例的大小以及绝对性能值,而每位研究人员可能考虑这些因素的权重并不完全相同。

4　结束语

随着基于图应用数量和重要性的提高,大型图表正受到越来越多的关注。但大型图的有效处理仍面临挑战[7],其中一个原因是图遍历时的随机内存访问模式。

本文提出了并行 DFS 实现的新方法,其中的多核 CPU 方法能够简单有效地利用应用内存带宽,而且随着图形的增大性能优越性更明显。同时,还提出了一个混合方法,对 DFS 每一分支迭代动态地选择最佳的实现,混合方法对大图和小图都有利,能防止出现最坏情况的性能。实验结果表明,对 DFS 执行来说,一个高端的 GPU 执行的性能相当于一个 4-socket 高端 CPU,影响性能的主要因素是随机存储器访问带宽。

参考文献:

[1] NICKOLLDALLYS J, DALLY W J. The GPU computing era[J]. IEEE Micro, 2010,30(2):56-69.

[2] 卢凤顺,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

[3] COFFMAN T, GREENBLATT S, MARCUS S. Graph-based technologies for intelligence analysis[J]. Communications of the ACM, 2004, 47(3): 45-47.

[4] SIM R, ROY N. Global a-optimal robot exploration in slam[C]// Proc of IEEE International Conference on Robotics and Automation. 2005:661-666.

[5] BRANDES U. A faster algorithm for betweenness centrality[J]. The Journal of Mathematical Sociology, 2001,25,(2):163-177.

[6] TONG A, DREES B, NARDELLI G, *et al*. A combined experimental and computational strategy to define protein interaction networks for peptide recognition modules[J]. Science, 2002,295(5553):321-324.

[7] LUMSDAINE A, GREGOR D, HENDRICKSON B, *et al*. Challenges in parallel graph processing[J]. Parallel Processing Letters, 2007,17(1):5-20.

[8] 许建,林泳,秦勇,等. 基于 GPU 的并行协同过滤算法[J]. 计算机应用研究,2013,30(9):2656-2659.

[9] 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.

[10] HONG S ,OGUNTEBI T, OLUKOTUN K. Efficient parallel graph exploration on multi-core CPU and GPU[C]//Proc of International Conference on Parallel Architectures and Compilation Techniques Digital Object Identifier. 2011: 78-88.

[11] AGARWAL V, PETRINI F, PASETTO D, *et al*. Scalable graph exploration on multicore processors[C]// Proc of ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis. Washington DC: IEEE Computer Society, 2010:1-11.

[12] HONG S, KIM S, OGUNTEBI T, *et al*. Accelerating CUDA graph algorithms at maximum warp[C]//Proc of the 16th ACM Symposium on Principles and Practice of Parallel Programming. 2011:267-276.

[13] BADER D A, MADDURI K. Snap: small-world network analysis and partitioning[EB/OL]. http://snap-graph. sourceforge. net.

[14] HARISH P, NARAYANAN P J. Accelerating large graph algorithms on the GPU using CUDA[C]//Proc of the 14th International Conference on High Performance Computing. Berlin: Springer-Verlag, 2007:197-208.

[15] BADER D, MADDURI K. Snap, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks[C]//Proc of IEEE International Symposium on Parallel and Distributed Processing. 2008:1-12.

(上接第 2968 页)

[6] YONG Long-quan, WANG Jian-liang, ZHANG Guang-hua, *et al*. Novel global harmony search algorithm for absolute value equation [J]. Journal of Information & Computational Science 2012,9(16): 4911-4918.

[7] YONG Long-quan, LIU S Y, FENG Q X, *et al*. Hybrid differential evolution with biogeography-based optimization for absolute value equation [J]. Journal of Information & Computational Science, 2013,10(8): 2417-2428.

[8] SAEED K, HOSSEIN M, SAEED F. Optimal error correction of the absolute value equation using a genetic algorithm[J]. Mathematical and Computer Modelling, 2013,57(9-10): 2339-2342.

[9] YANG Xin-she. A new metaheuristic bat-inspired algorithm[M]// GONZALEZ J R, PELTA D A, CRUZ C, *et al*. Nature Inspired Cooperative Strategies for Optimization. Berlin: Springer, 2010: 65-74.

[10] YANG Xin-she. Bat algorithm for multi-objective optimization [J]. International Journal Bio-Inspired Computation, 2011,3(5): 267-274.

[11] YANG Xin-she, GANDOMI A H. Bat algorithm: a novel approach for global engineering optimization [J]. Engineering Computation, 2012,29(5):464-483.

[12] 刘长平,叶春明,刘满成. 来自大自然的寻优策略:像蝙蝠一样感知[J]. 计算机应用研究,2013,30(5): 1320-1322,1356.

[13] RUBINSTEIN R Y, KROESE D P. The cross-entropy method: a unified approach to combinatorial optimization, monte carlo simulation and machine learning [M]. New York: Springer-Verlag, 2004.

[14] MARGOLIN L. On the convergence of the cross-entropy method [J]. Annals of Operations Research, 2005,134(1):201-214.

[15] KROESE D P, PORTSKY S, RUBINSTEIN R Y. The cross-entropy method for continuous multi-extremal optimization [J]. Methodology and Computing in Applied Probability,2006,8(3): 383-407.

[16] ROHN J. An algorithm for solving the absolute value equation[J]. Electronic Journal of Linear Algebra, 2009(18): 589-599.