计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2020.20200115
57(6): 1152-1163, 2020

面向高通量计算机的图算法优化技术

张承龙[1,2] 曹华伟[1] 王国波[1,2] 郝沁汾[1] 张 洋[1] 叶笑春[1] 范东睿[1,2]
[1](计算机体系结构国家重点实验室(中国科学院计算技术研究所) 北京 100190)
[2](中国科学院大学计算机与控制学院 北京 100049)
(caohuawei@ict.ac.cn)

Efficient Optimization of Graph Computing on High-Throughput Computer

Zhang Chenglong[1,2], Cao Huawei[1], Wang Guobo[1,2], Hao Qinfen[1], Zhang Yang[1], Ye Xiaochun[1], and Fan Dongrui[1,2]
[1](*State Key Laboratory of Computer Architecture* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)
[2](*School of Computer and Control Engineering*, *University of Chinese Academy of Sciences*, *Beijing* 100049)

Abstract With the rapid development of computing technology, the scale of graph increases explosively and large-scale graph computing has been the focus in recent years. Breadth first search (BFS) is a classic algorithm to solve graph traverse problem. It is the main kernel of Graph500 benchmark that evaluates the performance of supercomputers and servers in terms of data-intensive applications. High-throughput computer (HTC) adopts ARM-based many-core architecture, which has the characteristics of high concurrency, strong real-time, low-power consumption. The optimization of BFS algorithm has made a series of progress on single-node systems. In this paper, we first introduce parallel BFS algorithm and existing optimizations. Then we propose two optimization techniques for HTC to improve the efficiency of data access and data locality. We systematically evaluate the performance of BFS algorithm on HTC. For the Kronecker graph with $2^{scale}=2^{30}$ whose vertices are 2^{30} and edges are 2^{34}, the average performance on HTC is 24.26 GTEPS and 1.18 times faster than the two-way x86 server. In terms of energy efficiency, the result on HTC is 181.04 MTEPS/W and rank 2nd place on the June 2019 Green Graph500 big data list. To our best knowledge, this is the first work that evaluates BFS performance on HTC platform. HTC is suitable for data intensive applications such as large-scale graph computing.

Key words breadth first search (BFS); high throughput; Graph500; graph algorithm; super computing

摘 要 随着互联网技术的蓬勃发展,图数据的规模呈爆炸式增长.如何高效地处理大规模图数据逐渐成为工业界和学术界关注的焦点.宽度优先搜索算法是解决图遍历问题的经典算法,也是 Graph500 基准的核心测试程序之一.高通量计算机采用 ARM 架构的众核体系结构,具有高并发、强实时、低功耗等适于大数据计算的特点.在单节点上,BFS 算法的优化已取得一系列进展,首先对现有的优化技术进行

收稿日期:2020-02-27;**修回日期**:2020-04-15
基金项目:国家重点研发计划项目(2018YFB1003501);国家自然科学基金项目(11904370,61732018,61672499);计算机体系结构国家重点实验室创新项目(CARCH4509)
This work was supported by the National Key Research and Development Program of China (2018YFB1003501), the National Natural Science Foundation of China (11904370, 61732018, 61672499), and the Innovation Project of the State Key Laboratory of Computer Architecture (CARCH4509).

系统的介绍,并在此基础上提出 2 种面向高通量计算机的优化手段,通过减少冗余访存和提高缓存局部性,有效提高了算法的访存效率.通过这些优化手段,在高通量计算机上对 BFS 算法的性能进行了系统的评估.对于顶点规模为 2^{30} 的 Kronecker 图(顶点数为 2^{30},边数为 2^{34}),优化后的 BFS 算法在高通量计算机上的平均性能为 24.26 GTEPS.与两路 x86 架构服务器相比,单节点具有 1.18 倍的性能优势.在性能功耗比方面,高通量计算机的结果为 181.04 MTEPS/W.在 2019 年 6 月份的 Green Graph500 面向大数据集的排行榜上取得第 2 名的成绩.综上,高通量计算机的高并发和低功耗等特点非常适合处理大规模图计算等数据密集型应用.

关键词 宽度优先搜索;高通量;Graph500;图算法;超算

中图法分类号 TP301

图具有极强的抽象性和灵活性,是一种表示和分析大数据的有效方法.图计算可进行巨量、稀疏、超维关联的挖掘和分析,已广泛应用于社交网络、交通网络、生物信息网络、知识图谱等[1-4].随着图数据规模的爆炸式增长和图计算需求的不断增加,如何高效地处理大规模图数据成为大数据领域的研究重点,也是工业界关注的焦点.

宽度优先搜索(breadth first search, BFS)算法是一种经典的图遍历算法,也是众多图分析算法的基础,例如连通分量(connected component, CC)、中介核心性(betweenness centrality, BC)以及单源最短路径(single-source shortest path, SSSP)等算法.BFS 应用具有数据局部性差、计算访存比低、并行效率低和扩展性差等数据密集型应用的典型特征.2010 年,国际上提出了 Graph500 排行榜来评估服务器和超级计算机处理数据密集型应用的能力,而 BFS 算法是 Graph500 的基准测试程序之一[5].

针对数据密集型应用,中国科学院计算技术研究所高通量计算机研究中心在 2018 年的中国计算机大会上发布了新一代高通量计算机"金刚"[6].高通量计算机(high-throughput computer, HTC)采用高通量众核体系结构,在一系列体系结构创新技术的加持下,高通量计算机具有高并发、强实时、低功耗等适于大数据计算的特点.目前,高通量计算机已经针对深度学习、高通量视频处理、科学大数据处理和信息安全监测等典型场景开展示范引用.

本文将对单节点下 BFS 算法的优化技术进行系统介绍,并在此基础上提出 2 种面向高通量计算机的优化手段,通过减少冗余访存和提高缓存局部性,有效提高了算法的访存效率.基于这些优化手段,我们对 BFS 算法在高通量计算机上的性能进行系统的评估,同时与通用服务器进行对比,展现高通量计算机处理数据密集型应用的优越性.

1 相关工作

为了获取更高的性能和更低的能耗,近年来 BFS 的并行算法在共享式内存系统、分布式内存系统和 GPU 等平台上都得到了广泛的研究,国际上针对 BFS 算法的优化工作也取得了一系列的进展.

在 Cray 的 MTA 平台上,Bader 等人[4]通过挖掘算法的并行度,采用多线程技术来隐藏遍历过程中的访存延迟.Agarwal 等人[7]针对数据结构进行优化,改善了数据的局部性,提高了算法性能.为了充分利用 GPU 的多线程技术,Hong 等人[8]提出一种面向 GPU 平台的并行 BFS 算法,结合多线程技术和细粒度同步机制提高了系统的整体性能.2011 年,Beamer 开创性地提出了一种自底向上(bottom-up)的遍历算法[9],通过结合传统自顶向下(top-down)的算法,实现了方向性优化的 BFS 算法,通过减小冗余的访存开销,提高了算法性能.2015 年,叶楠等人在在众核体系结构上对 BFS 算法进行了测试,发现当前典型的众核结构无法满足 BFS 算法的需求,新型处理器体系结构需要支撑大量离散随机访问以及更为简单的执行机制[10].2013 年,Yasui 等人针对非统一内存访问架构(non-uniform memory access architecture, NUMA)提出了相应的 NUMA 感知和度数感知优化技术,在 Intel 的四路 Xeon E5-4640 服务器上取得了 29.1 GTEPS 的性能[11].基于上述研究,我们发现 BFS 并行处理时仍然存在严重的负载不均衡性,基于此提出一种静态 shuffle 优化,改善了众核间的负载均衡性,显著提高了线程利用率,同时也避免了动态调度方式的一些缺点[12].

本文在已有工作的基础上,提出了一种高效的基于块查找的位图访问方式,减少了不必要的缓存访存;此外,我们对 Bottom-up 算法的遍历过程做了

进一步优化,有效改善了缓存数据的局部性,提高了访存效率.

本文主要贡献有3个方面:

1) 为了充分利用高通量计算机的大容量缓存,提出一种高效地位图访问方法,显著提高了未访问点的查找效率,能够显著减少对cache的访问,提高寄存器访问的局部性.

2) 针对Bottom-up的核心算法进行了深入优化,将3个bitmap减少为2个bitmap,在此基础上将Degree-Aware中的2次遍历过程合并为一次,提高了缓存局部性和分支效率.

3) 在高通量计算机上对BFS的性能和能耗表现进行了系统评估,并与x86架构的服务器进行了对比.详细分析了所提算法的有效性,优化后的算法显示出性能和性能功耗比优势.

2 算法背景介绍

本节我们将对BFS算法及现有的优化技术进行系统的介绍和总结,包括方向性优化、去零点优化、度数感知优化、顶点排序优化和静态shuffle优化等.最后的实验也是集合了这些优化技术.

2.1 方向性优化

传统的BFS算法是一种自顶向下(top-down)的遍历方法,遍历中后期会产生大量的冗余遍历.Beamer创造性地提出了一种Bottom-up算法来减少冗余边的遍历.如算法1所示.

算法1. Bottom-up BFS算法.

输入:图$G=\{V,E\}$、遍历起始点s;

输出:存储父亲信息的数组*parent*.

① $visit[v]\leftarrow 0$, for $\forall v\in V$;
② $visit[s]\leftarrow 1$;
③ $parent[s]\leftarrow s$;
④ *current frontier*$\leftarrow\{s\}$;
⑤ *next frontier*$\leftarrow\varnothing$
⑥ while *current frontier* not empty do
⑦ for each $u\in V$ in parallel do
⑧ if $visit[u]==0$ then
⑨ for each w adjacent to u do
⑩ if $w\in$*current frontier*
⑪ parent$[u]\leftarrow w$;
⑫ $visit[u]\leftarrow 1$;
⑬ *next frontier*$\leftarrow$*next frontier*$\cup\{u\}$;
⑭ break;
⑮ end if
⑯ end for
⑰ end if
⑱ end for
⑲ swap *current frontier* with *next frontier*;
⑳ *next frontier*$\leftarrow\varnothing$;
㉑ end while
㉒ return *parent*.

Bottom-up采用与Top-down完全相反的思想,在算法1执行的每一层优先选择所有未访问的顶点(行⑧),通过查找其邻居顶点是否位于*current frontier*来替子节点寻找父节点.一旦未访问的顶点在*current frontier*中找到父节点,则跳出循环结束对剩余邻居顶点的访问,有效减少了冗余的访存开销.由此导致Bottom-up算法在初始几层遍历时产生大量的无效遍历.基于此,Beamer等人提出结合Top-down和Bottom-up的方向性优化算法[9].在算法1中,首先使用Top-down算法开始遍历,当*current frontier*队列变得足够大时切换到Bottom-up执行,结合二者的优势,从而显著提高BFS性能.

2.2 去零点优化

在Graph500中,数据集采用的是Kronecker生成图.研究发现[11],Kronecker生成图中存在大量的孤立顶点,即这部分顶点的出度为0.通过统计,孤立顶点在不同规模的生成图中所占比例如表1所示:

Table 1 Ratio of Isolated Vertices in Kronecker Graphs with Different Scales

表1 不同规模下的Kronecker生成图中孤立顶点的比例

2^{scale}	Ratio of Isolated Vertices/%
2^{23}	45
2^{24}	47
2^{25}	49
2^{26}	51
2^{27}	53
2^{28}	54
2^{29}	56

从表1可以看出,孤立顶点在Kronecker生成图中的占比超过50%,并且随顶点规模的增加而增长.Bottom-up算法在遍历每一层的过程中,首先会从所有顶点中找出所有未被访问过的顶点,然后从这些未访问过的顶点出发去搜索其父亲节点.孤立顶点的存在,使得Bottom-up算法在每层遍历中,

产生大量对孤立顶点的无效遍历和无效访存.因此,在预处理中去掉孤立顶点,会进一步提高遍历效率.

2.3 度数感知优化

如2.1节所述,Bottom-up算法遍历中会扫描所有未访问顶点的邻居顶点,只要找到一个邻居顶点在*current frontier*中,则结束对剩余邻居节点的遍历.如果终止的越早,则可以减少邻居检查的数量.理想情况下,Bottom-up方法只检查未访问顶点的第1个相邻节点,就能够检查成功,其后将跳过所有剩余的邻居顶点.然而事实上,我们难以确定邻居列表的最佳顺序,选定不同的源顶点,邻居列表的最佳顺序都可能是不同的.Yasui等人通过实验发现了顶点度数和访问频率之间的关联性,顶点的度数越大,其访问频率越高[13].为了进一步提高访存效率,Yasui将邻接列表A拆分为只含最高度数邻居顶点的A^{B+}与剩余按照度数降序排列的邻接列表A^{B-},将算法1中的一次遍历循环(行⑨~⑬)拆分为采用邻接列表A^{B+}与A^{B-}的2次循环.实验发现,度数感知优化可以极大减少冗余边的遍历,提高数据访问效率.

2.4 顶点排序优化

为了改善数据的局部性、提高访存效率,Yasui等人提出一种顶点排序的优化方法[14],基于顶点度数降序对行列表(row)中的顶点索引进行排序,将较低的索引分配给具有高度数的顶点.尽管顶点排序方法可以改进数据的局部性来减少访存开销,但是在动态调度过程中,数据块粒度和调度开销存在博弈.当块粒度很小时,可以很好地解决线程间的负载均衡问题,但频繁的线程调度将显著提高执行时间的开销,导致整体性能下降.当块粒度很大时,由于图数据顶点度数分布遵循幂律分布[15],因此相邻顶点块中度数分布变化很大.高度数块的处理时间远大于低度数块的执行时间,导致严重的负载不均衡问题.

图1显示了顶点排序方法下线程的平利用率,可以看到采用动态调度的方式每个线程的利用率波动很大,线程间存在严重的负载极不均衡问题.

2.5 静态shuffle优化

针对高通量计算机高并发的体系结构特点,我们在之前的工作中提出了一种按照顶点度数轮询分配的静态shuffle优化方法[12],在利用顶点排序优势的同时,改善了高通量计算机核间的负载均衡性,提高了并行BFS算法在高通量计算机上的性能.

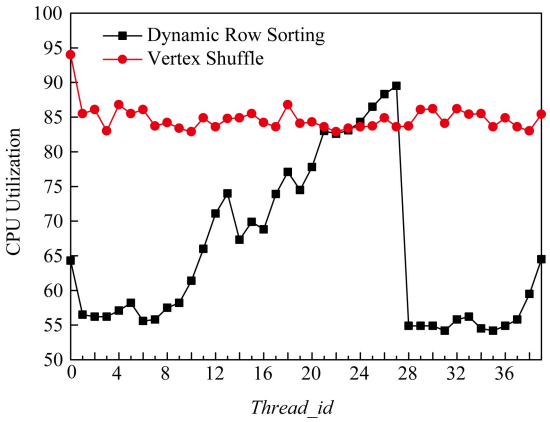

Fig. 1 The thread utilization of two methods when $2^{scale}=2^{27}$

图1 当规模为2^{27}时2种方法的线程利用率

如算法2所示,在图数据的CSR存储结构中,对行列表row中的顶点按度数降序排序,以轮询的方式依次将顶点分配给不同段数组(分别存储线程执行顶点信息),保证高度数顶点与低度数顶点均匀地分配给每个线程,并在线程段数组内依然保持顶点按度数的降序排列.与此同时,建立新顶点ID到旧顶点ID的映射,从而可以在图遍历结束后恢复原始图拓扑的父节点信息.

算法2. 顶点轮询分配的静态shuffle算法.

输入:划分个数*segmentNum*、原始的排好序的row缓存*old_array*;

输出:shuffle之后的row缓存*new_array*、节点shuffle前的编号*new2old*.

① $num \leftarrow vertexNums/segmentNums$;
② for ($tid=0;tid<segmentNums;tid++$) do
③ for ($j=0;j<num;j=j+1$) do
④ $new\_array[tid\times num+j] \leftarrow old\_array[tid+segmentNums\times j]$;
⑤ $new2old[tid\times num+j] \leftarrow tid+segmentNums\times j$;
⑥ end for
⑦ end for
⑧ return *new_array*, *new2old*.

通过上述静态shuffle分配,一方面保持了顶点排序的数据局部性,提高了内存带宽的利用率;另一方面,在预处理时已经将每个线程处理的顶点信息保存下来,避免了频繁的线程调度引起的开销,以及动态分配中块粒度的经验参数调整.

3 BFS 算法优化

针对高通量计算机高并发和大容量缓存的体系结构特点,我们在第 2 节优化工作的基础上,提出了基于块查找的位图访问优化和 Bottom-up 遍历优化 2 种方法,进一步提高 BFS 算法在高通量计算机上的性能.具体来说,我们提高了未访问顶点的搜索效率与片上缓存的数据局部性,有效提高了 BFS 算法的访存效率.

3.1 基于块查找的位图访问

BFS 在遍历过程中采用位图的形式来标记顶点的访问状态.位图是一种紧凑的数据结构,可以放在最后一级缓存(cache)中,有效减少了片外 DRAM 的访问,降低了访存开销.在 Bottom-up 算法中,每次迭代都通过顺序扫描 *visit* 位图来寻找未访问顶点(算法 1 行⑦～⑧).由于 Kronecker 图结构的幂律分布特点,经过几次 Bottom-up 迭代后,未访问顶点的数量将急剧减少,并呈稀疏分布.在处理大规模图数据时,顺序扫描 *visit* 位图将对性能产生极大的影响.

基于此,我们提出一种基于块(block)查找的 *visit* 位图访问优化技术,在每一层的 Bottom-up 遍历中,能快速找到未访问顶点在 block 中的位置.如图 2 所示,在遍历过程中以 *visit* 位图中的 block 为单位进行顶点状态的粗粒度判断.现代通用处理器中通用寄存器的最大位宽为 64 位,最多可以表示 64 个点的访问状态,因此一个 block 的大小设置为 64.在遍历开始时,首先将 *visit* 位图中第 i 个 block 的访问状态从内存中加载到 64 位宽的寄存器 *block_visit* 中.然后,对 *visit* 位图的第 i 个 block 对应的整型变量 *block_visit* 取反,得到寄存器变量 *block_no_visit*,取反后状态位为 0 是已遍历的顶点,状态位为 1 则为待查询的未访问顶点.如果 *block_no_visit* 中的状态位全为 0,则说明当前 block 不包含未访问顶点,直接跳过对当前 block 中所有顶点的状态访问;如果 *block_no_visit* 的状态位不全为 0,通过算法 3 可以快速定位 *block_no_visit* 中最右侧的未访问顶点的位置(最右侧 1 bit 位),并将寄存器 *block_no_visit* 中该点对应的位置置 0,依次循环,从右到左依次访问 *block_no_visit* 中的未访问顶点(bit 位为 1),如图 2 所示.在从 *block_no_visit* 中查找未访问点时,其循环次数仅与 bit 位为 1 的数目(即未访问的顶点数目)有关,而与 block 大小无关.而传统算法在查找 block 中未访问点时采用顺序遍历,即使 block 中只有 1 个未访问点,也需要迭代 block 大小次数.

算法 3. 基于块查找的顶点快速定位.

输入:一个 block 的遍历状态(uint64_t) *block_no_visit*(*block_no_visit*=1 表示尚未遍历,*block_no_visit*=0 表示已经遍历);

输出:一个 block 寄存器变量中最低位未遍历点的位置 *pos*、更新之后的 *block_no_visit*.

① uint64_t *bit_no_visit*=*block_no_visit* & (−*block_no_visit*);

② uint64_t *mask*=*bit_no_visit* −1;

③ *pos*=__builtin_popcountl(*mask*);

④ *block_no_visit*=*block_no_visit* & ～*bit_no_visit*; /* set the lowest unvisited vertex to visited */

⑤ return *pos*,*block_no_visit*.

算法 3 中核心函数的算法复杂度为 $O(\log(k))$,优于传统顺序查找的算法复杂度 $O(k)$.优化后,将原有的细粒度逐位扫描方式,转换为粗粒度的快速定位方式,在 Bottom-up 的后几层遍历过程中,减少了大量的访存操作.此外,BFS 算法的局部性非常差、cache miss 率高,会发生频繁的 cache 替换.本节提出的基于块查找的位图访问方法,将一个 block

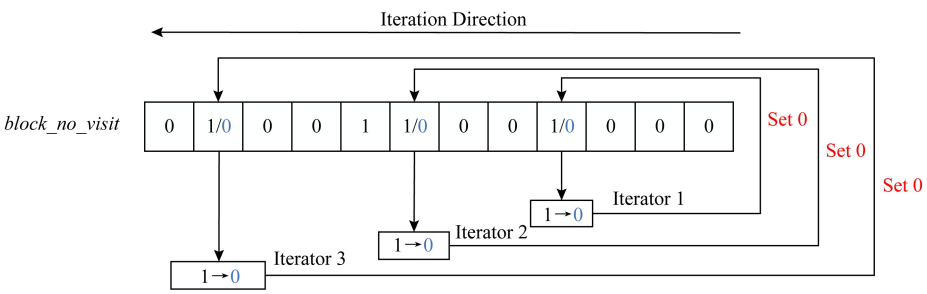

Fig. 2 The quick search for unvisited vertices

图 2 未访问顶点的快速遍历示意图

的位从原有的 *visit* 位图存入寄存器,减少了对 cache 的频繁访问,提高了访问效率.引入块查找位图访问后的 Bottom-up 遍历如算法 4 所示.

算法 4. 使用 3 个 bitmap 的块查找的 Bottom-up 算法.

输入:图 $G=\{V,E\}=\{V,(A^{B+},A^{B-})\}$、遍历起始点 s;

输出:存储父亲信息的数组 *parent*.

① $visit[v]\leftarrow 0$, for $\forall v\in V$;
② $visit[s]\leftarrow 1$;
③ $parent[s]\leftarrow s$;
④ $current\ frontier\leftarrow\{s\}$;
⑤ $next\ frontier\leftarrow\varnothing$
⑥ $BlockNum\leftarrow vertex\_num/BlockSize$; /* 顶点总数 *vertex_num* 除以 block 宽度 *BlockSize* */
⑦ while *current frontier* not empty do
⑧ for $i\leftarrow 0\sim BlockNum$ in parallel do /* 当前层遍历,依次取每个 block */
⑨ $block\_visit\leftarrow getBlock(visit,i)$; /* 获取第 i 个 block 的访问状态,存入寄存器变量 *block_visit* */
⑩ $block\_no\_visit\leftarrow\sim block\_visit$;
⑪ $block\_next\leftarrow 0$; /* block 中顶点是否属于下一层的寄存器变量 *block_next* */
⑫ while $block\_no\_visit\neq 0$ do /* 当前 block 有未遍历且未检测的点 */
⑬ $pos\leftarrow BlockSearchUnvisitedVertex(block\_no\_visit)$; /* block 中最右侧未访问点的位置 *pos* */
⑭ $w\leftarrow i\times BlockSize+pos$;
⑮ $v\leftarrow A^{B+}(w)$;
⑯ if $getBit(current\_frontier,v)$ then /* 如果 v 在当前层 */
⑰ $block\_visit\leftarrow block\_visit\mid(1\ll pos)$;
⑱ $block\_next\leftarrow block\_next\mid(1\ll pos)$;
⑲ $parent(w)\leftarrow v$;
⑳ else
㉑ while $v\in A^{B-}(w)$ do
㉒ if $getBit(current\_frontier,v)$ then /* 如果 v 在当前层 */
㉓ $block\_visit\leftarrow block\_visit\mid(1\ll pos)$;
㉔ $block\_next\leftarrow block\_next\mid(1\ll pos)$;
㉕ $parent(w)\leftarrow v$;
㉖ break;
㉗ end if
㉘ end while
㉙ end if
㉚ end while
㉛ /* 将第 i 个 block 的访问状态写回 *visit bitmap* */
㉜ $writeBlock(visit,i,block\_visit)$;
㉝ /* 将第 i 个 block 中每个点在下一层的状态写回 *next_frontier bitmap* */
㉞ $writeBlock(next\_frontier,i,block\_next)$;
㉟ end for
㊱ $swap(current\_frontier,next\_frontier)$;
㊲ end while
㊳ return *parent*.

表 2 给出了优化前后 Bottom-up 每层遍历的顶点数目对比.可以看出,在遍历到第 5 层时,99.8% 的顶点是已访问状态,顺序扫描 *visit* 位图将产生大量无效访存.优化后,每次以粗粒度的 block 进行判断,对于包含未被访问的顶点,基于块查找的位图访问优化可以高效地定位未访问顶点.相对于顺序扫描,最多能节省 98.4% 的顶点访存状态判断,极大地提升了算法的遍历效率.

Table 2 Number of Traversed Vertices with and without Bitmap Optimization

表 2 位图访问优化前后的顶点遍历数目

Level	Visited Vertex	Pre-check Vertex	Now-check Vertex	Jump Ratio/%
3	335 542	63 080 814	62 745 312	0.53
4	46 515 386	63 080 814	16 867 850	73.26
5	62 965 277	63 080 814	1 007 631	98.40

3.2 Bottom-up 遍历优化

在本节中,我们提出针对 Bottom-up 遍历方式的优化.在原有的 Bottom-up 算法中,需要 *current_frontier*,*next_frontier* 和 *visit* 三个数据结构来保存遍历过程中更新的信息.*current_frontier* 保存当前层的活跃顶点,*visit* 保存已经被遍历过的顶点,*next_frontier* 用于存放下一层将遍历的活跃顶点.事实上,*visit* 中包含处于 *current_frontier* 中的活跃顶点信息,位于 *current_frontier* 中的顶点一定已经在 *visit* 中置位,因此在判断未访问顶点 u 是否有邻居顶点在 *current_frontier* 中时,我们考虑用

visit 位图代替来达到同样的效果,减少额外数据结构的使用.如算法 5 的行⑪～㉑所示,结合使用 *visit* 和 *visit_new*,能够在遍历过程中省去对当前层的跃顶点结构 *current_frontier* 的访问,将 Bottom-up 遍历过程中所需要的 3 个数据结构减少为 2 个,每次迭代只有一次 *visit* 读操作和一次 *visit_new* 写操作,其余操作都是对寄存器 *block_visit* 进行的,提高了 cache 访问的缓存局部性,进而提高了访存效率.

算法 5. 基于块查找的 Bottom-up BFS 算法(block search bottom-up BFS, BSBB).

输入:图 $G=\{V,E\}=\{V,(A^{B+},A^{B-})\}$、遍历起始点 s;

输出:存储父亲信息的数组 *parent*.

① $visit[v]\leftarrow 0$, for $\forall v\in V$;
② $visit[s]\leftarrow 1$;
③ $parent[s]\leftarrow s$;
④ $current\ frontier\leftarrow\{s\}$;
⑤ $next\ frontier\leftarrow\varnothing$;
⑥ $BlockNum\leftarrow vertex\_num/BlockSize$;
/* 顶点总数 *vertex_num* 除以 block 宽度 *BlockSize* */
⑦ while *current frontier* not empty do
⑧ for $i\leftarrow 0\sim BlockNum$ in parallel do /* 当前层遍历,依次取每个 block */
⑨ $block\_visit\leftarrow getBlock(visit,i)$;
/* 获取第 i 个 block 的访问状态,存入寄存器变量 *block_visit* */
⑩ $block\_no\_visit\leftarrow\sim block\_visit$;
⑪ while $block\_no\_visit\neq 0$ do /* 当前 block 有未遍历且未检测的点 */
⑫ $pos\leftarrow BlockSearchUnvisitedVertex(block\_no\_visit)$; /* block 中最右侧未访问点的位置 *pos* */
⑬ $w\leftarrow i\times BlockSize+pos$;
⑭ $v\leftarrow A^{B+}(w)$;
⑮ if $getBit(visit,v)$ then /* 如果 v 在当前层 */
⑯ $block\_visit\leftarrow block\_visit\,|\,(1\ll pos)$;
⑰ $parent(w)\leftarrow v$;
⑱ else
⑲ while $v\in A^{B-}(w)$ do
⑳ if $getBit(visit,v)$ then
/* 如果 v 在当前层 */
㉑ $block\_visit\leftarrow block\_visit\,|\,(1\ll pos)$;
㉒ $parent(w)\leftarrow v$;
㉓ break;
㉔ end if
㉕ end while
㉖ end if
㉗ end while
㉘ /* 将第 i 个 block 更新后的访问状态写回 *visit_new bitmap* */
㉙ $writeBlock(visit\_new,i,block\_visit)$;
㉚ end for
㉛ $swap(visit,visit\_new)$; /* 交换 *visit* 和 *visit_new bitmap* */
㉜ end while
㉝ return *parent*.

另一方面,在 Yasui 等人提出的度数感知优化中[14],阐明了 Bottom-up 算法的主要性能瓶颈是对未访问顶点的邻居顶点的访存开销.他们将邻接列表 A 拆分为只含最高度数邻居顶点的 A^{B+} 与剩余按照度数降序排列的邻接列表 A^{B-},分别遍历 2 个邻接列表来提高邻居顶点在 *current_frontier* 中的搜索效率.我们进一步统计了 Bottom-up 算法在邻接列表 A^{B+} 与 A^{B-} 的遍历效率,测试结果如算法 5 所示.

表 3 可以看到,在 Bottom-up 算法的每层遍历中,通过邻接列表 A^{B+} 更新的活跃顶点数目超过 96%,并且随遍历层数的推进,活跃顶点的比重不断增加.在测试样例的第 5 层遍历中,所有的活跃顶点都是通过邻接列表 A^{B+} 更新.基于此,在我们优化后的 Bottom-up 算法中,我们将对邻接列表 A^{B+} 与 A^{B-} 的遍历合并为一个,在保证 A^{B+} 数据局部性的同时,又减少了对 A^{B-} 结构无效的访存次数.

Table 3 Number of Active Vertices Under Degree-aware Optimization for Graph Scale 2^{27}

表 3 顶点规模为 2^{27} 时度数感知优化下 Bottom-up 每一层遍历的顶点数目

Level	Total Updated Vertex	A^{B-} Updated Vertex	A^{B+} Updated Vertex	A^{B+} Updated Vertex Ratio/%
3	46 179 844	1 860 649	44 319 195	96
4	16 449 891	288	16 449 603	99.99
5	73 664	0	73 664	100

4 实验结果与分析

为了评估高通量计算机处理大规模图数据的能力,我们搭建了2套测试系统用于评估BFS算法的性能.一套基于自研的高通量计算机,配备一颗ARM架构的Centriq 2434众核处理器,该处理器包含40个处理核心和50 MB的L3 cache,TDP为110 W;另一套系统采用传统的x86架构服务器,配有两路Intel Xeon E5-2683处理器,每路CPU包含14个核心和35 MB L3 cache,TDP为120 W.系统的详细配置信息如表4所示,2套系统均使用gcc7.3.0进行编译.

Table 4 Configure Information

表4 系统配置信息

Parameter	High-Throughput Computer	x86 Server
Operation System	CentOs7	CentOs7
CPU	Centriq 2434	Xeon E5-2683 v3
CPU Speed/GHz	2.30	2.00
Socket/s	1	2
Cores per Socket	40	14
L3 Cache Size/MB	50	35
Memory Capacity/GB	384	384
Memory Type	DDR4	DDR4
TDP/W	110	120

测试数据集采用Graph500基准测试中的Kronecker生成图,其参数设置为默认值($A=0.57$, $B=C=0.19$, $D=0.05$).Kronecker生成图可以通过输入*scale*和*edgefactor*等参数来调整图的性质,其中,*scale*参数表示图的顶点规模,*edgefactor*表示每个顶点的平均度数.生成的图数据满足幂律分布,包含2^{scale}的顶点数目以及$2^{scale}\times edgefactor$的总边数.对于每个测试数据图,随机选择64个源顶点执行BFS算法并取平均时间,采用指标每秒遍历亿条边(giga-traversed edges per second, GTEPS)来对性能进行评估.

4.1 高通量计算机介绍

实验用的高通量计算机主要用于数据中心大数据处理[16],其微结构主要包括:1)高吞吐处理核.共有40个处理核,每个处理核采用变长流水线和多发射结构,显著提高了指令吞吐效率,有效地掩盖了图遍历过程中邻居边访问时延.2)大容量片上存储.高通量计算机具有较大容量的L3 cache,大小为50 MB,可以将更大的图放到缓存中,进一步提高了访存局部性.3)易扩展片上网络.处理核通过环形总线进行互联,可以提供250 GB/s的聚集带宽,能够满足图遍历时的核间数据共享带宽需求.4)快速同步机制.支持基于数据流的核间细粒度快速同步,相比传统的基于内存的同步机制,性能可获得数量级的提升,有效缓解了每次迭代后的层同步压力.5)可编程数据通信机制.可编程数据传输引擎结构,可以快速实现数据的水平(片上处理器核之间)和垂直(从内存到片上存储)搬运,实现了数据通信的低延迟.高通量计算机有效解决了高通量典型应用BFS的缓存利用率低、内存访问效率低等问题.

4.2 优化手段评估

图3给出了顶点规模为$2^{24}\sim2^{30}$,平均度为16的Kronecker生成图在高通量计算机上执行BFS算法的性能变化趋势.在不做任何优化时,Graph500基准程序的平均性能只有0.69 GTEPS,生成图顶点规模对性能的影响较小.采用方向性优化后(Hybrid),性能得到明显提升,平均性能达到了6.52 GTEPS,性能提升趋势与Beamer采用四路Xeon E7-8870的结果(5.12 GTEPS)[9]基本一致.基于Yasui提出的去零点优化(remove zero, RZ)和度数感知优化(degree aware, DA)[13],我们提出了静态shuffle优化[12]来改善线程间的负载均衡性.如图3所示,优化前的最优性能为25.92 GTEPS,在顶点规模为2^{26}时取得.随着图顶点规模的增加,性能呈下降趋势.增加本文提出的位图访问优化和Bottom-up遍历优化之后,由于访存效率的改善,对于2^{26}的图,BFS

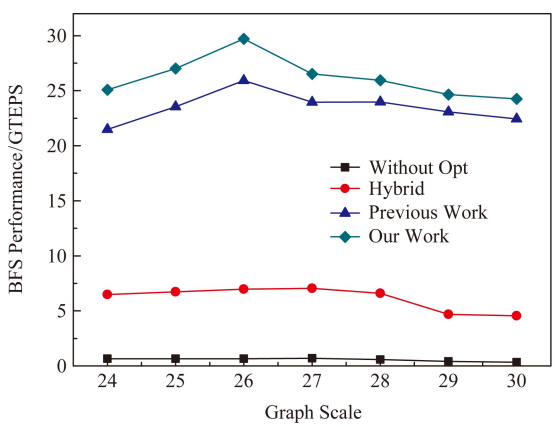

Fig. 3 BFS performance on HTC with different graph scale and *edgefactor*=16

图3 不同规模的图数据在平均度数为16的高通量计算机上的BFS性能

算法在高通量计算机上提升到了 29.71 GTEPS,且性能随图顶点规模的变化趋势与之前基本一致.相比 Graph500 的基准测试程序,实现了 43.01 倍的性能提升.对于所有 SCALE 的性能,性能平均提高 11.4%.充分说明了本文提出方法的有效性.表 5 对近年来基于 CPU 和高通量计算机的 BFS 优化工作进行了汇总.通过对比,进一步说明高通量计算机在图处理领域具有显著的优越性.

Table 5 Compare with Related Work

表 5 与相关工作对比

Machine	Opt-Method	2^{scale}	GTEPS
4-way Xeon E7-8870[9]	Hybrid	2^{28}	5.1
4-way Xeon E7-4640[13]	Hybrid+NUMA+ Degree aware	2^{27}	29.0
HTC	Hybid	2^{28}	6.52
HTC	All opt	2^{26}	29.73
HTC	All_opt	2^{30}	24.26

4.3 扩展性评估

为了研究优化后的 BFS 算法在高通量计算机上的扩展性,针对顶点规模为 2^{25} 的图数据,图 4 和表 6 给出了不同线程数目下的 BFS 性能变化.在平均度为 16 时,40 线程下的高并发处理比单线程的性能提升了将近 17.8 倍,同时性能随线程数目的增加呈近似的线性扩展.随后我们对平均度数的影响进行了研究.由图 4 可以看出,随着平均度数的增加,算法性能提升幅度明显,平均度数越高,性能越好.在平均度为 32 时,算法的平均性能可以达到 40.2 GTEPS,优于平均度为 16 时的 27.2 GTEPS 和平均度为 8 时的 13.14 GTPES.这主要是由 Kronecker 生成图的特性造成的.随着平均度的增加,Kronecker 生成图将会产生大量的重复边,降低了图数据的稀疏性.这些重复边在 BFS 的遍历过程中不会对运行时间造成影响,进而表现出更优的搜索效率.为了保证图拓扑结构的稀疏性,我们在后续的实验中选用 Graph500 中默认的 16 作为生成图的平均度.

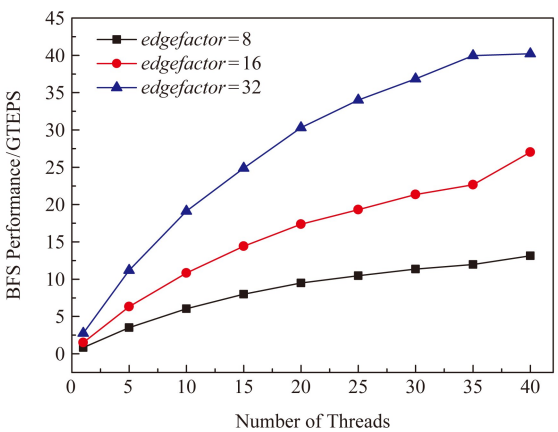

Fig. 4 Scalability of BFS Algorithm on HTC

图 4 BFS 算法在高通量计算机上的扩展性

Table 6 Scalability and Speed-up Under Different Numbers of Threads

表 6 不同线程数目下的扩展性和加速比

Thread	*edgefactor*=16		*edgefactor*=32	
	GTEPS	Speed-up	GTEPS	Speed-up
1	1.51	1	2.76	1
5	6.33	4.05	11.19	4.05
10	10.84	7.17	19.11	6.92
20	17.36	11.49	30.31	10.98
30	21.26	14.07	36.86	13.35
40	27.02	17.89	40.20	14.56

4.4 CPU 性能分析

本文提出的块查找位图访问优化、Bottom-up 遍历优化主要是对 L1 Data Cache 访存和分支跳转进行的优化,因此本节使用 Perf 对优化前后的 Bottom-up 函数进行了代码插装,获取了 L1 Data Cache(L1D)读访问次数、Branch 指令数的变化情况,进一步验证我们算法的有效性.如图 5 所示,L1D 读访问次数平均减少了 24.29 倍,因为块查找位图访问优化对 L1 Data Cache 按照 block 的粒度进行读取,将顶点数据读取到寄存器中,即使 block 中所有点都需要判断,也只需读取一次 cache,后续操作都是针对寄存器的运算.Branch 指令数平均减少了 39.34 倍,这是因为块查找位图算法复杂度低,

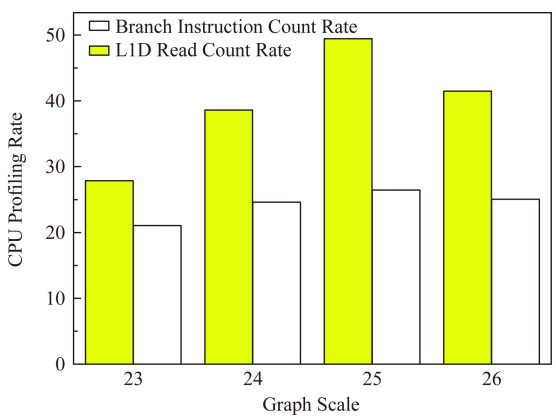

Fig. 5 CPU profiling parameters with and without the optimizations

图 5 优化前后的 CPU 参数对比

能够快速找到未判断的点,减少了分支指令次数.上述性能分析说明本文的优化手段有效改善了缓存局部性和分支效率,提高了访存效率.

4.5 性能对比

本节将对比分析 BFS 算法在高通量计算机和 x86 服务器上的性能,图 6 给出了优化后的 BFS 算法在高通量计算机和通用 x86 服务器下执行的性能对比,默认采用的优化手段为方向性优化、去零点优化、度数感知优化、静态 shuffle 优化以及本文提出的优化方法.如表 4 所示,我们使用的 x86 服务器拥有 2 个 NUMA 节点,当 CPU 访问非本地 NUMA 上的数据时,产生较长的访存时延.在文献[11]中,作者提出针对多 NUMA 架构的性能优化方法,保证线程在执行 BFS 遍历时只访问本地的 DRAM,避免了远程 DRAM 的访问开销.因此,在 x86 服务器上,我们增加了 NUMA 感知优化手段.由图 6 可以看出,BFS 算法在两路 x86 服务器上的整体性能与高通量计算机大致相仿,在顶点规模为 2^{26} 时,x86 服务器上的最优性能为 23.94 GTEPS.在绝对性能方面,高通量计算机展现了优越的表现.表 7 给出了不同图规模下高通量计算机相比 x86 服务器的性能加速比.可以看出,在大规模的图数据集下,高通量计算机的执行性能全面占优.在顶点规模为 2^{27} 时,最大性能加速比有 1.31 倍.

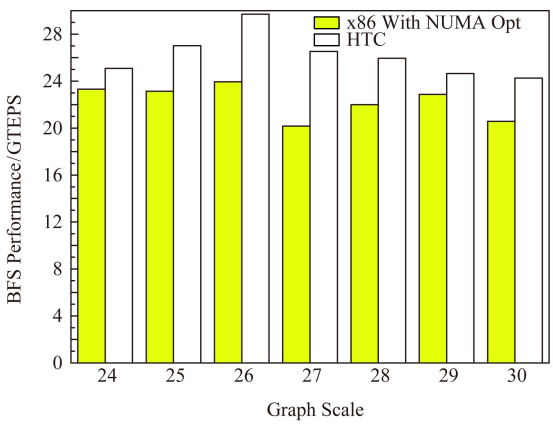

Fig. 6 Performance comparison of BFS between HTC and x86 Server

图 6 BFS 算法在高通量计算机与 x86 服务器的性能对比

Table 7 Performance Comparison Between HTC and x86 Server

表 7 高通量计算机相比 x86 服务器的性能加速比

2^{scale}	Speed-up
2^{24}	1.07
2^{25}	1.17
2^{26}	1.24
2^{27}	1.31
2^{28}	1.18
2^{29}	1.08
2^{30}	1.18

然而,多 NUMA 感知的优化方法增加了额外的编程复杂度,需要针对 Top-down 和 Bottom-up 过程分别进行不同的数据预处理,而且为避免线程跨 NUMA 节点访问远程数据,需要在每个 NUMA 节点的本地内存增加额外的数据结构,满足 Top-down 和 Bottom-up 的遍历需求,这同时也会引起内存膨胀.如图 7 所示,在使用 NUMA 优化后,内存的占用率提升了将近 1.3 倍.在图的顶点规模为 2^{30} 时,高通量计算机系统的内存使用接近 350 GB,而采用 NUMA 感知优化的 x86 服务器内存占用超过 420 GB.对于共享式内存系统,内存占用率是影响图数据处理规模的核心因素.综上,高通量计算机本身虽然无需采用 NUMA 优化,但是加入本文优化方法后在性能上可以超过额外增加 NUMA 优化的 x86 服务器,说明本文提出的方法能够发挥高通量计算机的性能,而且与传统计算机相比,高通量计算机占用的内存更少、编程复杂度也更小,高通量计算机在图数据处理方面具有显著的优势.

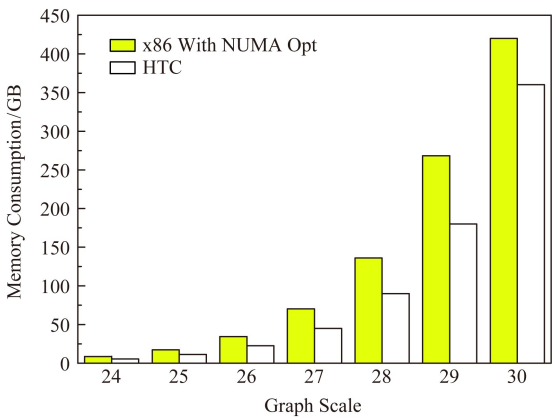

Fig. 7 Memory Consumption with and without NUMA optimization

图 7 NUMA 优化前后的内存使用

4.6 性能功耗比

本节将围绕能效方面展开研究.测试数据采用顶点规模为 2^{30}、平均度为 16 的 Kronecker 大图数据,并采用微型电力监测仪实时监测 2 套系统的功耗.实验结果如表 8 所示,未经优化的 Graph500 基准测试

程序的性能功耗比仅为 2.41 MTEPS/W.采用之前的优化方法优化后性能功耗比为 165.97 MTEPS/W,增加本文提出的优化方法 BSBB 后,高通量计算机的整机功耗为 134 W,性能功耗比为 181.04 MTEPS/W,性能功耗比显著提高.x86 服务器的整机功耗为 286 W,性能功耗比为 71.92 MTEPS/W.面向超大规模的图数据集,高通量计算机相比 x86 服务器在单节点上拥有 2.51 倍的性能功耗比优势.高通量计算机的性能功耗比结果在 2019 年 6 月发布的 Green Graph500 面向大数据集的排行榜上可以排名第 2 位[17].综上,高通量计算机的高并发和低功耗等特点非常适合处理大规模图计算等数据密集型应用.

Table 8 Energy Efficiency of BFS Implementation Between HTC and x86 Server

表 8 高通量计算机与 x86 服务器的性能功耗对比 ($2^{scale}=2^{30}$)

Machine	Opt-Method	GTEPS	Watt	MTEPS/W
HTC	Unopt	0.33	137	2.41
HTC	Pre	22.24	134	165.97
HTC	Pre+BSBB	24.26	134	181.04
x86	Pre+BSBB	20.57	286	71.92

5 总 结

本文对单节点的并行 BFS 算法和 BFS 算法的优化技术进行了介绍,并在此基础上提出 2 种面向高通量计算机的优化手段块查找位图优化和 Bottom-up 遍历优化,提高 BFS 的缓存局部性和访存效率.基于上述的优化手段,我们对 BFS 算法在高通量计算机上的性能进行了系统的评估.对于顶点规模为 2^{30} 的大图,在性能方面,高通量计算机在单节点的平均性能为 24.26 GTEPS;在性能功耗比方面,高通量计算机的结果为 181.04 MTEPS/W,在 2019 年 6 月发布的 Green Graph500 面向大数据集的排行榜上可以排名第 2 位[17].与通用的 x86 服务器相比,高通量计算机在单节点具有 1.18 倍的性能优势以及 2.51 倍的性能功耗比优势.由于高通量计算机的高并发、强实时和低功耗等特点,在处理大规模图数据时,结合应用特征采用新型的高通量计算机系统,可获得更高的处理能效.

参 考 文 献

[1] Barabási A L, Albert R. Emergence of scaling in random networks [J]. Science, 1999, 286(5439): 509-512

[2] Haveliwala T H. Topic-sensitive pagerank: A context-sensitive ranking algorithm for Web search [J]. IEEE Transactions on Knowledge and Data Engineering, 2003, 15(4): 784-796

[3] Mislove A, Marcon M, Gummadi K P, et al. Measurement and analysis of online social networks [C] //Proc of the 7th ACM SIGCOMMConf on Internet Measurement. New York: ACM, 2007: 29-42

[4] Bader D A, 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 2008 IEEE Int Symp on Parallel and Distributed Processing. Piscataway, NJ: IEEE, 2008: 1-12

[5] Murphy R C, Wheeler K B, Barrett B W, et al. Introducing the graph 500 [J]. Cray Users Group, 2010, 19: 45-74

[6] Sciencenet. King Kong, the first high-throughput concept computer released [OL]. [2018-10-26]. http://news.sciencenet.cn/htmlnews/2018/10/419158.shtm (in Chinese)
(科学网. 首款高通量概念计算机"金刚"发布[OL]. [2018-10-26]. http://news. sciencenet. cn/htmlnews/2018/10/419158.shtm)

[7] Agarwal V, Petrini F, Pasetto D, et al. Scalable graph exploration on multicore processors [C] //Proc of the 2010 ACM/IEEE Int Conf for High Performance Computing, Networking, Storage and Analysis(SC'10). Piscataway, NJ: IEEE, 2010: 1-11

[8] Hong S, Oguntebi T, Olukotun K. Efficient parallel graph exploration on multi-core CPU and GPU [C] // Proc of 2011 Int Conf on Parallel Architectures and Compilation Techniques. Piscataway, NJ: IEEE, 2011: 78-88

[9] Beamer S, Asanović K, Patterson D. Direction-optimizing breadth-first search [J]. Scientific Programming, 2013, 21(3/4): 137-148

[10] Ye Nan, Hao Ziyu, Zheng Fang, et al. Adaptability of BFS algorithm and many-core processor [J]. Journal of Computer Research and Development, 2015, 52(5): 1187-1197 (in Chinese)
(叶楠, 郝子宇, 郑方, 等. BFS 算法与众核处理器的适应性研究[J]. 计算机研究与发展, 2015, 52(5): 1187-1197)

[11] Yasui Y, Fujisawa K, Goto K. NUMA-optimized parallel breadth-first search on multicore single-node system [C] // Proc of 2013 IEEE Int Conf on Big Data. Piscataway, NJ: IEEE, 2013: 394-402

[12] Zhang Chenglong, Cao Huawei, Ye Xiaochun, et al. Highlyefficient breadth-first search on CPU-based single-node system [C] // Proc of 2019 IEEE 21st Int Conf on High Performance Computing and Communications. Piscataway, NJ: IEEE, 2019: 2066-2071

[13] Yasui Y, Fujisawa K, Sato Y. Fast and energy-efficient breadth-first search on a single NUMA system [C] // Proc of Int Supercomputing Conf. Berlin: Springer, 2014: 365-381

[14] Yasui Y, Fujisawa K. Fast and scalable NUMA-based thread parallel breadth-first search [C] // Proc of 2015 Int Conf on High Performance Computing & Simulation (HPCS). Piscataway, NJ: IEEE, 2015: 377-385

[15] Gonzalez J E, Low Y, Gu H, et al. Powergraph: Distributedgraph-parallel computation on natural graphs [C] //Proc of the 10th USENIX Conf on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2012: 17-30

[16] Fan Dongrui, Ye Xiaochun, Bao Yungang, et al. The road to independent research and development of high-throughput computers in China [J]. Bulletin of Chinese Academy of Sciences, 2019, 34(6): 648-656 (in Chinese)
(范东睿, 叶笑春, 包云岗, 等. 中国高通量计算机的自主研发之路[J]. 中国科学院院刊, 2019, 34(6): 648-656)

[17] Graph500. JUNE 19 Green Graph500: BIG DATA [OL]. [2020-04-10]. http://graph500.org/?page_id=724

Zhang Chenglong, born in 1991. PhD candidate. Student member of CCF. His main research interests include high throughput computing and parallel computing.

Cao Huawei, born in 1989. PhD. Member of CCF. His main research interests include parallel computing and high throughput computing architecture.

Wang Guobo, born in 1993. Master. His main research interests include high throughput computing architecture and parallel computing.

Hao Qinfen, born in 1969. PhD. Professor. Member of CCF. His main research interests include optic enabled computer architecture, microarchitecture, graph computing.

Zhang Yang, born in 1981. PhD. Engineer. Member of CCF. His main research interests include high throughput computing, parallel computing, and simulation of computer architecture.

Ye Xiaochun, born in 1981. PhD. Associate professor. Member of CCF. His main research interests include algorithm paralleling and optimizing software simulation, and architecture for high throughput computer.

Fan Dongrui, born in 1979. PhD. Professor, PhD supervisor. Senior member of CCF. His main research interests include many-core processor design, high throughput processor design, and low power micro-architecture.