文章编号:1007－6735(2012)06－0527－04

基于多核心工作站机群的并行介数算法

毛国勇[1]， 张 宁[2]

(1. 常州工学院 电子信息与电气工程学院,常州 213002; 2. 上海理工大学 管理学院,上海 200093)

摘要:针对计算大规模复杂网络时介数的空间和时间复杂度问题,根据网络数据的存储特点,设计了减少内存占用并能提高查找速度的数据结构.根据介数计算的特点,用 Python 语言设计了粗粒度并行算法,在多核心工作站机群实现了并行算法.实验结果表明:并行算法不仅能够适用于上亿条边规模的网络,而且能够获得线性加速比,使 120 个计算核心的加速比达到了 71 左右,为分析大规模复杂网络数据的特性提供了易操作的方案.

关键词:大规模; 复杂网络; 介数; 并行计算

中图分类号: TP 338.6 **文献标志码**: A

Parallel Algorithm of Betweenness Centrality in Multicore Cluster of Workstations

MAO Guo-yong[1], ZHANG Ning[2]

(1. *Department of Electronic Information and Electric Engineering, Changzhou Institute of Technology, Changzhou 213002, China*; 2. *Business School, Shanghai University of Science and Technology, Shanghai 200083, China*)

Abstract: Focusing on the space and time complexity problems in computing betweenness centrality in large complex network, the data structure that can reduce the memory consumption and increase execution speed was brought forward based on the storage property of network data. A coarse grain parallel algorithm was designed using Python language according to the features of computing betweenness centrality. The parallel algorithm was realized in the cluster of multi-core workstations. The test results indicate that the algorithm can be applied in the analysis of large network with hundred millions of edges, and has linear speedup. The speedup can reach 71 when 120 cores are used, so as to provide an operational method for the analysis of large scale complex network data.

Key words: *large scale*; *complex network*; *betweenness centrality*; *parallel computing*

大规模网络分析在许多重要领域,如社会网络、信息传播网络和生物网络等领域中有着广泛的应用[1],而介数[2](*BC*)是分析大规模复杂网络时广泛使用的一个指标.通过这个指标,可以对图中节点的重要程度进行定量分析,可以衡量一个节点在网络通信中的重要性,识别出网络中的关键节点.较高的

收稿日期:2012－09－19

基金项目:国家自然科学基金资助项目(70971089);江苏省高校优秀中青年教师和校长境外研究计划资助项目;常州工学院自然科学基金资助项目(YN1004)

作者简介:毛国勇(1971－),男,副教授.研究方向:并行计算、图论算法.E-mail:maogy@czu.cn

BC 指标表明，一个节点到其它节点有相对较多的最短路径，或者这个节点位于其它两个节点最短路径上的可能性很大. *BC* 反映了相应的节点在整个网络中的作用和影响力，是一个重要的全局几何量，具有很强的现实意义. 例如，在社会关系网或技术网络中，介数的分布特征反映了不同人员、资源和技术在相应生产关系中的地位，这对于发现和保护关键资源、技术和人才具有重要意义. *BC* 的计算是网络分析的重要内容，它占了大部分的计算时间. 所以，对这个算法进行并行加速具有重要意义.

1 *BC* 算法简介

节点 v 的 *BC* 为

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{1}$$

式中，σ_{st} 为从节点 s 到节点 t 的最短路径总数目；$\sigma_{st}(v)$ 为这些路径中经过顶点 v 的路径条数. 计算一个图中所有顶点的 *BC* 需要计算所有顶点间的最短路径，所需时间为 $O(n^3)$. 对于无权图，用 Brandes 算法计算所需时间为 $O(n \times m)$[3]，其中，n 为顶点的数目，m 为边的数目.

Brandes 算法由宽度优先搜索(breadth first search，BFS)、部分和累加、标准化等3个部分构成. 对于图1所示的 *BC* 算法示例图中的 b 点，其 *BC* 值为

$$BC(b) = (\sigma_{ac}(b)/\sigma_{ac} + \sigma_{ad}(b)/\sigma_{ad} + \sigma_{ae}(b)/\sigma_{ae} + \sigma_{cd}(b)/\sigma_{cd} + \sigma_{ce}(b)/\sigma_{ce} + \sigma_{de}(b)/\sigma_{de})/6 = (1/1 + 1/1 + 2/2 + 1/2 + 0 + 0)/6 = 3.5/6 \approx 0.583$$

式中，$\sigma_{ac}(b)$ 为经过 b 点从 a 到 c 的最短路径条数，σ_{ac} 为从 a 到 c 的最短路径条数. 假定0除0等于0. 为了使最终的 *BC* 位于[0,1]之间，还需要对结果进行标准化. 所谓标准化对于无向图而言，就是除以 $(n-1)\times(n-2)/2$. 对于有向图，就是除以 $(n-1)\times(n-2)$. 图1为无向图，$n=5$，$(n-1)\times(n-2)/2=6$.

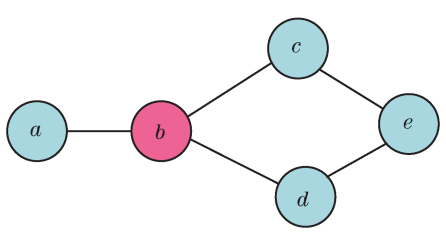

图1 *BC* 算法示例图

Fig.1 Example graph of *BC* algorithm

2 并行 *BC* 算法

2.1 算法的数据结构

求解 *BC* 时，复杂网络的规模也是计算中必需面对的问题. 在空间复杂度方面，目前复杂网络主要有邻接表和邻接矩阵两种表示方法. 如果用邻接矩阵来表示复杂网络，其空间复杂度为 V^2，其中 V 是顶点的数目；对于节点数为1 000 000的复杂网络，计算机需要数T的内存才能表示，这对于目前的计算机几乎是不可能的. 对于稀疏网络而言，用邻接表表示则占用较少的内存空间，因为它不需要存储不存在的边. 如果使用数组来表示邻接表，对于无向图需要 $8E$ 字节的存储容量，其中 E 是边的数目，每条边使用4字节，每一条边在邻接表中出现两次；而对于有向图，只需要 $4E$ 字节的存储容量，因为每条边在邻接表中只出现一次. 由于复杂网络往往是稀疏网络，因此，采用邻接表来存储网络.

本算法的开发语言为 Python 2.7[4]，它是一种面向对象、直译式的计算机程序设计语言. 其提供了 Dict(字典)、Set(集合)、List(列表)及 Tuple(元组)等几种数据结构，其中，字典有时称为关联数组或者哈希表. 它们通过键将一系列值联系起来，这样就可以使用键从字典中取出一项. 本算法中采用字典来表示邻接表，将键值相同的值放在一个列表中，实现从键到值的多重映射. 例如，对于以边表形式表示的图数据

1	2
1	3
1	4
2	4
2	5

其对应的字典为{1:(2,3,4);2:(4,5)}，顶点1和2为键，(2,3,4)及(4,5)分别为与键1和键2对应的值. 由于 Python 对内存的使用非常节省，加上用整型的顶点编号来代替字符型的顶点名称，因此，对于325 729个顶点及1 497 134条边的网络[5]，其内存占用不到200 M. 对于2 508 811个顶点及25 278 346条边的网络[6]，其内存占用不到2G，因此，采用这种数据结构，能在目前流行的服务器上对上亿条边的复杂网络进行分析.

2.2 算法流程

本并行算法的开发环境为并行 Python[7]，它是一个 Python 模块，能够在 SMP 及工作站机群 上并

行运行 Python 代码. *BC* 的并行算法从每个源节点开始做宽度优先遍历,然后回溯累加得到 *BC*. 采用并行 Python 来实现粗粒度并行,将每个从源节点开始的遍历及回溯累加过程看作一个任务,算法需要 n 个任务,求得每个节点的部分 *BC* 值. 最后,将每个任务的部分 *BC* 值累加,得到每个顶点的最终 *BC* 值,这 n 个任务可以并行处理,而且在并行计算过程中,各个节点间不需要通信,只需要一个并行规约操作得到最终的值. 算法流程如图 2 所示.

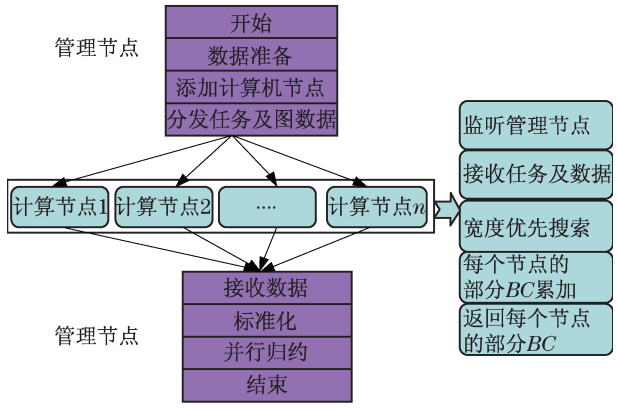

图 2　并行算法在管理节点及计算节点的执行流程

Fig. 2　Execution workflow of parallel algorithm in management and computation of nodes

在每个计算节点,都需要以源点 s 表示图的字典 a Dict,以及图的顶点集合 G 作为参数调用宽度优先遍历子程序,返回该源点对应的部分 *BC* 值,从该源点开始遍历时可以访问的顶点集合 S,以及开始遍历时,节点 v 的前驱集合 Pred,则对应的算法为

```
def single_source_shortest_path(G, a Dict, s):
1     S = []
2     Pred = {}
3     for v in G:
4     Pred[v] = []
5     sigma = dict.fromkeys(G,0.0)
      #对所有顶点的 sigma 置 0
6     D = {} //用来表示从源点开始到某一个顶点的路径条数
7     sigma[s] = 1.0
8     D[s] = 0
9     Q = [s]// Q 用来存放当前点的邻接节点
10    while Q:  #用宽度优先搜索来查找最短路径
11        v = Q.pop(0)
12        S.append(v)
13        Dv = D[v]
14        sigma v = sigma[v]
15        if v in a Dict:
16        for w in a Dict[v]:
17        if w not in D:
18        Q.append(w)
19        D[w] = Dv + 1
20        if D[w] = = Dv + 1:
          #找到了一条最短路径
21        sigma[w] + = sigma v
22        Pred[w].append(v)   #将 v 加入到前驱集合
23        return S,Pred,sigma
```

由于每个节点部分 *BC* 累加,规范化以及并行规约的算法比较简单,限于篇幅,这里不再表述.

2.3　性能评价

算法中,第 15 行用来从字典中查找顶点 v 是否存在于字典的键中,第 16 行用来在键中查找 v,并返回与 v 对应的值,第 17 行用来判断 w 是否在字典 D 中,这 3 步是 while 循环中最耗时的部分. 但由于 Python 字典采用哈希表实现,因此,能快速地检查一个键在字典中是否存在,查找效率可以达到 $O(1)$. 需要指出的是,本算法只是实现了粗粒度的并行,因此并行效率不如考虑了图的邻接信息的中粒度及细粒度并行[8],但正因为不用考虑图的细节,因此实现起来相对简单,能够方便地应用到更多核心、更多工作站的机群.

3　实验结果

测试用的服务器包括一台用作管理节点的 Dell PowerEdge R910 工作站,它有 32 个至强 L7555 物理核心,128G 内存,另有 10 台用作计算节点的 Dell PowerEdge R810 工作站,它们都有 12 个至强 X5660 物理核心,24G 内存. 测试的软件环境为 CentOS5、intel icc 编译器 11.1、Python2.7 以及 parallel python1.6.1,Python 使用 icc 编译器编译以提高 Python 程序的执行速度.

本文测试了文献[5]的数据,它含有 325 729 个点,1 497 134 条边. 不同处理器数目 k 的所需计算时间 t 及加速比 r 如表 1 所示(见下页),相应的加速比曲线如图 3 所示(见下页).

表1 处理器数目与计算时间及加速比对应关系表

Tab.1 Relationship between number of processors, computing time and speedup

并行性能	k						
	1	12	24	48	72	96	120
t/s	1 263 720	114 426	59 549	33 858	25 262	20 779	17 798
r	1	11	21.2	37.3	50	60.8	71

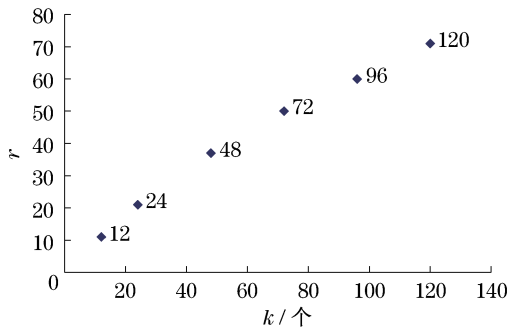

图3 加速比曲线

Fig.3 Curve of speedup

从表1可以看出，串行算法计算 *BC* 需1 263 720 s，约14 d，如果在10台工作站共120个处理器中进行计算，所需时间只有17 798 s，不到5 h.从图3可以看出，随着处理器数目的增加，加速比也有线性上升的趋势.

4 结论及展望

在多核心工作站机群实现了计算介数的并行算法，为解决介数计算的时间与空间复杂度问题提供了易操作的方案，为分析更大规模复杂网络的特性打下了良好的基础.

未来的工作将探讨并行计算时的负载平衡问题，减少并行规约前的等待时间，提高计算速度.

参考文献：

[1] Wang X F, Chen G R. Complex networks: small-world, scale-free and beyond [J]. IEEE Circuits and Systems Magazine, 2003, 3(1): 6 - 20.

[2] Freeman L C. A set of measures of centrality based on betweenness[J]. Sociomtry, 1977, 40(1): 35 - 41.

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

[4] Official website for pathon. python2. 7. python [EB/OL]. [2012 - 09 - 29]. http: // www. python. offical website for python. org/index. html.

[5] Albert R, Jeong H, Barabási A L. Internet: diameter of theworld wide web[J]. Nature, 1999, 401(6749): 130 - 131.

[6] 苏磊，张宁，马良. 一种新的大规模网络最短路径的近似算法[J]. 复杂系统与复杂性科学，2008，5(2)：51 - 54.

[7] Offical website for parallel python. Parallel python1. 6. 1. parallel python[EB/OL]. [2012 - 09 - 30]. http: // www. parallelpython. com/index. html.

[8] 涂登彪，谭光明，孙凝晖. 无锁同步的细粒度并行介度中心算法[J]. 软件学报，2011，22(5)：986 - 995.