J. Parallel Distrib. Comput. 76 (2015) 106–119

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Regularizing graph centrality computations

Ahmet Erdem Sarıyüce [a,b,*], Erik Saule [d], Kamer Kaya [a], Ümit V. Çatalyürek [a,c]

[a] Department of Biomedical Informatics, The Ohio State University, United States
[b] Department of Computer Science and Engineering, The Ohio State University, United States
[c] Department of Electrical and Computer Engineering, The Ohio State University, United States
[d] Department of Computer Science, University of North Carolina at Charlotte, United States

HIGHLIGHTS

- We propose parallel algorithms to compute centrality on accelerators.
- We apply multiple breadth-first search operations simultaneously.
- Vectorization is applied to make the closeness computation faster.
- All the algorithms and techniques are experimentally validated.
- We get better performance than the best existing centrality computation solutions.

ARTICLE INFO

Article history:
Received 27 March 2014
Received in revised form
27 July 2014
Accepted 29 July 2014
Available online 7 August 2014

Keywords:
Betweenness centrality
Closeness centrality
BFS
CPU
GPU
Intel Xeon Phi
Vectorization

ABSTRACT

Centrality metrics such as betweenness and closeness have been used to identify important nodes in a network. However, it takes days to months on a high-end workstation to compute the centrality of today's networks. The main reasons are the size and the irregular structure of these networks. While today's computing units excel at processing dense and regular data, their performance is questionable when the data is sparse. In this work, we show how centrality computations can be regularized to reach higher performance. For betweenness centrality, we deviate from the traditional fine-grain approach by allowing a GPU to execute multiple BFSs at the same time. Furthermore, we exploit hardware and software vectorization to compute closeness centrality values on CPUs, GPUs and Intel Xeon Phi. Experiments show that only by reengineering the algorithms and without using additional hardware, the proposed techniques can speed up the centrality computations significantly: an improvement of a factor 5.9 on CPU architectures, 70.4 on GPU architectures and 21.0 on Intel Xeon Phi.

1. Introduction

The centrality metrics play an important role in network and graph analysis since they are related with several concepts such as reachability, importance, influence, and power [31,12,18,23,29]. Betweenness and closeness (BC and CC) are two such metrics. However, the complexity of the best algorithms to compute them is unbearable for today's large-scale networks: for unweighted networks, it is $\mathcal{O}(nm)$ where n is the number of vertices and m is the number of edges in the corresponding graph [5]. For weighted networks, the complexity is more, $\mathcal{O}(nm + n^2 \log n)$. Although this already makes the problem hard even for medium-scale graphs, considering million- and even billion-scale ones, it is clear that we need efficient high performance computing (HPC) techniques.

There are several GPU-based algorithms and parallelization techniques for computing betweenness [11,24,29,22] and closeness [11,29] centrality. However, as we will show in this paper, since these techniques process only a single graph traversal at a time and employ pure fine-grain parallelism, they cannot fully utilize the GPU and reach the device's peak performance. In addition to these studies, parallel breadth-first search (BFS), which is the main building block to compute closeness centrality values, has been widely studied on shared-memory systems such as GPUs [10,15,19] and Intel Xeon Phi [27]. Since these works focus on the parallelization of a single BFS, their natural extension to CC will yield the iterative execution of a fine-grain parallel CC kernel responsible from a single graph traversal. In this work, we propose

* Correspondence to: 250 Lincoln Tower, 1800 Cannon Drive, Columbus, OH 43210, United States.
E-mail addresses: sariyuce.1@osu.edu (A.E. Sarıyüce), esaule@uncc.edu (E. Saule), kamer@bmi.osu.edu (K. Kaya), umit@bmi.osu.edu (Ü.V. Çatalyürek).

http://dx.doi.org/10.1016/j.jpdc.2014.07.006
0743-7315/

novel and efficient algorithms and techniques to compute betweenness centrality on GPU and closeness centrality on GPU and Intel Xeon Phi. Although we agree that fine-grain parallelism is still necessary due to the memory restriction of the cutting-edge many-core architectures at hand, we leverage the potential of the hardware by enabling a hybrid coarse/fine-grain parallelism technique that executes multiple simultaneous BFSs.

Although many of the existing techniques leverage parallel processing, one of the most common parallelism available in almost all of today's recent processors, namely instruction parallelism via vectorization, is often overlooked due to nature of the sparse graph kernels. Graph computations are notorious for having irregular memory access pattern, and hence for many kernels that require a single graph traversal, the available vectorization support, which is a great arsenal to increase the performance, is usually considered not very effective. It can still be used, for a small benefit, at the expense of some preprocessing that involves partitioning, ordering and/or use of alternative data structures. To exploit its full potential and enable it for simultaneous graph traversal approach, we provide an ad-hoc CC formulation based on bitwise operations and propose hardware and software vectorization for that formulation on cutting-edge hardware. Our approach for closeness centrality serves as an example to show how vectorization can be utilized for graph kernels that require multiple BFS traversals. As a result, we experimentally show that compared to the existing solutions, the proposed techniques can be significantly faster while computing exact betweenness and closeness centrality values, on the same device, i.e., without using an additional hardware resource. Furthermore, the proposed techniques can also be used to compute approximate BC and CC values for which the graph traversals are only initiated from a subset of vertices.

The rest of the paper is organized as follows: Section 2 presents the background information including the notation we used in the paper, basic sequential algorithms, and a summary of the existing parallelization approaches including accelerator-based algorithms for betweenness and closeness centrality. The proposed parallelization algorithms and techniques are explained in Section 3 and their performance is evaluated in Section 4. Section 5 concludes the paper.

2. Notation and background

Let $G = (V, E)$ be a simple undirected, unweighted graph modeling a network where each node is represented by a vertex in V, and an interaction between two nodes is represented by a single edge in E. Let n be the number of vertices, m be the number of edges, and $adj(v)$ be the set of vertices interacting with v.

A *path* is a sequence of vertices such that there exists an edge between consecutive vertices. If there is a path from $u \in V$ to $v \in V$, and hence from v to u, we say that u and v are connected. The shortest path distance between these vertices is denoted by $dst(u, v)$. If $u = v$ then $dst(u, v) = 0$. The graph G is *connected* if all vertex pairs are connected. Otherwise, G is *disconnected*. A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. Each maximal connected subgraph of G is a *connected component*, or simply a *component*, of G.

2.1. Betweenness centrality

Let $G = (V, E)$ be a connected graph. Let σ_{st} be the number of shortest paths from a source $s \in V$ to a target $t \in V$, and $\sigma_{st}(v)$ be the number of such s-t paths passing through a vertex $v \in V$, $v \neq s, t$. Let $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$, the fraction of the shortest s-t paths passing through v among all shortest s-t paths. The betweenness centrality of v is defined by

$$\texttt{bcent}[v] = \sum_{s \neq v \neq t \in V} \delta_{st}(v). \tag{1}$$

To compute $\texttt{bcent}[v]$ for all $v \in V$, Brandes proposed an algorithm that is based on the accumulation of pair dependencies over target vertices [5]. After accumulation, the dependency of v to $s \in V$ is

$$\delta_s(v) = \sum_{t \in V} \delta_{st}(v). \tag{2}$$

Let $\texttt{pred}_s(u)$ be the set of u's predecessors on the shortest paths from s to all vertices in V. That is,

$$\texttt{pred}_s(u) = \{v \in V : (v, u) \in E, dst(s, u) = dst(s, v) + 1\}.$$

Hence, $\texttt{pred}_s$ defines the *shortest path graph* rooted in s. Brandes observed that the accumulated dependency values can be computed recursively:

$$\delta_s(v) = \sum_{u : v \in \texttt{pred}_s(u)} \frac{\sigma_{sv}}{\sigma_{su}} \times (1 + \delta_s(u)). \tag{3}$$

Brandes' algorithm, which is given in Algorithm 1, computes $\delta_s(v)$ for all $v \in V \setminus \{s\}$ by using a two-phase approach: First, a breadth first search (BFS) is initiated from s to compute σ_{sv} and $\texttt{pred}_s(v)$ for each v: in this *forward phase*, the algorithm computes $\sigma[v]$ for $v \in V$ which is the number of shortest paths from the source vertex s to v. In addition, the predecessors of v on these shortest paths are stored in $\texttt{pred}[v]$. Then, in a *backward phase*, $\delta_s(v)$ is computed for all $v \in V$ in a bottom-up manner by using (3).

For undirected graphs, each phase of Algorithm 1 processes all the edges at most once, taking $\mathcal{O}(m + n)$ time. The phases are repeated for each source vertex. The overall complexity of SeqBC is $\mathcal{O}(mn)$. Currently, it is asymptotically the fastest known sequential algorithm to compute BC.

Algorithm 1: SeqBC($G = (V, E)$)

```
1  for all v ∈ V do
2      bcent[v] ← 0
3  for each s ∈ V do
4      stack ← ∅, queue ← ∅
5      queue.push(s), dst[s] ← 0, σ[s] ← 1
6      for all v ∈ V \ {s} do
7          dst[v] ← ∞, pred[v] ← ∅, σ[v] ← 0
       ▷Forward Phase
8      while queue is not empty do
9          v ← queue.pop(), stack.push(v)
10         for all w ∈ adj(v) do
11             if dst[w] < 0 then
12                 dst[w] ← dst[v] + 1
13                 queue.push(w)
14             if dst[w] = dst[v] + 1 then
15                 σ[w] ← σ[w] + σ[v]
16                 pred[w].push(v)
       ▷Backward Phase
17     for all v ∈ V do
18         δ[v] ← 0
19     while stack is not empty do
20         w ← stack.pop()
21         for v ∈ pred[w] do
22             δ[v] ← δ[v] + (σ[v]/σ[w])(1 + δ[w])
23         if w ≠ s then
24             bcent[w] ← bcent[w] + δ[w]
25 return bcent
```