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.

© 2014 Elsevier Inc. All rights reserved.

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/© 2014 Elsevier Inc. All rights reserved.

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
```

2.2. Closeness centrality

Given a graph G, the closeness centrality of u can be defined as

$$\texttt{ccent}[u] = \sum_{\substack{v \in V: \\ dst_G(u,v) \neq \infty}} \frac{1}{dst_G(u, v)}. \quad (4)$$

If u cannot reach any vertex in the graph $\texttt{ccent}[u] = 0$. Please note that, in the literature, an alternative of this formulation exists where summation is in the denominator (instead of in front of the fraction). The proposed techniques would work for both formulation. But for the sake of simplicity, we use the one above. Nevertheless, both require the shortest path distances between all vertex pairs.

Algorithm 2: SeqCC($G = (V, E)$)

```
1  for each u ∈ V do
2  |  ccent[u] ← 0
3  for each s ∈ V do
4  |  queue ← ∅
5  |  for all v ∈ V \ {s} do
6  |  |  dst[v] ← ∞
7  |  queue.push(s), dst[s] ← 0
8  |  while queue is not empty do
9  |  |  v ← queue.pop()
10 |  |  for all w ∈ adj(v) do
11 |  |  |  if dst[w] = ∞ then
12 |  |  |  |  dst[w] ← dst[v] + 1
13 |  |  |  |  queue.push(w)
14 |  |  |  |  ccent[s] ← ccent[s] + 1/dst[w]
15 return ccent
```

Algorithm 2, SeqCC, computes the closeness centrality values in G. For each vertex $s \in V$, the algorithm initiates a breadth-first search (BFS) from s, computes the distances to the other vertices, and accumulates to $\texttt{ccent}[s]$. (Notice that, for undirected graphs, as the ones discussed here, one could also accumulate to $\texttt{ccent}[w]$ instead of $\texttt{ccent}[s]$, since $dst_G(s, w) = dst_G(w, s)$.) Since a BFS takes $\mathcal{O}(m + n)$ time, and n BFSs are required in total, the complexity follows.

SeqCC visits the vertices in a *top-down* manner, i.e., the vertices with distance ℓ are processed to visit distance $\ell + 1$ vertices. Another way to do the same is processing the adjacency lists of the unvisited vertices and set their distance to $\ell + 1$ if they have a neighbor at level ℓ. This *bottom-up* variant is clearly expensive at first but it can be much cheaper for large ℓ values since there are much less unvisited vertices remaining. Beamer et al. used this observation to obtain a significantly faster *direction-optimized* BFS implementation by using the cheaper version at each BFS level [3]. In this work, we do the same within the CPU closeness centrality implementation that we use as one of the baselines in our experiments.

2.3. Parallelism for network centrality

The centrality computations can be parallelized in two ways: coarse- and fine-grain. In coarse-grain parallelism, the BFSs are shared among the threads, i.e., a shortest-path graph is constructed by a single thread. Hence, the threads need to work with separate memory regions in SeqCC and SeqBC, e.g., σ, δ, $\texttt{pred}$, *queue*, *stack*, and $\texttt{d}$. In fine-grain parallelism, a BFS is concurrently executed by multiple threads. Ligra [30] and SNAP [2] are two state-of-the-art shared-memory graph processing frameworks, both make use of fine-grain parallelism for BC computation and will serve as baseline for our BC parallelization techniques. In fine-grain parallelism, although the memory footprint is less, compared to the coarse-grain parallelism, concurrency can bring a significant overhead due to the necessity of (relatively) expensive tools such as atomic operations and conflict resolution. That being said, for devices with restricted memory and large potential for concurrent execution, such as GPUs, a fine-grain parallelism is usually necessary.

There are existing studies on computing closeness and betweenness centrality using GPUs; Shi and Zhang developed a software package to do biological network analysis [29]. Later, various parallelism techniques on GPUs for BC and CC computations are experimented by Jia et al. [11]. Concurrently, Pande and Bader studied computing BC of small-world networks on a GPU [22]. Recently, Sarıyüce et al. used a modified graph storage scheme to obtain better speedups compared to existing solutions [24]. Apart from the centrality computation, Merrill et al. [19] propose different fine-grain parallelization techniques for BFS computation, which is a building block for BC and CC computations, and prove to be the fastest solution on GPU architectures. All these studies employ a pure fine-grain parallelism and *level-synchronized* BFSs. That is, while traversing the graph, the algorithms initiate a GPU kernel for each level ℓ to visit the vertices/edges on that level and find the vertices on level $\ell + 1$. One interesting work which combines fine-grain and coarse-grain parallelism is [20]. Although we use the same combination, in [20], a queue-based implementation is employed and hence the number of simultaneous BFSs is limited due to the contention in the queue. Our work alleviates this problem by not employing a queue.

Another recent work on betweenness centrality computation investigates the edge and node parallelism on dynamic betweenness centrality computation [17]. In a preliminary version of this paper, we introduced vectorization support for efficient closeness centrality computation [25]. Other than that, to the best of our knowledge, there is no prior work on computing centrality using hardware and/or software vectorization.

In an earlier work, we had presented an early evaluation of the scalability of several variants of BFS algorithm on Intel Xeon Phi coprocessor using a pre-production card in which we had presented a re-engineered shared queue data structure for many-core architectures [27]. In another study, we had also investigated the performance of SpMV on Intel Xeon Phi coprocessor architecture, and show that memory latency, not memory bandwidth, creates a bottleneck for SpMV on Intel Xeon Phi [28].

A similar problem to centrality computation is all-pairs shortest path computation. There are several studies on GPU-based parallelization of this problem [13,16,21]. A shared memory cache efficient GPU implementation to solve transitive closure and the all-pairs shortest-path problem on directed graphs for large datasets is proposed in [13]. Their solution is able to handle graph sizes that are larger than the DRAM memory of available on the GPU. Matsumoto et al. [16] proposed a blocked algorithm for all-pairs shortest path problem to be used in a hybrid CPU–GPU system where the communication between CPU and GPU is minimized. In [21], authors present an algorithm to accelerate the all-pairs shortest path computation on GPUs by solving multiple single source shortest path problems at a time, allowing to efficiently access graph data by sharing the data between processing elements in the GPU. In our work, we focus on faster GPU parallelization of betweenness and closeness centrality computation on unweighted graph which have more regularity than the all-pairs shortest path computation allowing finer synchronization and optimization.

2.3.1. Graph storage schemes and parallelization

Many sparse matrix and graph algorithms such as sparse matrix–vector multiplication (SpMV) and BFS are known to be

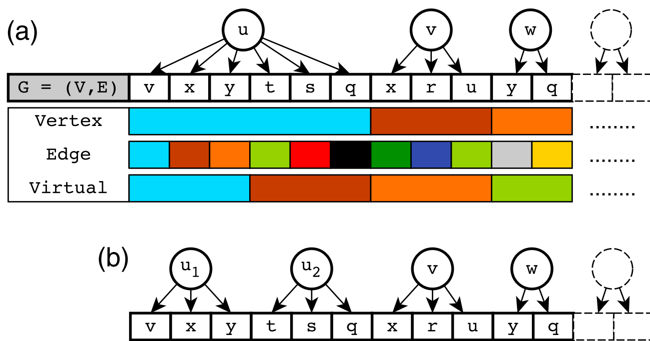


Fig. 1. (a) Vertex-, edge-, and virtual-vertex-based parallelization for centrality computation and the distribution of work to GPU threads which are shown with different colors. $\Delta = 3$ for virtual-vertex-based parallelization. (b) The graph structure with virtual vertices.

memory bound. Hence, the speedup one can achieve with a GPU significantly depends on to the irregularities in the graph such as the connectivity pattern and degree distribution which can significantly damage load balancing and memory coalescing. Thus, it may be beneficial to store the graph in the most suitable format that yields a better regularization of computation and memory usage, hence a better performance.

There are three parallelization techniques that have been used and experimented for closeness and betweenness centrality computations; vertex-based [11,29], edge-based [11], and virtual-vertex-based [24]. The difference between these techniques is the granularity of the parallelism which impacts both load balancing and the need for synchronization. One of the common storage format for graphs is compressed adjacency list, where the adjacency lists of the vertices are stored consecutively with a secondary pointer array that keeps the start/end pointers which require $(m+n+1)$ values in total. Another commonly used format stores the endpoints of each edge individually, hence $2m$ values are required. To ease the memory accesses, vertex-based parallelism uses the former and each thread processes an adjacency list at each kernel execution throughout a level-synchronized BFS. On the other hand, the edge-based parallelism uses the latter and each thread processes only a single edge.

As Jia et al. shows, the vertex-based parallelism on GPU suffers from load balancing especially for the graphs with skewed degree distributions [11]. On the other hand, the edge-based parallelism uses more memory and more atomic operations. For some irregular/graph applications with skewed work distributions, hybrid execution schemes, that splits the work required by each task (vertices in graph context), have been used in many application domains. Çatalyürek and Aykanat [8] proposed one of the first methods that directly optimizes such decomposition for irregular computations. In the graph analysis context, Yoo et al. [33] uses a similar approach to partition the edges of vertices to scale BFS computations on BlueGene/L. PowerGraph [9] also uses similar vertex splitting idea (called *vertex-cut* in their system) to reduce communication in their framework. In our earlier work on centrality computation in GPU [24], we proposed a simpler degree-based vertex splitting method called *virtual-vertex-based parallelism*.

Simply speaking, *virtual-vertex-based parallelism* replaces a problematic, high-degree vertex u with $n_\Delta(u) = \lceil adj(u)/\Delta \rceil$ virtual vertices each having at most Δ edges. That is $n_\Delta(u)$ threads are responsible from processing the edges of a vertex u; and these threads have the same amount of work which improves the load balance. Fig. 1 summarizes how the threads process the edges in vertex-, edge-, and virtual-vertex-based parallelization for centrality computation. In the figure, different threads are shown with different colors and $\Delta = 3$ is used. One can see that the load is imbalanced when *vertex-based* parallelism is used. The load is balanced using *edge-based* parallelism but the granularity of the computation is very fine which increases the synchronization cost (number of atomic operations). Using virtual vertices, each thread has a more balanced amount of work and overall less synchronization (atomic operations) are required. More details are available in [24].

For betweenness centrality, we use virtual-vertex parallelism on GPU since it performed better than the other techniques in our preliminary experiments [24]. We will use n_Δ for the number of virtual vertices and $adj_\Delta(u_*)$ to denote the adjacency list of the arbitrary virtual vertex u_* for an original vertex u in G. For closeness centrality, where we use hardware/software vectorization, we will use the compressed adjacency list format and vertex-based parallelism, since with vectorization multiple simultaneous BFSs, the load balancing problem is resolved almost automatically as we will describe later.

3. Faster network centrality

Surprisingly, all the existing algorithms proposed for betweenness and closeness centrality prefer a pure fine-grain parallelism that employs an iterative execution of a kernel responsible from a single parallel graph traversal. This approach makes sense for accelerators, since they are memory restricted especially considering the size of today's large scale networks. Hence, a coarse-grain approach in which each thread executes a single BFS is unfeasible, and fine-grain BFSs are almost necessary for the device. Yet, an immediate question still needs to be answered: why only one fine-grain BFS at a time? We believe that there is no valid answer. Furthermore, as we will show, doing otherwise can significantly enhance the BC and CC performance without using any additional hardware resource or one with different characteristics.

3.1. A more regular and denser betweenness centrality kernel on GPU

For GPU-based BC, we propose a novel parallelization technique which employs simultaneous BFSs where each thread is responsible for processing a (virtual) vertex in a single BFS. On a GPU, there are several ways to do that including manually partitioning the threads for the BFSs or using concurrent streams. In this work, we use interleaved BFSs to achieve a better memory access pattern.

As stated above, all the existing studies that focus on parallel centrality computations employ level-synchronized BFSs: the ℓth kernel execution is responsible from the ℓth level of the BFS, visits the vertices on it, and processes the corresponding adjacency lists to find the vertices in the $\ell + 1$th level to update their distance information. Note that there are at most L such kernel executions where L is the diameter of the shortest path graph, i.e., the longest distance from the BFS source to a vertex in G. However, the adjacency list of a specific (virtual) vertex u will be processed only in one of these L kernel executions. For the other $L - 1$ executions, the thread responsible for vertex u (u_*) in that BFS may be forced to wait for another thread in the same warp.

Algorithm 3 shows the baseline GPU implementation for the forward phase of Brandes' betweenness centrality algorithm that starts from level $\ell = 1$ and ends at $\ell = L$ when no new vertex is *visited* in the previous kernel execution (the backward phase starts with level $\ell = L$ and stops at $\ell = 1$, and has a similar structure). As described above, this baseline may not utilize the warps efficiently especially for the networks with a large diameter. In fact, the virtual-vertex parallelism solves this problem up to some level when the degrees in the network are considerably larger than

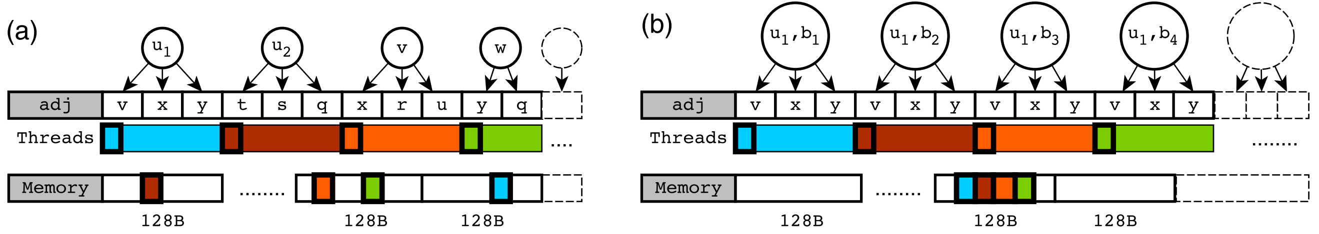


Fig. 2. A toy example given to show the uncoalesced and coalesced memory access patterns of the virtual-vertex-based scheme (left) and the proposed approach (right) respectively. On the left, three memory transactions are required whereas on the right a single transaction is sufficient (assuming the virtual vertex u_1 is on the same level in all the BFSs).

Algorithm 3: VIRBC ($G = (V, E)$)

```
1  ···
2  ℓ ← 0
   ▷Forward phase
3  visited ←true
4  while visited = true do
5      visited ← false
       ▷Forward-step kernel
6      for each thread t in parallel do
7          if t ≤ n_Δ then
8              u_* ← t                                  ▷virtual vertex
9              u ← the vertex in G corresponding to u_*
10             if dst[u] = ℓ then
11                 for each v ∈ adj_Δ(u_*) do
12                     if dst[v] = ∞ then
13                         dst[v] ← ℓ + 1, visited ← true
14                     if dst[v] = ℓ + 1 then
15                         σ[v]←σ[v] + σ[u]             ▷atomic
16     ℓ ← ℓ + 1
17 ···
   ▷Backward phase
18 ···
```

Δ, where Δ is the maximum number of edges a vertex can be connected. In this case, the consecutive threads will be responsible for the virtual vertices u_* each having Δ edges and coming from the same origin vertex u. Hence, the threads responsible from these virtual vertices will perform essentially the same amount of operation at the same time independently from the BFS source. Since there is less thread divergence, the warps, so the device, will be utilized more effectively.

Using virtual vertices is not a "be all and end all" solution to accelerate BC on a GPU. As we will show in the experiments, the performance it yields is not close the peak performance of the device for many cases. There are several reasons for this low performance: first, when the average degree in the network is low, which may be the case for many sparse networks, its impact on the execution scheme is minimal and considering its overhead, it can be even negative. Furthermore, virtual-vertex-based parallelism does not regularize the uncoalesced memory access pattern which is usually the most important problem of the memory-bounded GPU-based algorithms on sparse matrices, graphs, and networks.

In a GPU, the threads in half-warps coordinate global memory accesses into a single transaction. If these accesses are uncoalesced (coalesced to many memory blocks), the required information is transferred via multiple 32B, 64B, and 128B transactions which drastically reduce the performance. Consider the toy example in Fig. 2(a), where 4 consecutive threads in the same warp (and in the same half-warp) visit their virtual vertices and process the first (neighbor) vertices in the corresponding adjacency lists. Since these adjacency lists are different, the memory locations the threads access, e.g., dst[.], can be in different blocks. Considering how level-synchronized BFSs work, 3 transactions are required for the coordinated memory access in Fig. 2(a).

The key idea in this paper is deviating from the pure fine-grain, single-BFS parallelism to a hybrid, coarse/fine-grain parallelism with a motivation to regularize memory access patterns by employing multiple BFSs in batches. For GPU-based BC, we aim for sets of consecutive threads that process a (virtual) vertex simultaneously for multiple BFSs. That way the memory access patterns will gain some regularity in BC kernels which, typically, a single parallel BFS lacks.

Algorithm 4: VIRBC-MULTI ($G = (V, E)$)

```
   ▷𝓑: number of BFSs performed in a batch
1  ···
2  ℓ ← 0
   ▷Forward phase
3  visited ← true
4  while visited = true do
5      visited ← false
       ▷Forward-step kernel
6      for each thread t in parallel do
7          if t ≤ 𝓑 × n_Δ then
8              u_* ← ⌈t/𝓑⌉                              ▷virtual vertex
9              b ← t mod 𝓑                               ▷BFS id
10             u ← the vertex in G corresponding to u_*
11             ω_u ← u × 𝓑 + b
12             if dst[ω_u] = ℓ then
13                 for each v ∈ adj_Δ(u_*) do
14                     ω_v ← v × 𝓑 + b
15                     if dst[ω_v] = ∞ then
16                         dst[ω_v] ← ℓ + 1, visited ← true
17                     if dst[ω_v] = ℓ + 1 then
18                         σ[ω_u] ← σ[ω_v] + σ[ω_u]     ▷atomic
19     ℓ ← ℓ + 1
20 ···
   ▷Backward phase
21 ···
```

Let $\mathcal{B}$ be the number of BFSs in a batch. One can execute a kernel with $\mathcal{B} \times n_\Delta$ threads where the ith BFS is executed by the n_Δ threads starting from the thread $(i-1) \times n_\Delta$. However, this only handles the work in less kernel calls without regularizing the memory accesses. For this reason, we employ interleaved BFSs. Algorithm 4 implements the idea for the forward phase of BC. Its most important difference from Algorithm 3 lies within the memory accesses to the arrays dst[.] and σ[.]: in VIRBC, the neighbor vertex ids have a very high impact in the locality of memory accesses by consecutive threads. Since they can be different, the blocks that need to be accessed by a half-warp can be at various places of the memory. On the other hand, in VIRBC-MULTI, the memory index ω_v computed for a vertex/BFS pair will differ only by one for two consecutive threads processing the same virtual vertex. Hence, $\mathcal{B}$

consecutive threads will access consecutive memory locations and a single transaction may be sufficient for the coordinated memory access as Fig. 2(b) shows for our toy example. In Algorithm 4, the arrays dst[.] and σ[.] are of length $\mathcal{B} \times n$. Hence, except the graph G, the memory footprint of VirBC-Multi is $\mathcal{B}$ times larger than that of VirBC which is one of the drawbacks of our solution. That being said, a small $\mathcal{B}$ would be sufficient to increase the performance as the experiments will show.

We are aware that a vertex will not appear exactly in the same level for all $\mathcal{B}$ BFSs in a batch. Hence, although $\mathcal{B}$ consecutive threads are responsible from the same (virtual) vertex, it is not guaranteed that all these threads will process the adjacency lists in the same kernel execution. But even in the original virtual-vertex-based scheme with a single BFS execution, such a guarantee was there only for the consecutive virtual vertices generated from the same original vertex. For the warps processing such vertices, our modification on the parallelism can be considered as a trade-off of warp utilization (occupancy) and the ratio of coalesced memory accesses. On the other hand, for the warps which already process the original vertices with degree Δ or less, the utilization will probably not be harmed. Furthermore, recent studies show that most of the vertices in G appear only in the middle levels of any BFS for the networks with small-world properties, which typically includes social networks [3,26]. Thus the proposed scheme can even increase the warp occupancy.

The overhead of the proposed technique increases when the maximum level L for the BFSs fluctuates, i.e., when the variance of their distribution is high. If this is the case the BFSs in a pack that are already completed will stay in the process and wait for the one in the pack with the highest L value. Fortunately, the real-life networks exhibit small-world network characteristics and have small diameters, hence L does not fluctuate for the BFSs.

3.2. A more regular and denser closeness centrality kernel on GPU and Intel Xeon Phi

Having irregular memory access and computation that prevent a proper vectorization is a common problem of sparse kernels. The most emblematic sparse computation is certainly the multiplication of a sparse matrix by a dense vector (SpMV). In SpMV, the problem of improving vector-register (also called *SIMD register*) utilization and regularizing the memory access pattern was deeply studied and methods such as register blocking [7,32] or by using different matrix storage formats [4,14] have been proposed. Arguably, the most efficient method to regularize the memory access pattern is to multiply a sparse matrix by multiple vectors if this is possible. When the multiple vectors are organized as a dense matrix, the problem becomes the multiplication of a sparse matrix by a dense matrix (SpMM). While each nonzero of the sparse matrix causes the multiplication of a single element of the vector in SpMV, it causes the multiplications of as many consecutive elements of the dense matrix as its number of columns in SpMM.

Adapting that idea in closeness centrality essentially boils down to the computing multiple sources at the same time, simultaneously. But contrarily to SpMV, where the vector is dense hence each non-zero induces exactly one multiplication, in BFS, not all the non-zeros will induce operations. In other words, a vertex in BFS may or may not be traversed depending on which level is currently being processed. Therefore, the traditional queue-based implementation of BFS does not seem to be easily extendable to support multiple BFSs in a vector-friendly manner.

3.2.1. An SpMV-based formulation of closeness centrality

The main idea is to revert to a more basic definition of level synchronous BFS traversal. Vertex v is part of level ℓ if and only if one of the neighbor of v is part of level $\ell - 1$ and v is not part of any level $\ell' < \ell$. This formulation is commonly used in parallel implementation of BFS on GPU [11,22,29] but also in some shared memory [1] and distributed memory implementations [6].

The algorithm is better represented using binary variables. Let x_i^ℓ be the binary variable that is true if vertex i is part of the frontier at level ℓ for a BFS. The neighbors of level ℓ is represented by a vector $y^{\ell+1}$ computed by

$$y_k^{\ell+1} = \mathtt{OR}_{j \in adj(k)} x_j^\ell .$$

The next level is then computed with

$$x_i^{\ell+1} = y_i^{\ell+1} \mathtt{AND}\ \mathtt{not}\ (\mathtt{OR}_{\ell' \leq \ell} x_i^{\ell'}).$$

Using these variables, one can update the closeness centrality value of vertex i by adding $\frac{x_i^\ell}{\ell}$ if i is at level ℓ. One can remark that $y^{\ell+1}$ is the result of the "multiplication" of the adjacency matrix of the graph by x^ℓ in the (OR,AND) semi-ring.

Implementing BFS using such an SpMV-based algorithm changes its asymptotic complexity. The traditional queue-based BFS algorithm has a complexity of $\mathcal{O}(|E|)$. But the complexity of the SpMV-based algorithm described above depends on how the adjacency matrix is stored. If it is stored column-wise, then it is easy to traverse column j only if the value of x_j^ℓ is true. This leads to an $\mathcal{O}(|E|)$ implementation of BFS, and such an implementation is not essentially different from the queue-based implementation of BFS: they both follow a top-down approach. However, when x_j^ℓ is true, the updates on the entries of $y^{\ell+1}$ vector cause scattered writes to memory which are problematic when executed in parallel.

On the other hand, by storing the adjacency matrix row-wise, different values of x^ℓ are gathered to compute a single element of $y^{\ell+1}$. This yields a bottom-up implementation of BFS which has a natural write access pattern. However, it becomes impossible to only traverse the relevant nonzero of the matrix and the complexity of the algorithm becomes $\mathcal{O}(|E| \times L)$, where L is the diameter of the graph. This is the implementation that we favor and we do not feel that this asymptotically worse complexity is a problem since it has been noted many times before that social networks have small world properties. So, their diameter tends to be low. Note that the small world property only informs on the average distance between two vertices is proportional to $\log(|V|)$ while we are interested in the maximum distance. There could be small world graphs with a long chain on where our technique might not apply as gracefully.

3.2.2. An SpMM-based formulation of closeness centrality

It is easy to derive an algorithm from the formulation given above for closeness centrality that processes multiple sources at once (see Algorithm 5). The algorithm processes sources by batches of $\mathcal{B}$. For each level ℓ, it builds a binary matrix x^ℓ where $x_{i,s}^\ell$ indicates if vertex i is at distance ℓ of source vertex s where $0 \leq s < \mathcal{B}$ is the relative source id in the batch. The first part of the algorithm is Init which computes x^0.

After Init, the algorithm performs a loop that iterates over the levels of the BFSs. The second part is SpMM which builds the matrix $y^{\ell+1}$ by multiplying the adjacency matrix with x^ℓ. After each SpMM, the algorithm enters its Update phase where $x^{\ell+1}$ is computed and then the closeness centrality values are updated using the information of level $\ell + 1$.

By letting $\mathcal{B}$ be the size of the vector register of the machine used, a row of the x and y matrices exactly fits in a vector-register, and all the operations become vector-wide OR, AND and not and bit-count operations. Fig. 3 presents an implementation of this algorithm using AVX instructions ($\mathcal{B} = 256$). We use similar codes

Algorithm 5: CC-SpMM: SpMM-based centrality computation

Data: $G = (V, E)$, $\mathcal{B}$
Output: ccent[.]
▷Init
1 ccent[v] ← 0, $\forall v \in V$
2 $\ell \leftarrow 0$
3 partition V into k batches $\Pi = \{V_1, V_2, \ldots, V_k\}$ of size $\mathcal{B}$
4 **for each batch** *of vertices* $V_p \in \Pi$ **do**
5   $x^0_{s,s} \leftarrow 1$ if $s \in V_p$, 0 otherwise
6   **while** $\sum_i \sum_s x^\ell_{i,s} > 0$ **do**
    ▷SpMM
7     $y^{\ell+1}_{i,s} = \text{OR}_{j\in adj(i)} x^\ell_{j,s}, \forall s \in V_p, \forall i \in V$
    ▷Update
8     $x^{\ell+1}_{i,s} = y^{\ell+1}_{i,s}$ AND not$(\text{OR}_{\ell'\le\ell} x^{\ell'}_{i,s}), \forall s \in V_p, \forall i \in V$
9     $\ell \leftarrow \ell + 1$
10   **for all** $v \in V$ **do**
11   ccent[v] ← ccent[v] + $\frac{\sum_s x^\ell_{v,s}}{\ell}$
12 **return** ccent[.]

```
void cc_cpu_256_spmm (int* xadj, int* adj, int n, float* cc)
{
  int b = 256;
  size_t size_alloc = n * b / 8;
  char* neighbor = (char*)_mm_malloc(size_alloc, 32);
  char* current = (char*)_mm_malloc(size_alloc, 32);
  char* visited = (char*)_mm_malloc(size_alloc, 32);
  for (int s = 0; s < n; s += b) {
    //Init
    ...
    int cont = 1, level = 0;
    while (cont != 0) {
      cont = 0; level++;
      //SpMM
#pragma omp parallel for schedule (dynamic, CC_CHUNK)
      for (int i = 0; i < n; ++i) {
        __m256 vali = _mm256_setzero_ps();
        for (int j = xadj[i]; j<xadj[i+1]; ++j) {
          int v = adj[j];
          __m256 state_v = _mm256_load_ps((float*)(current + 32 * v));
          vali = _mm256_or_ps (vali, state_v);
        }
        _mm256_store_ps ((float*)(neighbor + 32 * i), vali);
      }
      //Update
      float flevel = 1.0f / (float) level;
#pragma omp parallel for schedule (dynamic, CC_CHUNK)
      for (int i = 0; i < n; ++i) {
        __m256 nei = _mm256_load_ps ((float *)(neighbor + 32 * i));
        __m256 vis = _mm256_load_ps ((float *)(visited + 32 * i));
        __m256 cu = _mm256_andnot_ps (vis, nei);
        vis = _mm256_or_ps (nei, vis);
        int bcnt = bitCount_256(cu);
        if (bcnt > 0) {
          cc[i] += bcnt * flevel; cont = 1;
        }
        _mm256_store_ps ((float *)(visited + 32 * i), vis);
        _mm256_store_ps ((float *)(current + 32 * i), cu);
      }
    }
  }
  _mm_free(neighbor); _mm_free(current); _mm_free(visited);
}
```

Fig. 3. Hardware vectorization using AVX for the SpMM-based formulation of closeness centrality.

to leverage 32-bit integer types, SSE registers and Xeon Phi's 512-bit registers in the experiments. The code uses three arrays to store the internal state of the algorithm. `current` stores x^ℓ for the current level ℓ, `neighbor` stores $y^{\ell+1}$ and `visited` stores $\text{OR}_{\ell'\le\ell} x^{\ell'}$. The function `bitCount_256(.)` calls the appropriate bit-counting instructions.

A potential drawback of the SpMM variant of the closeness centrality algorithm is that each traversal of the graph now accesses a wider memory range than the one used in an SpMV approach. This can harm the cache locality of the algorithm. To see the impact on cache-hit ratio, we wrote a simulator to emulate the cache behavior during the SpMM operation. The simulator assumes that the computation is sequential; the cache is fully associative; it uses cache-lines of 64 bytes; only the x vector (`current` array in the code) is stored in the cache; and the cache is completely flushed between iterations.

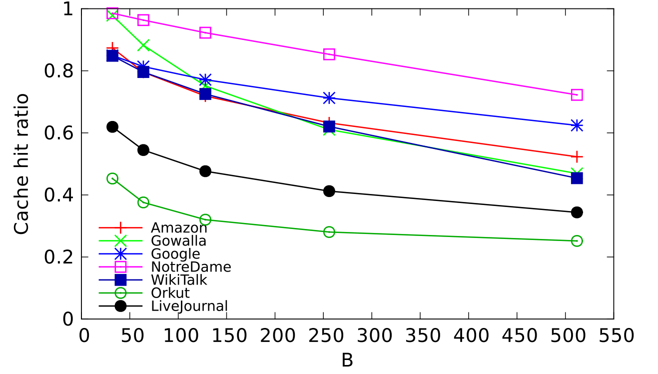


Fig. 4. Simulated cache-hit ratio of the SpMM variant on a 512 K cache (e.g., Intel Xeon Phi's L2 cache).

Fig. 4 presents the cache-hit ratios with a cache size of 512 K (the size of Intel Xeon Phi's L2 cache) for different number of BFSs and for the seven graphs we will later use in the experimental evaluation. The cache hit-ratio degrades by about 20%–30% when the number of concurrent BFSs goes from 32 to 512. This certainly introduces a significant overhead, but we believe it should be widely compensated by reducing the number of iterations of the outer loop by a factor of 16.

3.2.3. Software vectorization

The hardware vectorization of the SpMM kernel presented above limits the number of concurrent BFS sources to the size of the vector registers available on the architecture. However, there is no reason to limit the method to the size of a single register. One could use two registers instead of one and perform twice more sources at once. The penalty on the cache locality will certainly increase, but probably not by a factor of two.

Since we want to try various number of simultaneous BFS, the implementation effort for manual vectorization of each version becomes prohibitive. Therefore, we developed a unique code that allows to easily change the number of concurrent source traversed. Fig. 5 presents a fragment of this code which has been carefully written to allow the compiler to leverage vector instructions where possible. The key of this code is to specify the number of simultaneous traversals as a C++ template parameter instead of using a function parameter. This forces the compiler to generate a different object code for each value of the template parameter `vector_size` (expressed in number of 32-bit words). Therefore, it allows the compiler on a CPU architecture to utilize the SSE instructions if `vector_size` is 4 or to utilize the AVX instructions if it is more than 8. The right template parameter is selected in a wrapper function (not shown here).

Instead of using explicit registers, this compiler vectorized code expresses the state of the x vector as an array of 32-bit integers. The compiler is hinted at unrolling these accesses to prevent a loop and expose their vectorial nature. Though, the C++ language does not directly allow that vectorization to take place because the various pointers of the function might point to overlapped memory. The `__restrict__` language extension is used to instruct the compiler that none of the arrays will ever overlap, allowing the compiler to generate the vector instructions when it believes that they are appropriate. As the experiments will

```
template<int vector_size>
void cc_cpu_spmm_soft_vec_t (int* __restrict__ xadj,
                             int* __restrict__ adj,
                             int n, float* __restrict__ cc)
{
  int b = vector_size * 32, n_align = b / 8;
  size_t size_alloc = n * b / 8;
  char* __restrict__ neighbor = (char*)_mm_malloc(size_alloc, n_align);
  char* __restrict__ current = (char*)_mm_malloc(size_alloc, n_align);
  char* __restrict__ visited = (char*)_mm_malloc(size_alloc, n_align);
  for (int s = 0; s < n; s += b) {
    //Init
    ...
    int cont = 1, level = 0;
    while (cont != 0) {
      cont = 0; ++level;
      //SpMM
#pragma omp parallel for schedule (dynamic,CC_CHUNK)
      for (int i = 0; i < n; ++i) {
        int vali[vector_size];
#pragma unroll
        for (int k = 0; k < vector_size; ++k)
          vali[k] = 0;
        for (int j = xadj[i]; j < xadj[i+1]; ++j) {
          int v = adj[j];
#pragma unroll
          for (int k = 0; k < vector_size; k++)
            vali[k] = vali[k] | ((int*)current)[v*vector_size+k];
        }
#pragma unroll
        for (int k = 0; k < vector_size; ++k)
          ((int*)neighbor)[i*vector_size+k] = vali[k];
      }
      //Update
      ...
    }
  }
  _mm_free(neighbor); _mm_free(current); _mm_free(visited);
}
```

Fig. 5. Compiler vectorization for the SpMM-based formulation of closeness centrality.

show, the compiler-based vectorization in Fig. 5 perform almost as good as the manually vectorized code given in Fig. 3 which is useful in practice since the compiler-based vectorization is much more flexible to change the number of simultaneous BFSs.

3.2.4. Closeness centrality on GPU

The SpMM-based approach for closeness centrality can be directly adapted for GPU since the hardware is already modeled for SIMD execution. Simply put, one can consider one operation on a CUDA warp as one Xeon Phi SIMD operation. For our implementation, we used 64-bit integers to store parts of `current`, `neighbor`, and `visited` arrays per thread. Thus, each thread can use a bitwise operation to process 64 BFSs simultaneously. When a vertex is assigned to a single GPU warp (containing 32 threads), $\mathcal{B} = 32 \times 64 = 2048$ BFSs can be handled simultaneously. For memory-bound kernels such as a graph traversal, only a half-warp (16 threads) may also be considered as a counterpart of a SIMD operation on Xeon Phi, since the GPU coordinates the global memory accesses of the threads in a half-warp into a single transaction. Or similar to software vectorization, one can go wider and use more than a warp per vertex to support more than 2048 BFSs. We experimented with these three options and assign a vertex to 16, 32, and 64 threads. A similar direction one can follow to increase $\mathcal{B}$ is assigning more work to each thread. That is, by doubling the work of a single thread and assigning 128 BFSs to a thread and a vertex to a warp, one can handle $\mathcal{B} = 4096$ BFSs at once, and hence, halves the number of kernel executions. However, while following any of these approaches, we are always limited by the memory footprint of the kernel which is a problem for a memory-restricted device such as GPU.

For our GPU-based CC implementation, we used the traditional compressed adjacency list format instead of virtual-vertices we employed in our GPU-based BC implementation. As explained in Section 2.3.1, virtual vertices are proposed to improve the load balance inside a CUDA warp for a single BFS. Since each thread is responsible for a single BFS and when a vertex is not on the current level ℓ of the corresponding BFS, the thread needs to wait the others in the warp. However, in our CC implementation, since a thread is responsible for multiple BFSs (i.e., 64 of them) it is more likely that at least in one of these BFSs, the thread will need to work. Thus, warp occupancy is expected to be high for the SpMM-based CC. Furthermore, when a single vertex is assigned to a warp, each thread will visit the same adjacency list. Thus, there will not be a load balancing problem and the memory accesses will be highly coalesced. In our experiments, we compared the performance of the SpMM-based implementation (`GPU-SpMM`) with the virtual-vertex-based ones with one BFS at a time (`GPU-VirCC`) and multiple BFSs (`GPU-VirCC-Multi`), where the latter adapts the parallelization techniques we explained for BC in Section 3.1.

3.2.5. Implementation details

We improved the performance of the SpMM-based implementation given in Fig. 3 (as well as the compiler-vectorized one in Fig. 5 and GPU-based implementation), by employing two simple modifications. In the first modification, which is in the `SpMM` part of Fig. 3, before traversing the adjacency list of the ith vertex, the algorithm checks that if all the $\mathcal{B} = 256$ `visited` bits corresponding to $\mathcal{B}$ BFSs assigned to the thread were already set to 1 by the previous or current level expansions. If this is the case, since the vertex has already been visited in all the BFSs, the thread skips the `SpMM` part and directly goes to the `Update` part. For the GPU implementation, each thread checks the corresponding 64 bits in the `visited` array. Note that, a warp in the SpMM kernel can terminate only when the $32 \times 64 = 2048$ `visited` bits are already equal to 1.

The second modification is similar to the first one but this time it is in the *Update* part of Fig. 3: when all the $\mathcal{B}$ bits in the `visited` array were already set to 1, the code sets the corresponding `current` bits to 0 and ends the `Update` part without any other bitwise operations or bit counting. Similar to the first modification, in the GPU-based CC implementation, each thread in a warp takes this shortcut by using the 64 bits corresponding to the `visited` information of the 64 BFSs and sets the corresponding 64 bits in the `current` array to 0.

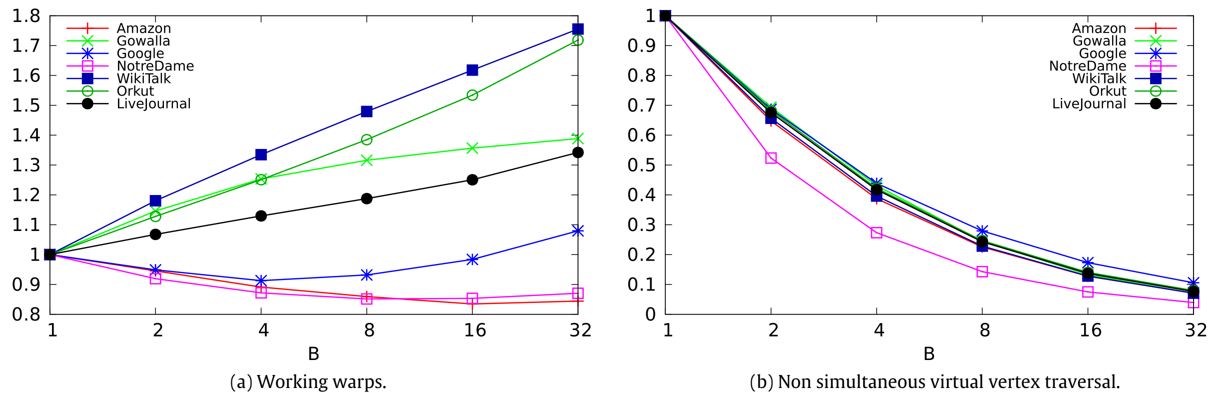
4. Experiments

The experiments were carried out on a system equipped with two Intel Sandy Bridge-EP CPUs clocked at 2.00 GHz and 256 GB of memory split across two NUMA domains. Each CPU has eight-cores (16 cores in total) and HyperThreading is enabled. Each core has its own 32 kB L1 cache and 256 kB L2 cache. The 8 cores on a CPU share a 20MB L3 cache. The machine is equipped with an NVIDIA Tesla K20c GPU featuring 13 Streaming Multiprocessors, 192 cores per SM clocked at 700 MHz (for a total of 2496 CUDA cores), and 4.8 GB of global memory clocked at 2.6 GHz. ECC is enabled. The system also has an Intel Xeon Phi coprocessor with 8 memory controllers and 61 cores clocked at 1.05 GHz. There is a 32kB L1 data cache, a 32 kB L1 instruction cache, and a 512kB L2 cache associated with each core. The bandwidth of each core is 8.4 GB/s where the cores' memory interface are 32-bit wide with two channels. Although the cores are expected to provide 512.4 GB/s, the bandwidth between the memory controllers and they are limited by the ring network in between which theoretically supports at most 220 GB/s.

On the software side, we run a 64-bit Debian with Linux 2.6.39-bpo.2-amd64. All the codes are compiled with GCC with the -O3 optimization flag in version 4.4.4. Xeon Phi codes are compiled with the Intel C++ Compiler in version 13.1 using -O3 optimization

Table 1
Properties of the largest connected components of the graph used in the experiments.

Graph	$\|V\|$	$\|E\|$	Avg. $\|adj(v)\|$	Max. $\|adj(v)\|$	Diam.
Amazon	403 K	2,443 K	6.0	2,752	19
Gowalla	196 K	950 K	4.8	14,730	12
Google	855 K	4,291 K	5.0	6,332	18
NotreDame	325 K	1,090 K	3.3	10,721	27
WikiTalk	2,388 K	4,656 K	1.9	100,029	10
Orkut	3,072 K	117,185 K	38.1	33,313	9
Live Journal	4,843 K	42,845 K	8.8	20,333	15



(a) Working warps.

(b) Non simultaneous virtual vertex traversal.

Fig. 6. Analyzing the behavior of VirBC-Multi. The values are normalized relatively to the case $\mathcal{B} = 1$ and accumulated over the iterations of a batch.

flag. CUDA 5.0 is used with flag -arch sm_20. We have carefully implemented all the algorithms using C++. To have a base-line comparison, we implemented OpenMP versions of the CPU-based betweenness and closeness centrality algorithms. Note that, our system has 16 cores. When implementing the CPU based closeness centrality code, we made use of the direction optimization technique, presented in [3]. Other than direction optimization, no particular optimizations have been applied to the CPU codes except the ones performed by the compiler. We also used various studies from the literature to evaluate the practical performance of our GPU-based betweenness centrality implementation with virtual vertices and multiple BFSs and SpMM-based closeness centrality implementation.

For the experiments, we used a set of graphs from the SNAP dataset.[1] Directed graphs were made undirected and the largest connected component is extracted and used in the experiments. The list of graphs and the properties of the largest components that are used in our experiments can be found in Table 1.

All the results presented in this section are computed by using the total application time from the moment where the graph is fully loaded into the main memory of the machine to the moment where the final centrality values are available in the main memory of the node. In particular, the time excludes reading the graph from the hard drive; but it includes the transformations such as virtualization and all the communications between the host and the device. Using these times, we computed the traversed edges per second (TEPS) values and report them on the figures in this section. Given the total application *time* (in seconds) for K sources/BFSs on a graph with m (undirected) edges, the TEPS value is equal to $(m \times K)/time$. Note that to process K sources, the algorithm needs $K/\mathcal{B}$ kernel executions where each of the kernels handles $\mathcal{B}$ sources.

[1] http://snap.stanford.edu/data/index.html.

4.1. Evaluating the proposed betweenness centrality algorithm VirBC-Multi

In this section, we investigate the efficiency of our virtual-vertex based BC algorithm. We first analyze the VirBC-Multi algorithm with different parameters, then present the absolute numbers on its performance by a comparison with existing work in the literature. We used $\Delta = 8$ for virtualization. Since the computations can be extremely long (months), we did not used all the n BFS sources in the graphs and measured the time for 1,024 sources/BFSs in total. We observed that the runtimes of single kernel executions to be very stable, allowing us to make a meaningful extrapolation. Thus, if necessary, the results can be used to linearly extrapolate the runtime for the whole graph.

4.1.1. Analysis of VirBC-Multi

We first investigate the validity of one of the assumptions we make: batching multiple traversals is useful because a vertex only appears in a small number of levels. This assumption is expected to lead to a high number of threads within a warp concurrently expending the same vertex. It should improve the computation by increasing the reutilization of the graph data structure and by structuring the memory accesses made by a warp into regular patterns. To verify this, we computed two indices.

The first index is the number of working warps; in each kernel call, a thread is said to be working if it passes the condition line 12 of VirBC-Multi (Algorithm 4); that is to say, if that vertex is expended. Similarly, a warp is said to be working if one of its 32 threads passes that line. When $\mathcal{B}$ grows, the structure of the warps change leading to differences in the number of working warps. The more working warps there are, the more computation the GPU will need to perform (this simplification may not be true for all the kernels that run on GPUs, but since we employ virtual vertices for BC, each warp takes essentially the same number of operations which makes the simplification valid). In Fig. 6(a),

Table 2
Performance of the betweenness centrality algorithms (in MTEPS).

Graph	CPU-SNAP	CPU-Ligra	CPU-BC	GPU-VirBC	GPU-VirBC-Multi
Amazon	23	116	297	349	591
Gowalla	12	103	535	349	657
Google	17	107	275	377	525
NotreDame	6	62	806	234	441
WikiTalk	9	67	316	346	491
Orkut	52	450	319	275	1,018
LiveJournal	29	253	225	268	701

we present how the number of working warps within the BC computation is impacted by $\mathcal{B}$. Surprisingly, the number of working warps evolves differently for different graphs. For some graphs, e.g., `Amazon`, `NotreDame`, and `Google`, the number initially decreases but then either stabilizes or increases. For some other graphs, e.g., `WikiTalk`, `Orkut`, `Gowalla`, and `LiveJournal`, the number increases. However, overall, the variation of the number of working warps is fairly small; the decrease is never more than 20% and the increase is never more than 75%. This indicates that batching sources has almost no impact on thread divergence and this impact is mostly better for performance.

The second index measures how many times a virtual vertex is non-simultaneously traversed. When $\mathcal{B} = 1$, each virtual vertex is traversed exactly once per source and each of these traversals is performed by a different warp. But when $\mathcal{B}$ increases then a virtual vertex might be traversed multiple times by a single warp, and we say that these two virtual vertices are traversed simultaneously. What we are interested in, overall, is how many different warps traverse a given virtual vertex. Fig. 6(b) shows that the number of non-simultaneous traversals sharply decreases for all the graphs when $\mathcal{B}$ increases. This number improves by more than 85% for all the graphs. This should significantly improve the coalescing of the memory operations performed by various kernels.

We can conclude that the previous increase in the number of working warps is most likely linked to the fact that the warps were naturally well structured because the consecutive vertices are close in the graph and are typically traversed in the same level, thanks to the virtualization. We show the actual impact of varying the number of simultaneous sources $\mathcal{B}$ in performance is in Fig. 7. All the values are normalized to the time taken by the variant that executes one BFS at a time. A first observation is that all the graphs benefit from executing multiple sources in a batch. But the rate of improvement is different. For instance the improvement seen on `Orkut` is very similar to the improvement in non-simultaneous virtual vertex traversal (Fig. 6(b)). On the other hand, other graphs, such as `Amazon`, incur lesser improvements. Finally for some graphs, the normalized time is V-shaped. We guess that the increase in memory occupation with large $\mathcal{B}$ values is detrimental for some cases. Alternatively, it is possible that for some cases, the memory accesses were already fairly well organized and the proposed techniques only have a limited impact.

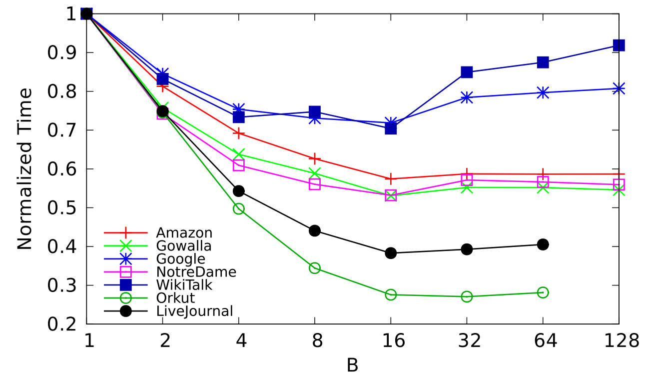


Fig. 7. Impact of $\mathcal{B}$ on VirBC-Multi run on an NVIDIA Tesla K20.

4.1.2. Evaluating the absolute performance

We experimentally evaluated the algorithms given in Section 3.1 on betweenness centrality kernels. There are mainly three variants: OpenMP-based parallel CPU implementation (Cpu-BC), GPU implementation with virtual vertices (VirBC) and VirBC-Multi. We also compared our techniques with the betweenness centrality kernels in the state-of-the-art shared-memory graph processing frameworks Ligra [30] and SNAP [2]. Comparisons are done in terms of *million traversed edges per second* (MTEPS) which is computed as $(1,024 \times |E|)/(10^6 \times time)$, where $|E|$ is the number of (undirected) edges and time is the time required to complete all the 1,024 BFSs we perform for a configuration. For VirBC-Multi, we only report the performance achieved using the best value for $\mathcal{B}$. This value is representative of the performance one can get on a real application since it can easily be discovered at runtime during the first few iterations of the overall algorithm.

Fig. 8 presents the MTEPS for BC algorithms when executed on the seven networks given in Table 1. Precise values are provided in Table 2 for comparison purpose. As Fig. 8 shows, VirBC-Multi is superior to the others on 6 of 7 graphs. On average, VirBC-Multi is 35 times faster than SNAP, 4.7 times faster than Ligra, 68% faster than Cpu-BC and 96% faster than VirBC. In terms of the performance, VirBC-Multi reaches to 1 GTEPS on `Orkut` network.

4.2. Evaluating the proposed SpMM-based closeness centrality algorithm

The closeness centrality experiments are performed using a total of 16384 sources. Hence, for a configuration with $\mathcal{B}$ simultaneous BFSs uses $16384/\mathcal{B}$ kernel executions. Similar to BC experiments, we did not observe a significant variance among the execution times of these executions. The presented TEPS results in the figures are computed by linear extrapolation for the entire graph.

4.2.1. SpMM-based closeness centrality on x86-based architectures

We will first have a look at the performance of our techniques on CPU and Intel Xeon Phi. Since they present similar patterns and Intel Xeon Phi obtains a better performance, we will only present the results for that architecture in this subsection. As a first experiment, we compare the manual and compiler-based hardware vectorization options. For manual vectorization, we implemented 32-bit, 128-bit SSE, 256-bit AVX (see Fig. 3), and 512-bit Intel Xeon Phi versions with various intrinsics supported by the hardware for a concurrent execution of 32, 128, 256 and 512 BFSs, respectively. For compiler-based vectorization, we used the code (partially) given in Fig. 5 without the modifications described in Section 3.2.5 and let the compiler optimize it with -O3 flag. Fig. 9 gives the performance results in terms of billions of traversed edges per second (GTEPS). The bars in the figure with `-comp`

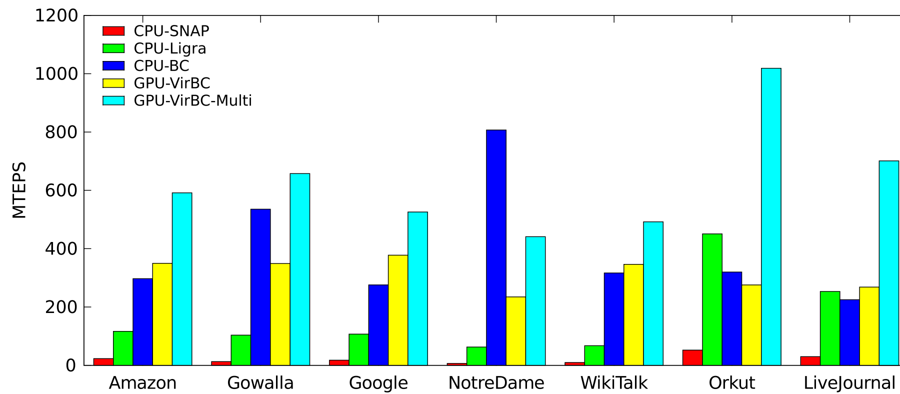


Fig. 8. Evaluation of the betweenness centrality algorithms in terms of MTEPS. The values for the proposed algorithms are the best ones we obtained with different $\mathcal{B}$ values.

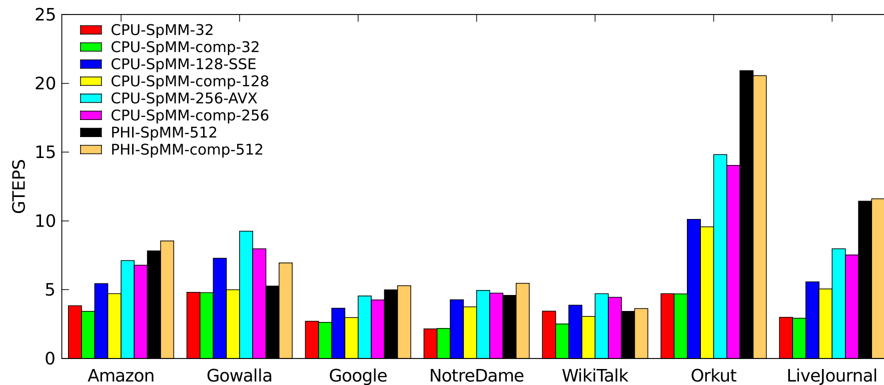


Fig. 9. The compiler- and manually-vectorized implementation of SpMM-based closeness centrality reach similar performance.

keyword are the ones with the compiler-vectorized versions. For almost all the graphs, 512-bit Intel Xeon Phi vectorization gives the best results. For `Gowalla`, `NotreDame`, and `WikiTalk`, 256-bit versions are better. Overall, manual vectorization is only slightly better than compiler-based vectorization. This shows that if the code is properly written, the compiler does its job and optimizes relatively well. We will mainly use the compiler-vectorized implementation in the rest of the text, since it is more flexible and its performance is comparable to the manually-vectorized one.

The implementation on Intel Xeon Phi uses the *offload mode*. The memory transfer time is optimized by using large memory pages whose size is set with an environment variable `MIC_USE_2MB_BUFFERS` = 4 K. The memory allocation is performed in two phases; as usual in Linux systems, the memory allocation routine is called to allocate the virtual pages and the physical pages are allocated when the memory is touched for the first time. On Intel Xeon Phi, the physical page allocation is fairly slow. To give a point of reference, in our preliminary experiments, the physical memory allocation required to perform 8192 BFSs on `Google` (2.44 GB) takes 0.88 s; while performing the BFSs takes 1.44 s. Still, this overhead increases only with 8192 and it does not change with the number of sources used for centrality computation. Hence, considering the number of vertices, n, is much larger than $\mathcal{B}$, the overhead are small compared to the time required to process the whole graph for exact centrality computation (we remark that the presented results are extrapolated taking the memory allocation time into consideration). However, their impact can be higher while using sampling and approximation techniques for closeness centrality.

We experimented with values of $\mathcal{B}$ from 32 to 8192 (higher values of $\mathcal{B}$ would lead to out-of-memory for the larger graphs). We present the performance of the compiler-vectorized implementation that benefits from the modifications presented in Section 3.2.5 in Fig. 10. In short, the performance increases with $\mathcal{B}$. We can observe that the rate of improvement is higher when hardware vectorization is leveraged, i.e., when $\mathcal{B}$ is smaller than the register size of Intel Xeon Phi, compared to the case when software vectorization is leveraged. Yet, software vectorization still provides significant performance improvements for all the graphs. In the rest of the experiments, we will use the configuration with 8192 simultaneous traversals since it obtains best performance.

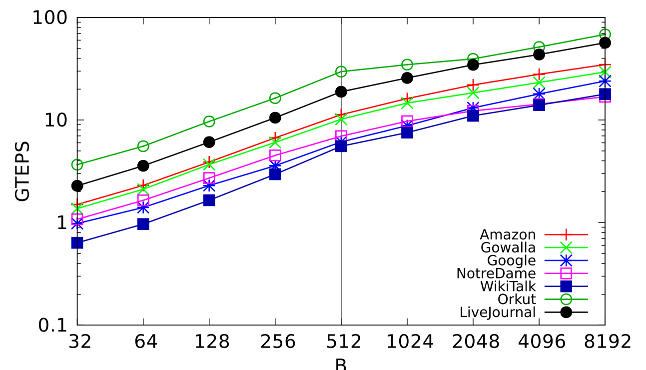


Fig. 10. Impact of the number of simultaneous BFS on the performance obtained on Intel Xeon Phi with the modifications described in Section 3.2.5. The separation between hardware and software vectorization is marked.

To put the results into perspective, in Fig. 11, we compare the performance of the fine-grain BFS technique developed for Intel Xeon Phi presented in [27] (`PHI-BFS-block`), a coarse-grain (32 threads) CC code (`PHI-DO`) that uses direction-optimized BFS idea [3], the hardware-vectorized code with $\mathcal{B}$ = 512 (`PHI-SpMM-512`), the compiler-vectorized version using $\mathcal{B}$ = 8192 BFSs with (`PHI-SpMM-opt-comp-8192`) and with-

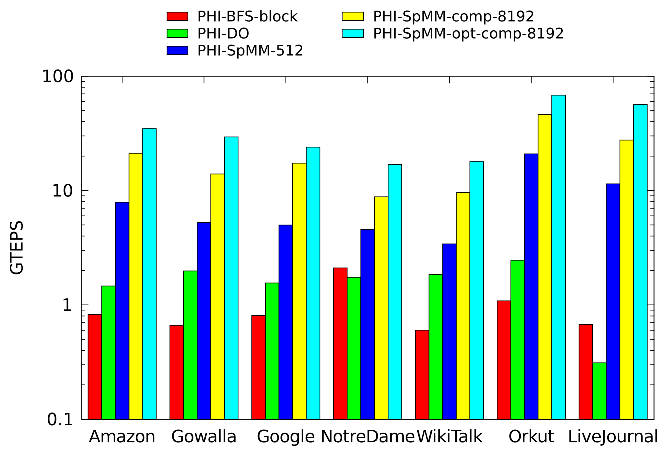


Fig. 11. Performance of the closeness centrality algorithms (and configurations) on Intel Xeon Phi.

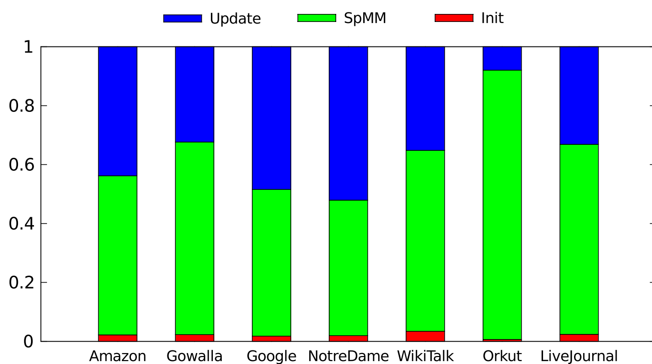


Fig. 12. Proportion of each section of the execution time of PHI-SpMM-comp-opt-8192.

out (PHI-SpMM-comp-8192) the modifications described in Section 3.2.5. The first two methods do not use the proposed densification techniques and obtain low performance: the performance of PHI-BFS-block ranges from 600 MTEPS to 2.1 GTEPS, while PHI-DO sees its performance range from 311 MTEPS to 2.4 GTEPS. On the other hand, PHI-SpMM-opt-comp-8192 is, on the average, 22.1 times faster than PHI-DO and its performance is between 16.8 GTEPS to 68.2 GTEPS. Also, the modifications to take shortcuts bring a 1.75 factor of improvement on the average.

As the other SpMM-based codes, PHI-SpMM-opt-comp-8192 is composed of three phases: Init, SpMM and Update. The relative proportions of the execution times of these phases depend on the structure of the graph as shown in Fig. 12. For all the graphs, the time required for Init is smaller compared to other phases. However, the relative time for SpMM and Update drastically changes with the graph, e.g., see NotreDame and Orkut. Fig. 13 shows how the time of the SpMM phase and the Update phase vary with the iterations among the levels of the BFSs and how many vertices are actually processed in each of these phases. One can see that the time spent in the SpMM phase is fairly well correlated to the number of vertices processed in this phase (thanks to the modifications in Section 3.2.5). A similar pattern exists on the Update phase. The non-modified version, which is not shown here, has much flatter execution times for these two phases; the amount of improvement provided by the modifications depends on the distribution of these skipped vertices which varies from one graph to the other.

4.2.2. SpMM-based closeness centrality on GPU

The performance of the SpMM-based approach on the GPU depends on how many traversals are performed simultaneously,

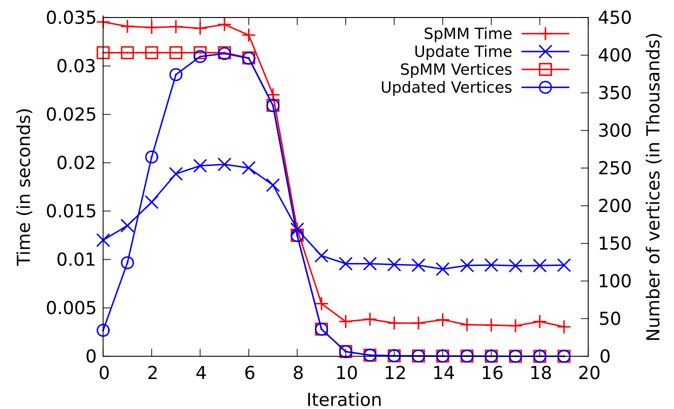


Fig. 13. Time break-down per iteration and number of updated vertices for the Amazon graph. The variation of the time is explained by the number of vertices processed during those phase.

the data type used, and how many threads/warps are used per vertex. Overall, as in Xeon Phi experiments, when $\mathcal{B}$ increases so does the performance. In addition to 64-bit integers, we also tried using 32-bit ones in our preliminary experiments which performed almost always worse than the 64-bit version. Therefore, Fig. 14 uses the 64-bit version and shows the performance of the GPU-based algorithm using different number of threads/warps per vertex. Since the NVIDIA Tesla K20 has a relatively small memory, which is 6 GB, $\mathcal{B}$, the maximum number of simultaneous BFSs, is set to 2048 for Orkut and LiveJournal, 4096 for WikiTalk and 8192 for Amazon, Gowalla, Google, and NotreDame. The figure does not contain 2-warp per vertex (64 threads) configuration for Orkut and LiveJournal since there are only $2048/64 = 32$ integers due to the memory restriction. Hence, even we assign 2-warps per vertex, one of the warps will stay idle.

The performance of the SpMM-based approach varies with the number of threads per vertex. The performance usually increases when we use a single warp (32 threads) instead of a half-warp (16 threads) per vertex (except LiveJournal). This is expected since although the memory accesses of the threads in each half are coordinated, a half still need to wait the other especially when the lengths of the adjacency lists assigned to these halves significantly differ. Using two warps (64 threads) also increases the performance but less frequently. For example, the increase for Amazon and WikiTalk, are not significant, and there is a performance decrease for Google. We will use 32 threads per each vertex in the rest of the text while presenting the performance of GPU-based implementation.

We compare the performance of GPU-SpMM with multiple baselines from the literature in Fig. 15. GPU-LinearBFS is the linear-time, fine-grain, parallel BFS implementation proposed for GPU [19]. GPU-VirCC is a direct adaptation of GPU-VirBC (from [24]), and GPU-VirCC-Multi is a direct adaptation of GPU-VirBC-Multi (from Section 3.1) to closeness centrality. Similar to BC experiments, we used $\Delta = 8$ for virtualization. With the help of simultaneous BFSs, GPU-VirCC-Multi performs better than the single-BFS variant GPU-VirCC. However, the GPU-SpMM algorithm performs one order of magnitude faster than the rest thanks to vectorization and a more compact formulation.

4.2.3. Summary of the closeness centrality experiments

In Fig. 16, we present the performance of the SpMM-based CC implementation on all the three architectures with the best non-vectorized algorithm from the literature and the best vectorized algorithm described in this paper. Table 3 gives precise values for comparison purposes. On CPU and Xeon Phi, 4096 and 8192

Table 3
Performance of the Closeness Centrality algorithms (in MTEPS).

Graph	CPU-DO	CPU-SpMM	PHI-DO	PHI-SpMM	GPU-VirCC	GPU-SpMM
Amazon	1,985	15,146	1,535	34,743	542	40,602
Gowalla	4,340	12,588	2,077	29,409	594	34,759
Google	1,736	10,391	1,632	23,953	516	43,206
NotreDame	2,925	8,956	1,828	16,858	418	22,462
WikiTalk	2,122	11,611	1,940	17,876	462	20,881
Orkut	3,073	28,393	2,548	68,290	801	85,335
LiveJournal	1,879	23,283	326	56,589	609	55,862

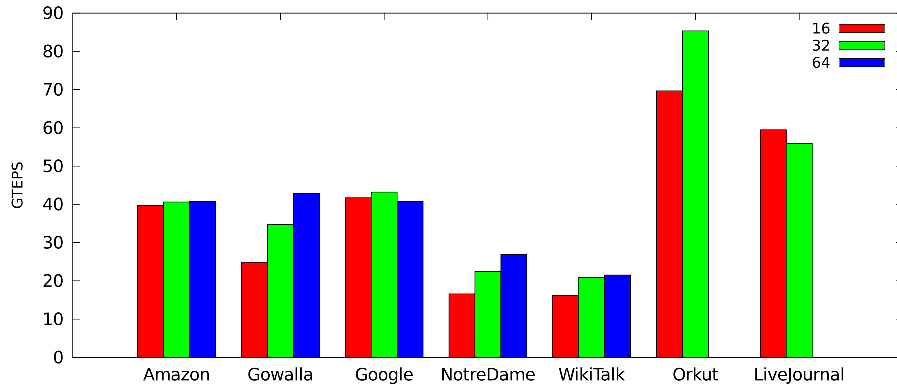


Fig. 14. Impact on the number of threads per vertex on the performance of GPU-SpMM.

simultaneous BFSs, respectively, are used. On GPU, the maximum possible simultaneous BFSs is used for each graph as described above. For the non-vectorized variants, the direction optimized CC variant performs the best on CPU and Xeon Phi, while the `GPU-VirCC` algorithm with simultaneous BFSs performs best on the GPU. On average, the vectorized algorithm is 5.9 times faster than the non-vectorized one on CPU, 21.0 times faster on Intel Xeon Phi, and 70.4 times faster on NVIDIA Tesla K20c than the best existing ones.

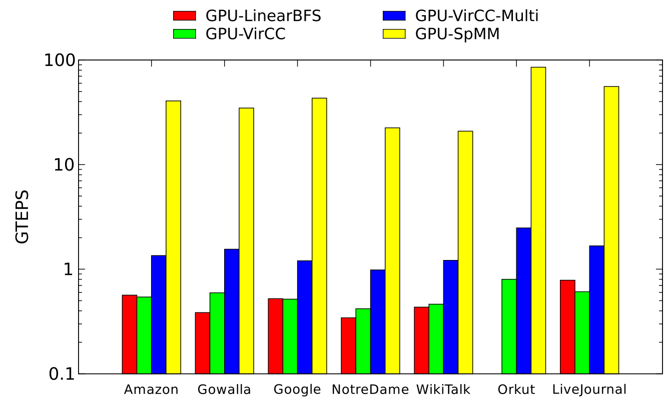


Fig. 15. Comparison of GPU-based closeness centrality algorithms.

5. Conclusion and future work

In this work, we proposed new algorithms and parallelization techniques to make betweenness and closeness centrality computations faster on commonly available cutting edge hardware. There are two traditional ways to execute centrality computations in parallel. Either each thread traverses the graph from a single source, or all the threads collaboratively traverse the graph from a unique source. We deviated from the traditional approaches by using all the threads in the system to collaboratively traverse the graph from many sources simultaneously. This scheme makes the computations more regular and allows a better utilization of modern computing devices. The experimental evaluation of the proposed algorithms shows that significant improvements can be obtained over the best known algorithms for centrality computation on the same device, without using an additional hardware: a improvement of a factor 5.9 on CPU architectures, 70.4 on GPU architectures and 21.0 on Intel Xeon Phi.

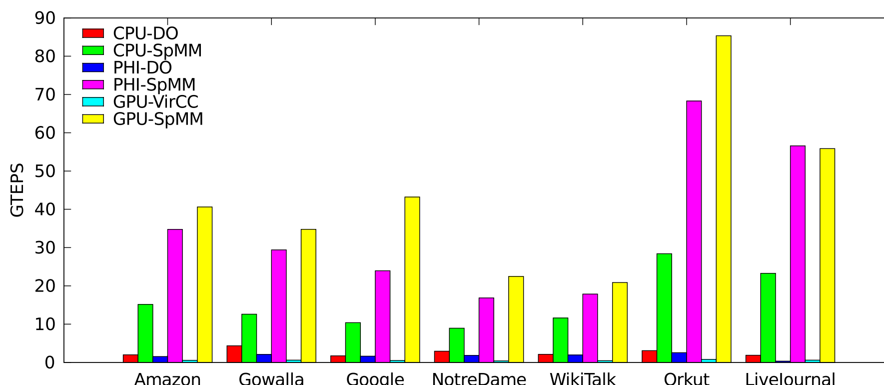


Fig. 16. Vectorization works: `CPU-SpMM` is the compiler-vectorized implementation executed on CPU (32 threads) with $\mathcal{B} = 4096$. `PHI-SpMM` is the corresponding Xeon Phi variant with $\mathcal{B} = 8192$. For the GPU-based implementation, the maximum possible $\mathcal{B}$ value is used for each graph, and a vertex is assigned to a warp (32 threads).

The techniques can be applied to these architectures at the same time. Hence, they are suitable for heterogeneous computing which is straightforward for centrality computations as we have shown in [24]. Furthermore, we believe that the proposed approach is also suitable to compute approximate centrality values and it can be investigated as a future work. In the future, we want to analyze the impact of vectorization in the streaming setting for dynamic networks. But more importantly, we want to investigate whether other common graph computations can be regularized.

Acknowledgments

This work was partially supported by the Defense Threat Reduction Agency grant HDTRA1-14-C-0007. We are also grateful to Intel for providing us the Intel Xeon Phi card used in the experiments and to NVIDIA for providing us the Tesla K20 card.

References

[1] V. Agarwal, F. Petrini, D. Pasetto, D.A. Bader, Scalable graph exploration on multicore processors. in: SuperComputing, 2010, pp. 1–11.

[2] D.A. Bader, K. Madduri, SNAP, Small-world Network Analysis and Partitioning: An open-source parallel graph framework for the exploration of large-scale networks, in: IPDPS, 2008, pp. 1–12.

[3] S. Beamer, K. Asanović, D. Patterson, Direction-optimizing breadth-first search, in: Proceedings of Supercomputing (SC), 2012.

[4] M. Belgin, G. Back, C.J. Ribbens, Pattern-based sparse matrix representation for memory-efficient SMVM kernels, in: Proceedings of ICS, 2009, pp. 100–109.

[5] U. Brandes, A faster algorithm for betweenness centrality, J. Math. Sociol. 25 (2) (2001) 163–177.

[6] A. Buluç, J.R. Gilbert, The combinatorial BLAS: Design, implementation, and applications, Internat. J. High Perform. Comput. Appl. (2011).

[7] A. Buluç, S. Williams, L. Oliker, J. Demmel, Reduced-bandwidth multithreaded algorithms for sparse matrix–vector multiplication, in: Proc. IPDPS, 2011.

[8] Ü.V. Çatalyürek, C. Aykanat, A hypergraph-partitioning approach for coarse-grain decomposition, in: ACM/IEEE SC2001 Denver, CO Nov. 2001.

[9] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: Distributed graph-parallel computation on natural graphs, in: 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 12, 2012, pp. 17–30.

[10] S. Hong, T. Oguntebi, K. Olukotun, Efficient parallel graph exploration on multi-core CPU and GPU, in: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques, PACT'11, 2011.

[11] Y. Jia, V. Lu, J. Hoberock, M. Garland, J.C. Hart, Edge vs. node parallelism for graph centrality metrics, in: GPU Computing Gems: Jade Edition, Morgan Kaufmann, 2011.

[12] S. Jin, Z. Huang, Y. Chen, D.G. Chavarría-Miranda, J. Feo, P.C. Wong, A novel application of parallel betweenness centrality to power grid contingency analysis, in: Proceedings of IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2010, pp. 1–7.

[13] G.J. Katz, J.T. Kider, All-pairs shortest-paths for large graphs on the GPU, in: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware GH'08, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association, pp. 47–55.

[14] X. Liu, M. Smelyanskiy, E. Chow, P. Dubey, Efficient sparse matrix–vector multiplication on x86-based many-core processors, in: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS'13, 2013.

[15] L. Luo, M. Wong, W.m. Hwu, An effective GPU implementation of breadth-first search, in: Proceedings of the 47th Design Automation Conference DAC'10, ACM, New York, NY, USA, 2010, pp. 52–55.

[16] K. Matsumoto, N. Nakasato, S. Sedukhin, Blocked all-pairs shortest paths algorithm for hybrid CPU–GPU system, in: High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on, 2011, pp. 145–152.

[17] A. McLaughlin, D.A. Bader, Revisiting edge and node parallelism for dynamic GPU graph analytics, in: 28th International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Multithreaded Architectures and Applications (MTAAP), 2014.

[18] E.L. Merrer, G. Trédan, Centralities: Capturing the fuzzy notion of importance in social graphs, in: Proceedings of the Second ACM EuroSys Workshop on Social Network Systems (SNS), 2009.

[19] D. Merrill, M. Garland, A. Grimshaw, Scalable GPU graph traversal, in: Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming PPoPP'12, ACM, New York, NY, USA, 2012, pp. 117–128.

[20] D. Mizell, K. Maschhoff, Early experiences with large-scale Cray XMT systems, in: 23rd International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Multithreaded Architectures and Applications (MTAAP), (May), 2009, pp. 1–9.

[21] T. Okuyama, F. Ino, K. Hagihara, A task parallel algorithm for finding all-pairs shortest paths using the GPU, Int. J. High Perform. Comput. Netw. 7 (2) (2012) 87–98.

[22] P. Pande, D.A. Bader, Computing betweenness centrality for small world networks on a GPU, in: 15th Annual High Performance Embedded Computing Workshop, HPEC, 2011.

[23] M.C. Pham, R. Klamma, The structure of the computer science knowledge network, in: Proceedings of International Conference on Advances in Social Networks Analysis and Mining, ASONAM, 2010.

[24] A.E. Sarıyüce, K. Kaya, E. Saule, Ü.V. Çatalyürek, Betweenness centrality on GPUs and heterogeneous architectures, in: Workshop on General Purpose Processing Using GPUs, GPGPU, in: Conjunction with ASPLOS March 2013.

[25] A.E. Sarıyüce, E. Saule, K. Kaya, Ü.V. Çatalyürek, Hardware/software vectorization for closeness centrality on multi-/many-core architectures, in: 28th International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum, IPDPSW, Workshop on Multithreaded Architectures and Applications (MTAAP), May 2014.

[26] A.E. Sarıyüce, E. Saule, K. Kaya, Ü.V. Çatalyürek, STREAMER: a distributed framework for incremental closeness centrality computation, in: Proc. of IEEE Cluster 2013, Sept. 2013.

[27] E. Saule, Ü.V. Çatalyürek, An early evaluation of the scalability of graph algorithms on the Intel MIC architecture, in: 26th International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Multithreaded Architectures and Applications, MTAAP, May 2012.

[28] E. Saule, K. Kaya, Ü.V. Çatalyürek, Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi, in: Proc. of the 10th Int'l Conf. on Parallel Processing and Applied Mathematics, PPAM, Sept. 2013.

[29] Z. Shi, B. Zhang, Fast network centrality analysis using GPUs, BMC Bioinformatics 12:149, 2011.

[30] J. Shun, G.E. Blelloch, Ligra: A lightweight graph processing framework for shared memory, in: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'13, 2013.

[31] Ö. Şimşek, A.G. Barto, Skill characterization based on betweenness, in: Proceedings of Neural Information Processing Systems, NIPS, 2008.

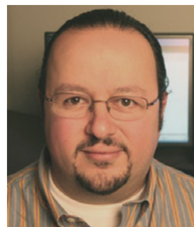
[32] R. Vuduc, J. Demmel, K. Yelick, OSKI A library of automatically tuned sparse matrix kernels, in: Proc. SciDAC 2005, J. of Physics: Conference Series (2005).

[33] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, Ü Çatalyürek, A scalable distributed parallel breadth-first search algorithm on BlueGene/L, in: Proceedings of SC2005 High Performance Computing, Networking, and Storage Conference, 2005.

Ahmet Erdem Sarıyüce is a Ph.D. candidate in the Department of Computer Science and Engineering at The Ohio State University. He received his B.S. in Computer Engineering from Middle East Technical University, Turkey, 2010. His research interests include graph mining, streaming graph algorithms and combinatorial scientific computing.

Erik Saule is an Assistant Professor in the Computer Science department of UNC Charlotte since August 2013. He received his License and Maitrise in Computer Science in 2003 from University of Versailles, France and his Master and Ph.D. in Computer Science, respectively in 2005 and 2008, from Grenoble Institute of technology, France. Dr. Saule has been a post-doctoral researcher in the Department of Biomedical Informatics at The Ohio State University from 2009 to 2013. His research interests revolve around the efficient use of modern computing platforms for compute intensive or data intensive applications.

Kamer Kaya is an Assistant Professor in the Department of Biomedical Informatics at The Ohio State University. He received his Ph.D. and Master in Computer Science, from Bilkent University, Turkey.



Ümit V. Çatalyürek is a Professor in the Depts. of Biomedical Informatics, Electrical and Computer Engineering, and Computer Science and Engineering. His research interests include combinatorial scientific computing, runtime systems for data-intensive computing, and high-performance computing in biomedicine. He received his Ph.D., M.S. and B.S. in Computer Engineering and Information Science from Bilkent University, Turkey, in 2000, 1994 and 1992, respectively.