2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Hardware/Software Vectorization for Closeness Centrality on Multi-/Many-Core Architectures

Ahmet Erdem Sarıyüce[1,2], Erik Saule[4], Kamer Kaya[1], Ümit V. Çatalyürek[1,3]
Depts. [1]Biomedical Informatics, [2]Computer Science and Engineering, [3]Electrical and Computer Engineering
The Ohio State University
Email:{*aerdem,kamer,umit*}*@bmi.osu.edu*
[4] Dept. Computer Science, University of North Carolina at Charlotte
Email: *esaule@uncc.edu*

***Abstract*—Centrality metrics have shown to be highly correlated with the importance and loads of the nodes in a network. Given the scale of today's social networks, it is essential to use efficient algorithms and high performance computing techniques for their fast computation. In this work, we exploit hardware and software vectorization in combination with fine-grain parallelization to compute the closeness centrality values. The proposed vectorization approach enables us to do concurrent breadth-first search operations and significantly increases the performance. We provide a comparison of different vectorization schemes and experimentally evaluate our contributions with respect to the existing parallel CPU-based solutions on cutting-edge hardware. Our implementations achieve to be 11 times faster than the state-of-the-art implementation for a graph with 234 million edges. The proposed techniques are beneficial to show how the vectorization can be efficiently utilized to execute other graph kernels that require multiple traversals over a large-scale network on cutting-edge architectures.**

***Keywords*-Centrality, closeness centrality, vectorization, breadth-first search, Intel Xeon Phi.**

I. Introduction

Many important graph analysis concepts, such as reachability and node influence [8], [11], [15], [18], [24], have a common fundamental building block: centrality computation. Closeness centrality (CC) is one of the widely-used centrality metrics. In spite of the existence of many application areas for CC, its efficiency remains a problem for today's large-scale social networks.

Given a graph with n vertices and m edges, the time complexity of the best sequential algorithm for CC on unweighted networks is $\mathcal{O}(nm)$. Weighted networks require more work with the complexity being $\mathcal{O}(nm + n^2 \log n)$. This complexity requirement makes the problem hard even for medium-scale networks. Existing studies focus on speeding up the centrality computation by algorithmic techniques such as graph compression [2], [20], by leveraging parallel processing techniques based on distributed- and shared-memory systems [9], [12], [14], GPUs [10], [19], [24], [17], or by a combination of these techniques [21].

Although many of the existing approaches 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 graph kernel computations. Graph computations are notorious for having irregular memory access pattern, and hence, for many kernels that require a single graph traversal, vectorization is usually 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. We conjecture that with continuous improvement of the vectorization capabilities of all recent CPUs, including undoubtedly the most common CPU architecture x86, algorithms that do not take advantage of this feature will not be able to fully utilize the CPU's computing power. Hence, in this study we investigate if vectorization can be leveraged for graph kernels that require multiple traversals; in particular, we use closeness centrality as our test case.

We propose algorithms that compute closeness centrality on different architectures using *hardware* and *software vectorization*. By leveraging vectorization, we perform multiple breadth-first search (BFS) computations at the same time, and hence, reduce the number of graph reads. Instead of using a primitive data type per vertex to store the BFS-related information, we use a single bit that yields automatic support for concurrent BFSs via bitwise operations. We will use the term *hardware vectorization* to describe the case where the number of BFSs is less than or equal to the vector register size of the architecture. Using vector operations enables $\mathcal{O}(1)$ updates for *visit* and *neighbor* data structures that are often used in BFS implementations to track and construct the current and the next frontier. One can continue to increase the number of BFSs beyond the register size and use multiple registers per vertex. Even though operations on those would require the successive use of hardware vector instructions, it could be still beneficial since it reduces the number of graph reads at the expense of using multiple vectors. We will call this approach *software vectorization*.

In order to take full advantage of vectorization, we first propose a sparse-matrix-vector multiplication (SpMV)-based formulation of closeness centrality, and from there, we move

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.156

to a sparse-matrix-matrix multiplication (SpMM)-based formulation. We present details of the manually vectorized implementation for x86 architectures, as well as a high-level C++ implementation that is suitable for automatic vectorization which is available in modern compilers. We evaluate the proposed algorithms on modern x86 architectures, in particular on a multi-core Intel Xeon CPU and on a many-core Intel Xeon Phi coprocessor. Results of the experiments on seven real-world networks show that using $8,192$ simultaneous BFSs with vectorization can achieve an improvement factor of 1.7 to 11.8 over the state-of-the-art techniques on CPU. Our comparison for manual and compiler-based vectorization shows that manual vectorization is only slightly better. Hence, if the code is written in a generic and smart way, the compiler does its job and optimizes relatively well.

The rest of the paper is organized as follows: Section II presents the notation we used in the paper, summarizes the existing parallel algorithms relative to closeness centrality and explains the recent come-back of SIMD architectures. The proposed techniques are formalized and described in Section III. Performance evaluation and comparison of the proposed solutions with the state-of-the-art are given in Section IV. Section V concludes the paper.

II. Background and Related Work

Let $G = (V, E)$ be a network modeled as a simple graph with $n = |V|$ vertices and $m = |E|$ edges where each node is represented by a vertex in V, and a node-node interaction is represented by an edge in E. Let $\Gamma_G(v)$ be the set of vertices which are interacting with v.

A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A *path* is a sequence of vertices such that there exists an edge between each consecutive vertex pair. Two vertices $u, v \in V$ are *connected* if there is a path between u and v. If all vertex pairs in G are connected we say that G is *connected*. Otherwise, it is *disconnected* and each maximal connected subgraph of G is a *connected component*, or a component, of G. We use $dst_G(u, v)$ to denote the length of the shortest path between two vertices u, v in a graph G. If $u = v$ then $dst_G(u, v) = 0$. If u and v are disconnected, then $dst_G(u, v) = \infty$.

A. Closeness centrality

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

$$\mathtt{cc}[u] = \sum_{\substack{v \in V \\ dst_G(u,v) \neq \infty}} \frac{1}{dst_G(u, v)}. \tag{1}$$

If u cannot reach any vertex in the graph $\mathtt{cc}[u] = 0$. An alternative formulation exists in the literature where summation is in the denominator and the centrality of u is equal to the reciprocal of the total distance from u to all other vertices. The proposed techniques would work for both, but for the sake of simplicity, we use the one above. Nevertheless, both require the shortest path distances between all the vertex pairs.

For a sparse unweighted graph $G = (V, E)$ the complexity of CC computation is $\mathcal{O}(n(m+n))$ [5]. The pseudo-code is given in Algorithm 1. 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 $\mathtt{cc}[s]$. If the graph is undirected, it could be accumulated to $\mathtt{cc}[w]$ instead 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.

Algorithm 1: CC: Basic centrality computation

```
   Data: G = (V, E)
   Output: cc[.]
1  cc[v] ← 0, ∀v ∈ V
2  for each s ∈ V do
3      Q ← empty queue
4      Q.push(s)
5      dst[s] ← 0
6      cc[s] ← 0
7      dst[v] ← ∞, ∀v ∈ V \ {s}
8      while Q is not empty do
9          v ← Q.pop()
10         for all w ∈ Γ_G(v) do
11             if dst[w] = ∞ then
12                 Q.push(w)
13                 dst[w] ← dst[v] + 1
14                 cc[w] ← cc[w] + 1/dst[w]
15 return cc[.]
```

B. Related work

In the literature, there are two ways to parallelize centrality computations: coarse- and fine-grain. In coarse-grain parallelism, multiple BFSs are partitioned among the processors/threads such that a shortest-path graph is constructed by a single processor/thread (as in Algorithm 1). On the other hand, fine-grain parallelism executes a single BFS concurrently by multiple processors/threads. The memory requirement of fine-grain parallelism is less than that of coarse-grain parallelism. Therefore, fine-grain parallelism is more suitable for devices with restricted memory, such as GPUs.

Some examples of centrality computation using fine-grain parallelism include [10], [19], [24]; Shi and Zhang [24] developed a software package for parallel betweenness centrality computation on GPUs to be used in biological network analysis. Jia et al. [10] experimented various parallelism techniques on GPUs for betweenness and closeness centrality computation. Recently, Sarıyüce et al. [19] used a modified graph storage scheme to obtain better speedups compared to existing solutions. All these works 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-grained and coarse-grained parallelism is [16]. In their work, the authors optimize the betweenness centrality computation on Cray XMT computers. 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 [22]. 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 [23].

C. SIMD-based architecture

Single instruction, multiple data (SIMD) is an important class of parallel computing paradigms in Flynn's taxonomy. It relies on leveraging the data parallelism where there are simultaneous computations on multiple data points with a single process at any moment. Modern CPU designs make use of SIMD paradigm, especially for multimedia use.

The first era of modern SIMD machines were represented by massively parallel-processing-style supercomputers. There were many limited-functionality processors that would work in parallel in those machines. When *multiple instruction, multiple data* (MIMD) systems that are based on commodity processors became more prevalent, interest in SIMD has declined. The current era of SIMD processors has emerged out of the desktop-computer market rather than the supercomputer market. As desktop processors started to support real-time gaming and multimedia processing, requirement for processing power has increased and vendors resurrected the SIMD processors to meet the demand. Intel's MMX extensions to the x86 architecture was the first widely-deployed desktop SIMD. Then, Intel introduced the all-new SSE system in 1999 which uses new 128 bit registers for vectorial operations. Since then, there has been many extensions to the SSE system, such as SSE2, SSE3, SSSE3, and SSE4. In 2008, Intel announced the AVX system and introduced 256 bit registers and recently, 512 bit registers are incorporated in the new Xeon Phi series, which use the *many integrated core* (MIC) architecture. A proper usage of these vectorial operations is as important as a proper usage of the multiple cores of a processor since they can improve 32-bit computation by a factor of 8 in AVX and by a factor of 16 in MIC.

III. A Vector-friendly Approach to Closeness Centrality

A. Regularity in Linear Algebra: From SpMV to SpMM

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], [25] or by using different matrix storage formats [4], [13] have been proposed.

Yet the most efficient method to regularize the memory access pattern is to multiply a sparse matrix by multiple vectors. 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 non-zero 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 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 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.

B. 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 at level ℓ if and only if one of the neighbors of v is at level $\ell - 1$ and v is not at any level $\ell' < \ell$. This formulation is commonly used in parallel BFS implementations on GPU [10], [17], [24] and also in shared memory [1] and distributed memory settings [6].

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

$$y_k^{\ell+1} = OR_{j \in \Gamma(k)} x_j^\ell.$$

That is $y^{\ell+1}$ is the result of the multiplication of the adjacency matrix of the graph by x^ℓ in the (OR,AND) semiring. The next level is then computed with

$$x_i^{\ell+1} = y_i^{\ell+1} \& \neg (OR_{\ell' \leq \ell} x_i^{\ell'}).$$

Using these variables, one can update the closeness centrality value of a vertex i by adding $\frac{x_i^\ell}{\ell}$ for each level ℓ.

Implementing BFS using such an SpMV-based algorithm changes its asymptotic complexity. The traditional queue-based BFS algorithm has a complexity of $\mathcal{O}(m+n)$. On the other hand, the complexity of the SpMV-based algorithm depends on how the adjacency matrix is stored. If a column-wise storage is used it is easy to traverse column j when the value of x_j^ℓ is **true**. This again leads to an $\mathcal{O}(m+n)$ BFS implementation which is not essentially different from the queue-based implementation of a single BFS: they both follow a *top-down* approach. When x_j^ℓ is **true**, the updates on the $y^{\ell+1}$ vector are scattered in memory, which would be problematic when executed in parallel.

By storing the adjacency matrix row-wise, different values of x^ℓ are gathered to compute a single element of $y^{\ell+1}$. This produces a *bottom-up* BFS implementation which has more natural write access patterns. However, it becomes impossible to only traverse the relevant non-zeros of the matrix and the complexity of the algorithm becomes $\mathcal{O}(|E| \times d)$ where d is the diameter of the graph. This is the implementation we favor and we believe that the asymptotically worse complexity is not a huge bottleneck for an efficient computation since it has been noted many times before that social networks have small world properties. So, their diameters are usually low.

C. 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 2). The algorithm processes the BFS sources by batches of b. For each level ℓ, it builds a binary matrix x^ℓ where $x^\ell_{i,s\%b}$ indicates if vertex i is at distance ℓ of source vertex s for the $(s\%b)$th BFS being executed. 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 $\ell+1$.

When b is set to 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. Figure 1 presents an implementation of this algorithm using AVX instructions (b = 256). We use similar codes to leverage 32-bit integer types, SSE registers and MIC'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 $OR_{l' \leq \ell} x^{\ell'}$. The function `bitCount_256(.)` calls the appropriate bit-counting instructions.

Algorithm 2: CC-SpMM: SpMM-based centrality computation

Data: $G = (V, E)$, b
Output: `cc`[.]
▷`Init`
`cc`$[v] \leftarrow 0, \forall v \in V$
$\ell \leftarrow 0$
partition V into k batches $\Pi = \{V_1, V_2, \ldots, V_k\}$ of size b
for each batch *of vertices* $V_p \in \Pi$ **do**
  $x^0_{s,s} \leftarrow 1$ if $s \in V_p$, 0 otherwise
  while $\sum_i \sum_s x^\ell_{i,s} > 0$ **do**
    ▷`SpMM`
    $y^{\ell+1}_{i,s} = OR_{j \in \Gamma(i)} x^\ell_{j,s}, \forall s \in V_p, \forall i \in V$
    ▷`Update`
    $x^{\ell+1}_{i,s} = y^{\ell+1}_{i,s} \& \neg(OR_{\ell' \leq \ell} x^{\ell'}_{i,s}), \forall s \in V_p, \forall i \in V$
    $\ell \leftarrow \ell + 1$
    for all $v \in V$ **do**
      `cc`$[v] \leftarrow$ `cc`$[v] + \frac{\sum_s x^\ell_{v,s}}{\ell}$
return `cc`[.]

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.

Figure 2 presents the cache-hit ratios with a cache size of 512K (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% to 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.

D. 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 BFSs, the implementation effort for manual vectorization

```
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
#pragma omp parallel for schedule (dynamic, CC_CHUNK)
    for (int i = 0; i < n; ++i) {
      __m256i neigh = _mm256_setzero_si256();
      int il[8] = {0, 0, 0, 0, 0, 0, 0, 0};
      if (i >= s && i < s + b)
        il[(i-s)>>5] = 1 << ((i-s) & 0x1F);
      __m256i cu = _mm256_set_epi32(il[7], il[6], il[5], il[4],
                                    il[3], il[2], il[1], il[0]);
      _mm256_store_si256 ((__m256i *)(neighbor + 32 * i), neigh);
      _mm256_store_si256 ((__m256i *)(current + 32 * i), cu);
      _mm256_store_si256 ((__m256i *)(visited + 32 * i), cu);
    }
    int cont = 1;
    int 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);
}
```

Figure 1. Hardware vectorization using AVX for the SpMM-based formulation of closeness centrality.

of each version becomes prohibitive. Therefore, we developed a unique code that allows to change the number of sources concurrently traversed. Figure 3 presents 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

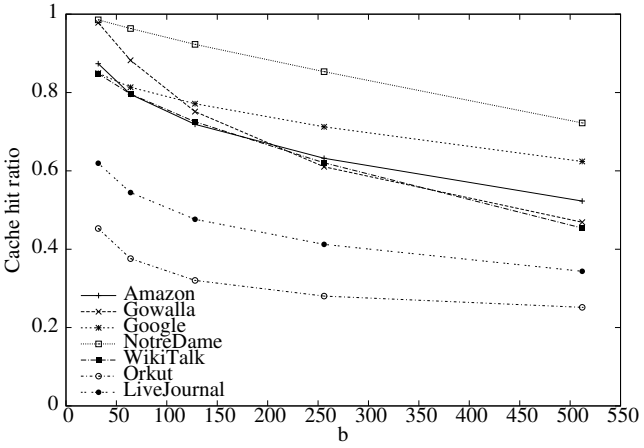


Figure 2. Simulated cache-hit ratio of the SpMM variant on a 512K cache (e.g., Intel Xeon Phi's L2 cache).

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.

IV. Experiments

The experiments are carried out on a system, equipped with two Intel Sandybridge-EP CPUs clocked at 2.00Ghz and 256GB 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 32kB L1 cache and 256kB L2 cache. The 8 cores on a CPU share a 20MB L3 cache. The CPUs support Streaming SIMD Extensions (SSE) instruction set which originally added eight new 128-bit registers and Advanced Vector Extension (AVX) which provides an enhanced 256-bit instruction set with wider vectors, newer syntax, and functionality.

The system also has an Intel Xeon Phi coprocessor with 8 memory controllers and 61 cores clocked at 1.05GHz. There is a 32kB L1 data cache, a 32kB L1 instruction cache, and a 512kB L2 cache associated with each core. The bandwidth of each core is 8.4GB/s per where the cores' memory interface are 32-bit wide with two channels. Although the cores are expected to provide 512.4GB/s, the bandwidth between the memory controllers and cores is limited by the

```
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;
  size_t size_alloc = n;
  size_alloc *= b / 8;
  int n_align = 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
#pragma omp parallel for schedule (dynamic, CC_CHUNK)
    for (int i = 0; i < n; ++i) {
      int cu[vector_size];
#pragma unroll
      for (int j = 0; j < vector_size; j++)
        cu[j] = 0;
      if (i >= s && i < s + b)
        cu[(i-s)>>5] = 1 << ((i-s)&0x1F);
#pragma unroll
      for (int k=0; k< vector_size; ++k)
        ((int*)current)[i*vector_size+k] = cu[k];
#pragma unroll
      for (int k=0; k< vector_size; ++k)
        ((int*)visited)[i*vector_size+k] = cu[k];
    }
    int cont = 1;
    int 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
      float flevel = 1.0f / (float)level;
#pragma omp parallel for schedule (dynamic,CC_CHUNK)
      for (int i = 0; i < n; ++i) {
        int cu[vector_size];
#pragma unroll
        for (int k = 0; k < vector_size; k++)
          cu[k] = ((int*)neighbor)[i*vector_size+k]
            & ~((int*)visited)[i*vector_size+k];
#pragma unroll
        for (int k = 0; k < vector_size; k++)
          ((int*)visited)[i*vector_size+k] = cu[k]
            | ((int*)visited)[i*vector_size+k];
        int bcount = 0;
#pragma unroll
        for (int k = 0; k < vector_size; k++)
          bcount += BitCount32(cu[k]);
        if (bcount > 0) {
          cc[i] += bcount * flevel;
          cont = 1;
        }
#pragma unroll
        for (int k = 0; k < vector_size; ++k)
          ((int*)current)[i*vector_size+k] = cu[k];
      }
    }
  }
  _mm_free(neighbor);
  _mm_free(current);
  _mm_free(visited);
}
```

Figure 3. Compiler vectorization for the SpMM-based formulation of closeness centrality.

Table I
PROPERTIES OF THE LARGEST CONNECTED COMPONENTS OF THE GRAPHS USED IN THE EXPERIMENTS. DIAMETER IS THE LENGTH OF THE LONGEST PATH IN THE GRAPH.

Graph	$\|V\|$	$\|E\|$	Avg. $\|adj(v)\|$	Max. $\|adj(v)\|$	Diam.
Amazon	403K	4,886K	12.1	2,752	19
Gowalla	196K	1,900K	9.6	14,730	12
Google	855K	8,582K	10.0	6,332	18
NotreDame	325K	2,180K	6.6	10,721	27
WikiTalk	2,388K	9,313K	3.8	100,029	10
Orkut	3,072K	234,370K	76.2	33,313	9
LiveJournal	4,843K	85,691K	17.6	20,333	15

ring network in between which theoretically supports at most 220GB/s.

The true potential of Xeon Phi lies in its vector processing unit. There are 32×512-bit vector registers on each of Intel Xeon Phi's cores, which can be used for integers or floating point number (in double or single precision). They can be used either as a vector of 8×64-bit values or as a vector of 16×32-bit values, respectively. Many instructions, such as addition or division, and mathematical operations can be performed in vector processing unit. It allows to perform 8 basic operations, such as addition or multiplication, per cycle in double precision (16 in single precision). A fused multiply add unit allows to double that rate for application that can leverage it.

On the software side, our system runs a 64-bit Debian with Linux 2.6.39-bpo.2-amd64. All the codes are compiled with icc version 13.1.3 with the -O3 optimization flag. We have carefully implemented all the algorithms using C++. To have a base-line comparison, we implemented OpenMP versions of the CPU-based closeness centrality codes. Algorithm 1 consecutively uses the vertices with distance k to find the vertices with distance $k+1$. Hence, it visits the vertices in a *top-down* manner. A BFS can also be performed in a *bottom-up* manner, i.e., after all distance-k vertices are found, the vertices with unknown distances are processed to see if they have a neighbor at level k. The top-down variant is expected to be much cheaper for small k values. However, it can be more expensive for the lower levels where there are much less unprocessed vertices remaining. For the baselines without vectorization, we follow the idea of Beamer et al. [3] and use a hybrid (top-down/bottom-up) BFS. That is while processing the nodes at a BFS level, we simply compare the number of edges need to be processed in the frontier to the number of edges adjacent to unvisited vertices in order to choose the cheaper variant. Other than *direction optimization* (DO) [3], no particular optimization have been applied to the OpenMP-based parallel CPU codes except the ones performed by the compiler.

To evaluate the algorithms, we used seven graphs from

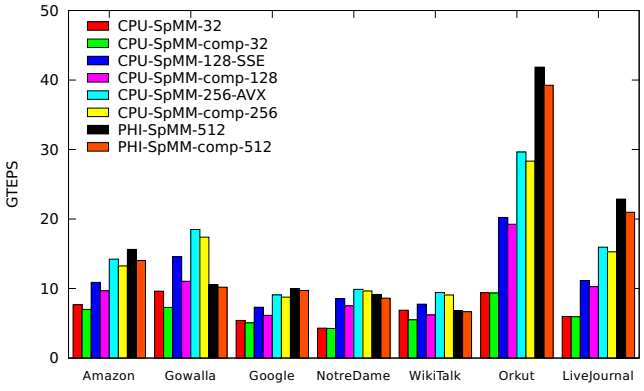


Figure 4. Manual vs. compiler-based hardware vectorization: the manually vectorized codes with hardware intrinsics such as SSE and AVX are only slightly better than the compiler-vectorized ones.

the SNAP dataset[1]. Directed graphs were made undirected and the largest connected component was extracted. The list of graphs and the properties of the largest components that are used in our experiments can be found in Table I.

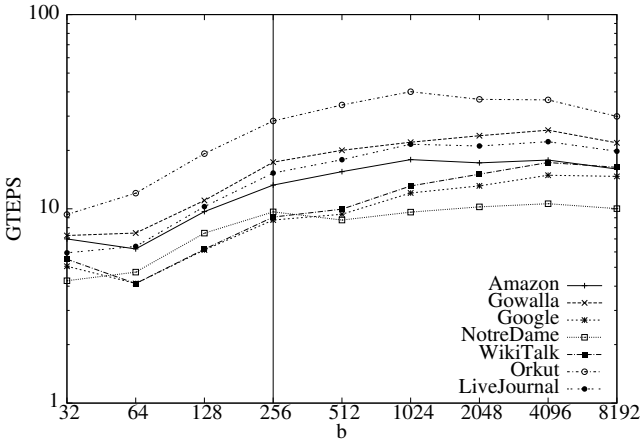
A. Manual vs. compiler-based hardware vectorization

As a first experiment, we compare manual and compiler-based hardware vectorization options to achieve better centrality performance. For manual vectorization, we implemented 32-bit, 128-bit SSE, 256-bit AVX (see Figure 1), 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 given in Figure 3 and let the compiler optimize it with -O3 flag. Figure 4 gives the performance results via a bar-chart in terms of billions of traversed edges per second (GTEPS). The bars in the figure with `-comp` keyword are the ones with the compiler-vectorized versions. Almost for 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 written in a generic and smart way, the compiler does its job and optimizes relatively well.

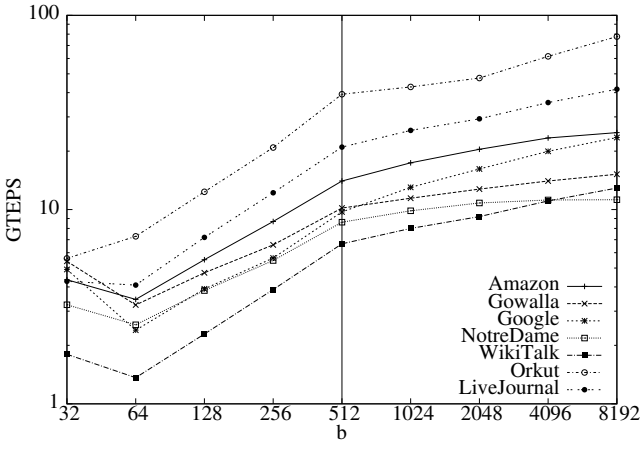
B. Impact of the number of BFSs on the performance

Motivated by Figure 4, we evaluate the performance of compiler-based hardware and software vectorization in terms of the number of concurrent BFSs on CPU and Intel Xeon Phi coprocessor. Figure 5 presents the results. Note that the registers in the CPU are 256 bits whereas the ones in Intel Xeon Phi are 512 bits (these values are marked with vertical lines in the figure). For both architectures, the acceleration of the performance decreases after those vertical lines, and software vectorization comes into play since the register sizes are reached. Hence, we can argue that as expected, hardware vectorization is much more effective to improve the performance. However, its use is limited by the register sizes. Besides, the performance usually increases even with software vectorization. On Intel Xeon Phi, such improvement is more visible.

For the CPU, there is no performance improvement on any of the graphs when the number of BFSs is increased from $4,096$ to $8,192$. This is expected since there are only 16 256-bit registers per core that can support only up to $4,096$ BFSs. For Xeon Phi, the number of 512-bit registers per core is 32 which can support up to $16,384$ BFSs. All of these conclude that the accelerator can make use of the vectorization in a much better way. As the figures show, on `Orkut`, Intel Xeon Phi reaches 77 GTEPS for the closeness centrality computation whereas the CPU can reach only 36.3 GTEPS.



(a) CPU



(b) PHI

Figure 5. Performance of the compiler-based hardware (left of the vertical lines) and software (right of the vertical lines) vectorization on CPU and Intel Xeon Phi.

An interesting observation from Figure 5 is that the

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

vectorized implementation shows a better performance and obtain better speedups on particular graphs. For example, the code shows its best performance on `Orkut` for both CPU and Intel Xeon Phi architectures. We have multiple reasonings for these performance differences as there are various parameters, e.g., graph structure and cache usage, that affect the performance. For example, increasing the number of BFSs also increases the memory footprint that can damage the cache utilization and be a bottleneck while improving the performance.

In a traditional BFS code, the graph is loaded from memory once a single BFS is executed. On the other hand, in the vectorized code, after a sufficient number of BFSs, the graph is loaded at most d times, one per each BFS level, where d is the diameter of the graph. In our experiments, all the graphs are generated from real-world social networks which already tend to have small diameters. However, `Orkut` has the smallest one among the seven we used.

After a sufficient number of BFSs, i.e., the case where each vertex appears at each BFS level for at least one BFS, the number of BFSs does not effect the number of graph loads. Hence, by using k times more concurrent BFSs, we reduce the overall number of graph loads k times. The impact of this gain on the overall performance depends on the percentage of graph-loads and edge-traversals on the overall execution time. The *density* of the graph, which is the average number of neighbors per vertex, is a logical metric to evaluate this impact since the rest of the operations, e.g., bit counting, usually depends the number of vertices. Indeed, comparing the graph densities in Table I and performances in Figure 5, it is easy to see that the performance improvements are positively correlated with graph densities. That is for denser graphs such as `Orkut`, `LiveJournal`, and `Amazon`, the performance is better.

For practical evidence, we break the execution of the compiler-based vectorization with 512 BFSs into three parts; initialization, SpMM, and update. We show the individual execution times of these parts on Intel Xeon Phi in Figure 6. As explained above, when the number of BFSs is increased, only the SpMM part is accelerated. Therefore, it is expected that if the proportion of SpMM is high in the overall execution then the performance improvement is more when the number of BFSs is increased. For `Orkut` in the figure, the percentage of the SpMM is higher compared to other graphs. Therefore, its performance continues to increase when the number of BFSs is increased.

Another reason of the performance differences is a simple observation from Figure 2. Recalling that `Orkut` already has a bad cache-hit ratio with 32 BFSs and it does not get worse after 256 BFSs, it is expected that the cache-hit ratio will not be a bottleneck on the performance improvement for `Orkut`. On the other hand, for `NotreDame`, the cache-hit ratio is really good with 32 BFSs and it gets significantly worse as the number of BFSs increases. This behavior negatively affects the performance improvement trend for `NotreDame` as shown in Figure 5.

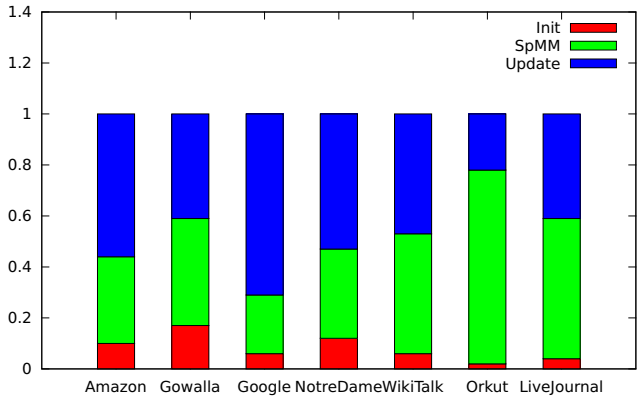


Figure 6. The Init, SpMM, and Update times (in proportion) on Intel Xeon Phi with 512 concurrent BFS for the seven graphs used in the experiments.

C. Comparing the vectorized code with non-vectorized one

We compare the proposed compiler-based hardware and software vectorized implementation (denoted with `-comp`) with the state-of-the-art non-vectorized parallel implementation in terms of the traversed giga-edges per second (GTEPS). Figure 7 presents the results of the comparison with the direction-optimized CPU code (denoted with `-DO`). The results are presented for the CPU and Intel Xeon Phi architectures separately. For software vectorization, $4,096$ and $8,192$ BFSs are used for CPU and Intel Xeon Phi, respectively. For all graphs and both architecture, the hardware vectorized variants perform better than the direction optimized baseline, and the software vectorized variants performs better than the hardware vectorized one. For example, on `Orkut`, the proposed approach with $8,192$ BFSs on Intel Xeon Phi is 11.08 times faster than parallel state-of-the-art technique on CPU. Table II presents the relative improvement achieved on both CPU and Phi using software vectorization for all the graphs compared to the state-of-the-art `CPU-DO`. The improvement on the CPU ranges from 1.67 to 6.29 times faster with an average of 3.71 times faster. On Intel Xeon Phi, it ranges from 1.71 to 11.82 times faster and averages at 4.55 times faster.

V. Conclusion and Future Work

In this work, we proposed hardware and software vectorization for closeness centrality computation. We utilized the given architectures efficiently by applying multiple BFS operations at the same time. Comparison of different vectorization schemes are presented and experimentally evaluated on cutting-edge hardware Intel x86 many-core and multi-core architectures. For vectorizing the code, we showed that one does not need to dwell into low-level hardware intrinsics, instead one can implement the code at a high-level in C++,

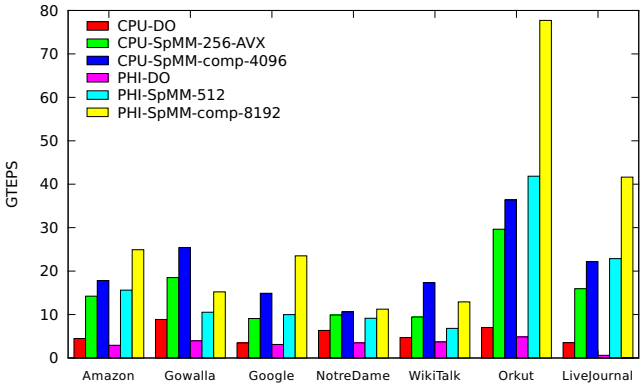


Figure 7. Comparison of the proposed vectorized implementation with the directed optimized implementation (aka. -DO).

Graph	DO	CPU SpMM-4096	PHI SpMM-8192
Amazon	4.4	17.8 (3.97x)	24.9 (5.55x)
Gowalla	8.8	25.4 (2.86x)	15.2 (1.71x)
Google	3.4	14.8 (4.30x)	23.5 (6.79x)
NotreDame	6.3	10.6 (1.67x)	11.2 (1.77x)
WikiTalk	4.7	17.3 (3.68x)	12.9 (2.74x)
Orkut	7.0	36.3 (5.19x)	77.7 (11.08x)
LiveJournal	3.5	22.1 (6.29x)	41.6 (1.82x)
Geo. Mean		(3.71x)	(4.55x)

Table II
PERFORMANCE OF THE STATE-OF-THE-ART ALGORITHM `CPU-DO` AND OF THE PROPOSED TECHNIQUE ON CPU (`CPU-SpMM-comp-4096`) AND ON XEON PHI (`PHI-SpMM-comp-8192`) USING SOFTWARE VECTORIZATION. PERFORMANCE IS EXPRESSED IN GTEPS, THE RATIO TO `CPU-DO` IS GIVEN BETWEEN THE PARENTHESIS.

and modern compilers can automatically vectorize the code, albeit for a very small performance deficit. Furthermore, we showed that using software vectorization, on top of hardware vectorization, we can further improve the performance. With all the techniques we proposed in this study, we achieve performance up to 11 times faster than the state-of-the-art techniques on CPU.

As a future work, we would like to extend our studies to other modern architectures and investigate an analytical model to chose the best vector size for a given input graph and architecture properties.

We believe that our proposed techniques can be applied for other graph kernels as well as other irregular computations that requires multiple traversals. Use of vectorization techniques, such as the ones proposed in this work, are inevitable to achieve good performance on the modern multi- and many-core architectures with the wide vector units.

ACKNOWLEDGMENT

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

REFERENCES

[1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *SuperComputing*, pages 1–11, 2010.

[2] M. Baglioni, F. Geraci, M. Pellegrini, and E. Lastres. Fast exact computation of betweenness centrality in social networks. In *Proceedings of International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2012.

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

[4] M. Belgin, G. Back, and C. J. Ribbens. Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In *Proc. of ICS*, pages 100–109, 2009.

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

[6] A. Buluç and J. R. Gilbert. The combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications (IJHPCA)*, 2011.

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

[8] Ö. Şimşek and A. G. Barto. Skill characterization based on betweenness. In *Proceedings of Neural Information Processing Systems (NIPS)*, 2008.

[9] M. Frasca, K. Madduri, and P. Raghavan. NUMA-aware graph mining techniques for performance and energy efficiency. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, 2012.

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

[11] S. Jin, Z. Huang, Y. Chen, D. G. Chavarría-Miranda, J. Feo, and 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)*, pages 1–7, 2010.

[12] R. Lichtenwalter and N. V. Chawla. DisNet: A framework for distributed graph computation. In *Proceedings of International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2011.

[13] X. Liu, M. Smelyanskiy, E. Chow, and 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, pages 273–282, New York, NY, USA, 2013. ACM.

[14] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. G. Chavarría-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *23rd International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Multithreaded Architectures and Applications (MTAAP)*, 2009.

[15] E. L. Merrer and 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.

[16] D. Mizell and 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)*, pages 1–9, May 2009.

[17] P. Pande and D. A. Bader. Computing betweenness centrality for small world networks on a GPU. In *15th Annual High Performance Embedded Computing Workshop (HPEC)*, 2011.

[18] M. C. Pham and 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.

[19] A. E. Sarıyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek. Betweenness centrality on GPUs and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, 2013.

[20] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. Shattering and compressing networks for betweenness centrality. In *SIAM Data Mining Conference (SDM)*, 2013.

[21] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. STREAMER: a distributed framework for incremental closeness centrality computation. In *Proc. of IEEE Cluster 2013*, Sep 2013.

[22] E. Saule and 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.

[23] E. Saule, K. Kaya, and 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)*, Sep 2013.

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

[25] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC 2005, J. of Physics: Conference Series*, 2005.