Research Article

Impact of AVX-512 Instructions on Graph Partitioning Problems

Md Maruf Hossain and Erik Saule

Computer Science, University of North Carolina at Charlotte, 9201 University City Blvd, Charlotte, 28213, NC, USA.

Contributing authors: mhossa10@uncc.edu; esaule@uncc.edu;

Abstract

Graph analysis now percolates society with applications ranging from advertising and transportation to medical research. The structure of graphs is becoming more complex every day while they are getting larger. The increasing size of graph networks has made many of the classical algorithms reasonably slow. Fortunately, CPU architectures have evolved to adjust to new and more complex problems in terms of core-level parallelism and vector-level parallelism (SIMD-level). In this paper, we are exploring how the modern vector architecture of CPUs can help with community detection, partitioning, and coloring kernels by studying three representatives algorithms. We consider the Intel SkylakeX and Cascade Lake architectures, which support gather and scatter instructions on 512-bit vectors. The existing vectorized graph algorithms of classic graph problems, such as BFS and PageRank, do not apply well to community detection; we show the support of gather and scatter are necessary. In particular for the implementation of the reduce-scatter patterns. We evaluate the performances achieved on the two architectures and conclude that good hardware support for scatter instructions is necessary to fully leverage the vector processing for graph partitioning problems.

Keywords: Graph Coloring, Community Detection, Vector Processing, AVX-512, Cascade Lake, SkylakeX, reduce-scatter.

1 Introduction

Graphs are at the center of most modern applications today: city and road analysis [1], social media analysis [2–4], biological data processing and medical research [4, 5], academic networks [6], intelligence [7]. And with the advent of the big data era, graph size has grown exponentially in recent years. We are particularly interested here in partitioning algorithms at large: coloring [8–10], clustering [11], partitioning [12], community detection [2, 13]. Recent interest in fast graph algorithms has met with a new look at how computer architectures can leverage. GPUs have been understandably popular because of the high flop rate, high memory bandwidth, and high power efficiency for graph problems [14]. CPU architectures have reacted by increasing core count but also by increasing SIMD width in a move to catch up in terms of performance and energy efficiency. In particular, modern Intel processors support AVX-512. These SIMD operations bring the expectation to provide higher energy efficiency than increasing the number of cores.

In this paper, we consider the use of these new instructions to solve graph problems in the class of partitioning. We pick three graph partitioning algorithms, namely a speculative parallel greedy algorithm for graph coloring, and the Louvain method for modularity optimization and Clustering using Label Propagation, as representative of graph partitioning algorithms. Section 3 describes these three problems. And we will study their performance on two different processors architecture; Intel Cascade Lake and SkylakeX.

With support for scatter operations, we designed, in Section 4, a strategy called ONPL, for One Neighbor Per Lane. Scatter operations enable us to write to the color of groups of neighbors at once. The operation in the Louvain Method adds some affinity values to the neighboring communities. Because the same community may appear multiple times, we call this operation a reduce-scatter, and we provide two implementations of this operation for different use cases.

Then, we show that the vectorization of these algorithms on x86-64 processors is impractical if they do not support scatter operations. Indeed the only feasible strategy in such a case is to use the different lanes of the vector to process different vertices at the same time. While this strategy applies to classic problems like BFS or SpMV, it requires reordering the graph so that no two vertices in a block of 16 vertices are neighbors for partitioning problems. This strategy only makes sense for the Louvain Method. The derived algorithm, presented in Section 5, is OVPL for One Vertex Per Lane.

A Part of this work [15] is published in scientific workshop. We extended our work by analysis performance of the Louvain method and Label propagation on the *RMAT* synthesis graph in section 7.3. It brings the perspective of what kind of graph structure well suited for the community detection problems. Section 6 presents the experimental settings, the code base used as baselines, and the set of graphs to be analyzed. Section 7.1 presents experimental results which show that ONPL can outperform the scalar implementation for graph coloring and label propagation for some networks. The Louvain Method is more computationally expensive. And using *ONPL* and *OVPL* in NetworKit leads to performance improvement on both architectures.

2 Notations

A graph is denoted by $G = (V, E)$ where V and E represent the vertex and edge set respectively. Edges are represented by (u, v) pair and are associated with an edge weight $\omega : E \rightarrow \mathbb{R}^+$. We use ζ to represent the community set and communities are represented by distinct integers. We use $N(u)$ to represent the neighbor set of a vertex $u \in V$. The *volume* of a node and a community are defined as $vol(u) = \sum_{\{u,v\}:v \in N(u)} \omega(u, v) + 2 \times \omega(u, u)$ and $vol(\zeta) = \sum_{u \in \zeta} vol(u)$ respectively.

3 Graph Partitioning Problems

Graph partitioning problems are seen here as a large class of graph algorithms that encompass graph coloring algorithms [8–10], partitioning to minimize edge cuts [12], modularity optimizing community detection algorithms [2, 13], overlapping community detection algorithms [16], label propagation, and certainly many others. All these algorithms have a similar structure in that each vertex is associated with a group of vertices (or multiple groups), and when considering the neighbors of a vertex, the group the neighbor belongs to is the key information rather than the neighbor itself.

We picked three classical partitioning algorithms to represent this class, namely Greedy Graph Coloring (for graph coloring), Louvain Method and Label Propagation (for non-overlapping community detection).

3.1 Speculative Parallel Greedy Graph Coloring

The distance-1 graph coloring algorithm assigns colors to the vertices of the graph so that no adjacent vertices have the same color. Minimize the number of colors is an NP-hard problem [17], and that is why various heuristic algorithms have proposed for the problem. In particular, a greedy algorithm can obtain near-optimal solutions [9]. The classic parallel algorithm for graph coloring is a speculative parallel greedy algorithm [18, 19] and presented in Algorithms 1, 2, 3.

Algorithm 1 Iterative Parallel Graph Coloring

Input: $G = (V, E)$

1: $C(v) \leftarrow 0$, for all $v \in V$
2: CONF $\leftarrow V$
3: **while** CONF $\neq 0$ **do**
4: AssignColors(G, C, CONF)
5: CONF $\leftarrow$ DetectConflicts(G, C, CONF)
6: **end while**
7: **return** C

Algorithm 2 AssignColors

Input: $G = (V, E), C$, CONF

1: Allocate private FORBIDDEN with size max degree
2: **for** $v \in$ CONF in parallel **do**
3: FORBIDDEN $\leftarrow false$
4: FORBIDDEN($C(u)$) $\leftarrow true$ for $u \in adj(v)$
5: $C(v) \leftarrow \min\{i > 0 \mid \text{FORBIDDEN}(i) = false\}$
6: **end for**
7: **return** C

Algorithm 3 DetectConflicts

Input: $G = (V, E), C$, CONF

1: NEWCONF $\leftarrow 0$
2: **for** $v \in$CONF in parallel **do**
3: **for** $u \in adj(v)$ **do**
4: **if** $C(u) = C(v)$ and $u < v$ **then**
5: ATOMIC NEWCONF $\leftarrow$ NEWCONF $\cup\, v$
6: **end if**
7: **end for**
8: **end for**
9: **return** NEWCONF

Algorithm 1 represents an iterative parallel graph coloring. It takes a graph G with vertex set V and edge set E as an input. It first initializes the set of colors C for all vertices by 0 and a set of conflicts **CONF** by all vertices. It will iteratively color the vertices in **CONF** using a speculative greedy algorithm. And then check whether two neighboring vertices use the same color in which case they are in conflict and need to be colored again.
Algorithm 2 is the algorithm that will be vectorized and handles the assignment of the color to vertices. It takes graph G, a set of color C and a set of conflicts **CONF** as input. It traverses all the conflict vertices and finds out all the forbidden colors **FORBIDDEN** for the particular vertex. To do that it iterates all its neighbors and track down their colors. Line 4 of Algorithm 2 represents this operation. After collecting all the forbidden colors, it assigns to the vertex the first color that is not in the **FORBIDDEN** set.
Algorithm 3 detects conflicts that could arise during parallel speculative coloring. It takes a graph G, a set of color C, and a previous conflict set **CONF** as input. It defines a new empty conflict set **NEWCONF**. It considers all the previous conflict set of vertices in parallel and for each visits the neighbors to detect if the edge has both ends with the same color. In that case, one of the two vertices is added to the new conflict set atomically.

3.2 Parallel Louvain Method

The *modularity* is defined as the fraction of edges that fall within the partitions minus the expected fraction that would be within the partition if the edges are distributed randomly. This definition enables to greedily optimize modularity by considering moving a vertex to one of its neighbor community. Indeed, if a node $u \in C$ moves to the neighboring community D, then the modularity gain is $\Delta mod(u, C \rightarrow D) = \frac{\omega(u,D/\ \{u\}) - \omega(u,C/\ \{u\})}{\omega(E)} + \frac{(vol(C/\ \{u\}) - vol(D/\ \{u\})) * vol(u)}{2*\omega(E)^2}$

The Louvain Method, first proposed by Blondel *et al.* [13], is one of the most popular methods to extract communities from a large network. It is a greedy multilevel algorithm that uses modularity as the objective function [20]. It alternates between two phases, the *Move Phase*, and the *Coarsening Phase*. In the *Move Phase*, nodes are repeatedly moved to adjacent communities to maximize modularity. This process repeats until the communities are stable. Then, the graph goes through a *Coarsening Phase* where each community collapse into a single vertex. The coarsened graph is then recursively processed with the same two phases. In that sense, the Louvain Method is representative of multi-level partitioning algorithms, such as [12].

The *Move Phase* (Algorithm 4) considers all the vertices in the network. For each vertex $u \in V$,

Algorithm 4 Louvain Method: Move Phase

Input: graph $G = (V, E, \omega)$, communities $\zeta : V \to \mathbb{N}$
Result: communities $\zeta : V \to \mathbb{N}$

1: **repeat**
2: **for** $u \in V$ **do**
3: $\delta \leftarrow \max_{v \in N(u)}\{\Delta mod(u, \zeta(u) \to \zeta(v))\}$
4: **if** $\delta > 0$ **then**
5: $C \leftarrow \zeta(\arg\max_{v \in N(u)}\{\Delta mod(u, \zeta(u) \to \zeta(v))\})$
6: $\zeta(u) \leftarrow C$
7: **end if**
8: **end for**
9: **until** ζ stable
10: **return** ζ

for each neighbor $v \in N(u)$, it calculates the modularity difference between having u in its current community and moving it to the community of v. The decision of highest modularity gain is retained, and it is enacted if the modularity gain δ is positive. The algorithm repeats until no vertex changes community.

For each vertex u, the *move phase* is split into two parts. First, calculate the affinity(*measures of similarity between pairs of vertices*) of each neighboring community $\zeta(v)$ by adding edge weight $\omega(u, v)$ of the each neighbor v of u. Second, assign the node to the community of highest affinity.

The affinity calculation of a vertex is the computationally expensive part of the algorithm. It is the part that we vectorize in this paper. We do not describe the *Coarsening Phase* since we will not make any changes to it. In this work, we only investigate the performances of the *Move Phase* of the *Louvain* method.

Many parallel methods exist to detect communities in massive networks. The most recent effort is included in NetworKit [21], GRAPPOLO [22] and studied in [20, 23]. GRAPPOLO uses a different and more complex algorithm than NetworKit. For simplicity, we present the Parallel Louvain Method (PLM), used by NetworKit.

PLM [20] is a shared-memory parallelization of the Louvain Method [13]. The algorithm performs the move phase in parallel by giving each thread different vertices to compute the affinity and their assignment to communities. It then coarsens the graph and recursively performs its optimizations. The runtime of PLM is mostly dictated by the first `move` phase; the process of converging the communities on the original graph, before any coarsening in done [20]. Trying to move vertices in parallel is not a race condition free process. Indeed, the algorithm may attempt to move two adjacent vertices simultaneously. PLM is optimistic and assumes that only a few benign race conditions will happen in practice. However, race conditions may cause the process not to converge; PLM stops the move phase after 25 iterations, whether communities have converged or not.

In practice, Parallel Community Detection codes have limited multi-core scalability [20]; in particular because of the noted convergence issues. Since using multiple core reaps little benefit, this paper focuses on using each core more efficiently by leveraging vector SIMD operations. And we consider improving multi-core scalability orthogonal to this work.

3.3 Parallel Label Propagation

Raghavan [2] *et. al* introduced a label propagation community detection method to extract community of a graph by the labeling of the vertices. First, all vertices are labeled as unique number, that means each node belong to their own singleton community. Then multiple iterations over the vertex set are performed: In each iteration, every vertex adopts the most frequent label in its neighborhood (breaking ties arbitrarily). Densely connected groups of vertices thus agree on a common label, and eventually a globally stable consensus is reached, which usually corresponds to a good solution for the network.

Algorithm 5 represents the label propagation method. From line 1 to 3, each vertex assign to a singleton community which is the label of that vertex. At line 4, a variable *updated* is initialize by the number of vertices $n = \mathsf{V}$. The main purpose of the *updated* variable is to terminate the method when the total number vertices that change their community is lower than the threshold value θ. Active vertex set V_{active} is initialized at line 5. Line 6 to 18 represent the repeative section to update the community. At the beginning of each iterative process(line 7) *updated* variable set to 0. From line 8 to 17, for each vertex u find the neighboring label l that maximize $\sum_{v \in N(u):\zeta(v)=l} \omega(u, v)$. If $l \neq \zeta(u)$ then label l assign to vertex u, and increment the

Algorithm 5 Label Propagation

Input: graph $G = (V, E)$
Result: communities $\zeta : V \to \mathbb{N}$

for $u \in V$ **do**
  $\zeta(u) \leftarrow id(u)$
end for
$updated \leftarrow n$
$V_{active} \leftarrow V$
repeat
  $updated \leftarrow 0$
  for $u \in V_{active}$ and $deg(u) > 0$ **do**
    $l^* \leftarrow \arg\max_l \{\sum_{v \in N(u):\zeta(v)=l} \omega(u, v)\}$
    if $\zeta(u) \neq l^*$ **then**
      $zeta(u) \leftarrow l^*$
      $updated \leftarrow updated + 1$
      $V_{active} \leftarrow V_{active} \cup N(u)$
    else
      $V_{active} \leftarrow V_{active} n \{u\}$
    end if
  end for
until $updated > \theta$
return ζ

variable *updated* by 1 and mark all neighbors of u as active. The process terminates when $updated \leq \theta$

4 ONPL: One Neighbor Per Lane

The first strategy that we investigate, One Neighbor Per Lane (ONPL), uses the entire vector to process different neighbors of the same vertex.

4.1 Speculative Greedy Graph Coloring

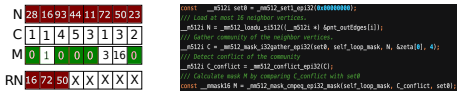
For graph coloring, the conflict detection method naturally vectorizes. Vectorization will be useful when marking which colors can not be used for a vertex. One can vectorize the loop that considers all the neighbors of a vertex. The operation boils done to loading 16 neighbors at a time with a `load` instruction; load the colors of these neighbors using a `gather` instruction. Then marking the used colors using `scatter` instruction. Identifying the first available color and identifying conflicting coloring vectorize naturally.

4.2 Louvain Method

Vectorized affinity calculation is complex because if two neighbors in the same community appear in the same vector, their contribution to the affinity of that community compounds. It will lead to conflicts during affinity calculation that requires resolving. We present the *One Neighbor Per Lane(*ONPL) vectorized Louvain method for community detection using intrinsic notations.

In `AVX-512`, the registers are 512 bits large so that it enables the ability to load 16 neighbors of a vertex at a time to process. Computing the affinity values requires a sequence of load, gather, addition, and scatter operations. Vectorized affinity will work well if all the vertices have their neighbors in different communities. Otherwise, blindly scattering causes some of the updates to be discarded, leading to incorrect affinity values. It requires summing the edge weights of every community before accessing the current affinity of adjacent communities. This operation is essentially a *reduce and scatter*. Unfortunately, no instruction directly does this operation. But the `AVX-512F` and `AVX-512CD` instruction sets enable two different ways to handle this. *ONPL* uses either one of them, depending on circumstances and these two instruction sets are sufficient to implement the algorithm.

Consider the extreme case where all the communities in the vector are different. It is typical at the beginning of the execution of the community detection code. In such a case, the addition and scattering can occur independently without requiring any reduction. If we know that all the lanes are independent, then no two lanes will write to the same location. Fortunately, the AVX-512CD instruction set provides `_mm512_conflict_epi32` instruction that tests each 32-bit element of an array `A` for equality with all other elements in `A` closer to the least significant bit. Each element's comparison forms a zero extended bit vector in `dst`. This instruction(`_mm512_conflict_epi32`) is the basis of the *conflict detection* method for reduce and scatter as it enables the extraction of different sets of communities and neighbors that can safely process at the same time. Figure 1(a) represents the process. Here, N is the list of neighbors of a vertex, and C is the corresponding list of communities. Instruction(`_mm512_conflict_epi32`) is



(a) Conflict Detection.

(b) Code snippet to calculate mask M. *pnt_outEdges* represents the list of out edges, *self_loop_mask* is the mask to prevent the self-loop and *zeta* represents the list of community.

Fig. 1 Perform reduce scatter using conflict detection. The neighbors (N) are in their Communities (C). A mask (M) is derived from C to denote the entries that will be processed (in green). Some neighbors will remain (RN) to be processed.

(a) In-vector Reduction.

(b) Code snippet to calculate mask M. *pnt_outEdges* represents the list of out edges, *self_loop_mask* is the mask to prevent the self-loop and *zeta* represents the list of community.

Fig. 2 Perform reduce scatter by compressing the communities. The neighbors (N) are in their Communities (C). A mask (M) is derived from C to denote the entries that will be processed (in green). Some neighbors(RN) and communities(RC) will remain to be processed.

applied on C to calculate the mask M. Figure 1(b) shows the code snippet to calculate the mask M. There are two techniques to handle the conflicted case: the first iteratively performs the vector operation on the non-conflicted sets and performs as many iterations of vector operations as there are non-conflicted sets; the second one applies vector operation on a non-conflicted set of neighbors only once and performs the remaining entries using purely scalar operations. Indeed, in practice, this conflict detection method uses many instructions. And it only useful if many communities can process at once. The vector will process one entry at a time with expensive vector operations if adjacent vertices belong to the same community. One can avoid the problem by performing vector operation only on the first set of independent communities and use the scalar operations afterward in the conflict detection method.

Another extreme case comes when all the communities in the vector are identical. This case arises when the process has mostly converged. In this case, an *in-vector reduction* is preferable. This method (sketched in Figure 2(a)) masks out all the entries of the vector besides the one mapping to a particular community. Figure 2(b) shows the code snippet to calculate the mask M. Then the edge weight mapping to this community is reduced with a masked reduction instruction `_mm512_mask_reduce_add_ps` and is finally added back to the affinity of that community. In Figure 2(a), RN represents the remaining vertices that are not processed yet, and RC is the list of their corresponding communities. Similar to the conflict detection method, there are two ways to proceed. Successive communities can use mask and reduce; however, this can lead to an issue for vertices that sit at the border of many communities causing potentially a large vector overhead. In practice, ONPL only processes vector operations for the first community of the vector and defaults to scalar implementation for the remaining communities.

The calculation of modularity from the affinity and the assignment of vertices to the community is done with simple vector processing and does not pose particular challenges.

4.3 ONLP: One Neighbor Per Lane Label Propagation

Nodes traverse in a parallel fashion, which brings the randomization on the node selection. For each node, it loads 16 neighbors and gathers their corresponding labels at once. For each distinct label, it sums the neighbor edge weight to create a vector with label weight. Each vector lane handles one neighbor of the vertex. Then an Intrinsic instruction *_mm512_reduce_max_ps* applied to find out the heaviest neighbor label. A vertex participates in the next iteration if any of its neighbor labels change.

5 OVPL: One Vertex Per Lane

In the One Vertex Per Lane (OVPL) method, each SIMD lane processes different vertices. Initially, vertices of the graph are group into multiple blocks where the size of blocks is the multiple of the vector lanes. We have to restructure the network for the efficiency and convergence of the algorithm.

Because two vertices in a block will be processed simultaneously, OVPL requires two vertices in the same block not to be neighbors. Reordering the graph to have that property requires solving a graph coloring problem. Therefore it makes no sense to deploy OVPL for graph coloring. We only consider OVPL for the community detection problem.

5.1 Preprocessing

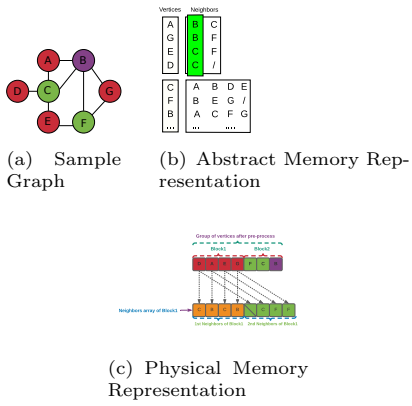
Vertices that are part of the same block will always be processed simultaneously. This property might induce race conditions that can prevent convergence. If the adjacent vertices are processed simultaneously, the affinity calculation performs on the changing information. The simplest case is a graph with two vertices that swaps their community infinitely, but the issue also appears on numerous complex networks.

To prevent this from happening, we first solve a graph coloring problem: we allocate a color to each vertex so that no two adjacent vertices have the same color. We then group the vertices where each group holds vertices with the same color. That will make sure that no vertices are adjacent in a group. While finding the coloring with a minimal number of colors is an NP-Complete problem [17], we do not require such a high-quality solution. We use the speculative parallel greedy graph coloring algorithm [18] we described in Section 3.1.

After grouping the vertices, we sort the vertices in each group by non-increasing degrees. Sorting will help to minimize wasted computation during execution.

Finally, we split each group of non-adjacent vertices into small blocks of equal size equal to a multiple of the number of lanes. We reformat the vertices of each block to enable vectorization by interleaving the representation of the different vertices. That also reduces unaligned memory accesses. The format is similar to sliced ELLPACK [24]. A contiguous memory of size $max\_deg\_of\_block \times block\_size$ holds each block of vertices. The index from $(i-1) \times block\_size$ to $i \times block\_size$ will represent the i^{th} neighbor of the vertices of each block. Edge weights also follow a similar representation.

Figure 3 shows a sample graph and its block structure. In the example, we assume the vector length is 4 for readability (instead of 16). So, the initial block will hold vertices that are not adjacent by selecting the same color. But in the second group, there are no four vertices with the same color; that is why it contains vertices of different colors to fill the vector.



(a) Sample Graph

(b) Abstract Memory Representation

(c) Physical Memory Representation

Fig. 3 OVPL reorders the graph using a graph coloring methods and structure it in blocks of vertices so that the neighbors of the vertices of a block can be loaded in a vector (sketched in green) simultaneously.

5.2 Moving a Block of Vertices

Rather than moving a vertex to its most preferable community, OVPL *moves* a block of vertices at once. It calculates the *affinity* of all the vertices of a block concurrently. Therefore OVPL has a much higher memory utilization than PLM because it keeps $block\_size$ affinity structures in memory.

OVPL computes the affinity of each vertex of the block one neighbor at a time. OVPL first loads the first neighbor of each vertex of the block at once and gathers the community of the first neighbors. Then it gathers the affinity of the neighbor communities from the different affinity arrays. OVPL adds the edge weights to the obtained affinity and scatters the updated values back to the appropriate locations. Note that because of this, it was not possible to perform this vectorization on x86 processors before scatter was introduced with AVX-512.

This process repeats until all the neighbors of all the vertices of the block are processed, i.e., until the maximum degree of the block. However, some vertices may have a lower degree, so OVPL needs to check the existence of the neighbor. This check increases the number of instructions and causes

the algorithm to use masked vector instructions. OVPL does not perform that check before the minimum degree of the block neighbors has been considered. The difference between the maximum and minimum degree in each block leads to wasted SIMD lanes. Preprocessing sorted the different color groups per degree to minimize the degree difference. Also representing the blocks by interleaving the vertices, enables access to the graph to aligned loads.

The assignment of vertices to new communities is done without particular optimization using a natural way of performing this task.

6 Experimental Settings

Hardware Platform and Operating System.

We used two different machines for the two architectures we study in this paper. We refer to the first machine as `SkylakeX`. It is a node with two Intel Xeon Gold 6154 processors (SkylakeX architecture, 18 cores per processor, no hyperthreading, 25MB L3 Cache) and 388 GB of DDR4 memory. The second machine is `Cascade Lake`, which is equipped with two Intel Xeon Gold model 6248R (Cascade Lake architecture, 24 cores per processor, no hyperthreading, 36MB L3 Cache) and 384GB GB of DDR4 memory. Both processors support Intel `AVX-512F` and `AVX-512CD` instruction sets with among others. Both machines use Linux `3.10.0`.

Software Environment.

All the codes are compiled by the Intel C++ compiler `icpc` version `16.0.0.109`. Codes also compile with optimization flag `-O3` and `xCORE-AVX512` flags, so the compiler generates a binary optimized for the architecture. We pick existing established code bases for both algorithms to confirm we start from implementations of reasonable good qualities.

We build graph coloring and community detection experiments on top of Kokkos [25] and *NetworKit* [21], respectively. We intended to compare to the original PLM implementation from [20]. During our experiments, we realized that PLM suffered from various memory management issues like large buffers were allocated and deallocated for each vertex traversed. We created a Modified PLM implementation (MPLM) that preallocates memory per thread. And then reuse the same buffer for the computation rather than deallocating and reallocating memory over and over. After confirming that MPLM is an improvement on PLM (See section 7.4.1), we will perform all other comparisons with MPLM.

Graphs.

We perform our experiments on real-world data sets to avoid the bias introduced by random graph generator. We select graphs from the *Stanford Large Network Dataset Collection* (SNAP) [26] and *DIMACS* [27, 28] data sets that are well known for graph algorithm research. Graphs are from different categories like Social networks, clustering instances, sparse matrices, internet topology networks, citation networks. We expect that the coverage in the type of graphs enables deriving conclusions that are more general and bias-free than picking all graphs from a single category. Table 1 presents the list of undirected graphs that we use in the experiments. The table also includes basic statistics such as the number of nodes (V), edges (E) of the graph, the maximum degree of the graph (Δ), and average degree (δ).

Table 1 List of graphs used in the experiment

Graph	Nodes (V)	Edges (E)	Δ	δ
333SP	3,712,815	11,108,633	28	5
AS365	3,799,275	11,368,076	14	5
M6	3,501,776	10,501,936	10	5
NACA0015	1,039,183	3,114,818	10	5
NLR	4,163,763	12,487,976	20	5
Oregon-2	11,806	32,730	2,432	5
asia	11,950,757	12,711,603	9	2
belgium	1,441,295	1,549,970	10	2
delaunay_n24	16,777,216	50,331,601	26	5
europe	50,912,018	54,054,660	13	2
germany	11,548,845	12,369,181	13	2
in-2004	1,382,908	13,591,473	21,869	19
kkt_power	2,063,494	6,482,320	95	6
loc-Gowalla	196,591	950,327	14,730	9
luxembourg	114,599	119,666	6	2
netherlands	2,216,688	2,441,238	7	2
nlpkkt200	16,240,000	215,992,816	27	26
roadNet-PA	1,088,092	1,541,898	9	2
uk-2002	18,520,486	261,787,258	194,955	28

Collection of Result Sets.

All the variants are run 25 times for each graph. The reported values of time and modularity

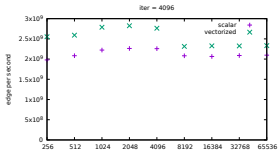


Fig. 4 Microbenchmark performance on SkylakeX.

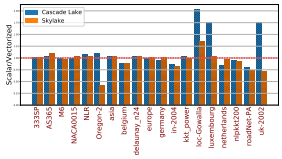


Fig. 5 [Graph Coloring] Impact of vectorization of Graph Coloring on both architectures. Y-axis represents the normalized version of the runtime comparison between scalar and vectorized. Scalar/Vectorized = 2.5 means vectorized version is 2 times faster than scalar.

are average of the 25 runs. For runtime, we only measure the time taken by the community detection(*Move-Phase*) and graph coloring algorithm itself, not the time spent reading the graph from the file system. We computed the 95% *confidence interval* [29] for the results of all the experiments. Once we realized the confidence intervals were very narrow and that the visible differences in the plots were statistically significant, we choose not to report them to improve figures readability.

7 Performance Results

7.1 Microbenchmark

The microbenchmark simulates the affinity calculation of a single vertex in a fairly dense graph (with 4096 neighbors per-vertex packed along the diagonal). The code does a sequence similar to the operations of the algorithms we consider: load, gather, and scatter when running vectorially. The benchmark is written to compare a scalar implementation and a vector implementation.

The results for the SkylakeX architectures (in Figure 4) highlight that there are little differences in SkylakeX between the scalar and vectorized performance, with the vector implementation being 20% faster than the scalar one.

This sets the expectation for our problems. The microbenchmark is essentially what graph coloring does. For a graph with a large degree and the best diagonal layout, SkylakeX is only 20% faster using vector instructions than scalar ones. The community detection problem could see higher improvements because the problem is more computational.

7.2 Speculative Greedy Graph Coloring

The performance of the ONPL vectorization on graph coloring is displayed in Figure 5 for the *Cascade Lake* and *SkylakeX* architectures. Vectorized speculative graph coloring on both processors shows moderate performance enhancement for some graphs over the scalar version. Vectorized graph coloring on the Cascade Lake and SkylakeX outperform the scalar version by at most factors of 2 and 1.4. Speculative parallel graph coloring has two main parts. One is the assignment of color, and another is conflict detection. We only apply vectorization on the color assignment portion. Graph coloring has a limited opportunity for vectorization that is why it shows a moderate performance for most of the graphs.

7.3 Performance on R-MAT Graph

7.3.1 R-MAT Graph

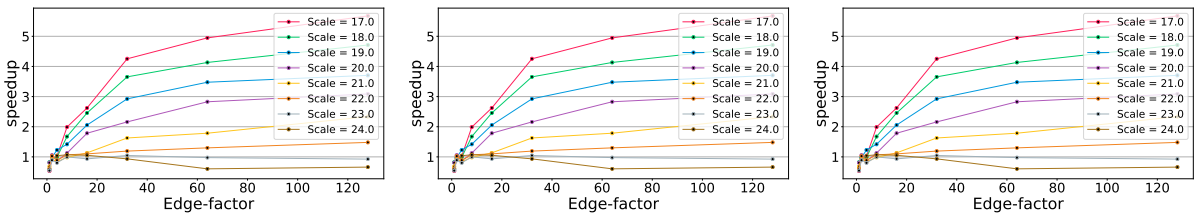
R-MAT [**?**] is one of the most common synthesis graph generators which aims to maintain the power law of the natural graph. To generate a graph using R-MAT, one usually 6 attributes. First one is the *scale* which determine the number of nodes (2^{scale}) in the graph. Next attribute is the *edge-factor* which maintain the average degree of the graph. Then 4 parameters (a, b, c, d) to maintain probability distribution of edges among nodes. Usually adjacency matrix divided into 4 quad and each edge choose one quad based on the probability of that quad. R-MAT graph useful to describe results based on the structure of the graph. Here is the list of parameters we used to generate R-MAT graphs and to make it fair we perform different version of Label Propagation and Louvain method on the same graph. Table 2 represents parameters we used to generate R-MAT graph.

7.3.2 Label Propagation

Figure 6 and 7 shows the performance of the Label propagation on R-MAT graph on the Cascade

Table 2 R-MAT Parameters

Scale	Edge-factor	Probability Distribution
17, 18, 19, 20, 21, 22, 23, 24	1, 2, 4, 8, 16, 32, 64, 128	a=33%, b=33%, c=33%, and d=1%
		a=40%, b=30%, c=20%, and d=10%
		a=57%, b=19%, c=19%, and d=5%



(a) RMAT parameters a=33%, b=33%, c=33%, and d=1%

(b) RMAT parameters a=40%, b=30%, c=20%, and d=10%

(c) RMAT parameters a=57%, b=19%, c=19%, and d=5%

Fig. 6 Performance gain of the ONPL Label propagation against scalar on the RMAT graph with different edge-factor on Cascade Lake processor.

Lake processor. We can see from Figure 6 that performance gain of the Label propagation increased with higher edge-factor. Now, ONPL perform vectorization on one neighbor per lane that means higher edge-factor enable higher vectorization. We can also notice that performance of the application higher for the lower scale graph. Which gives the insight of the overall graph size has huge impact on the performance. Number of edges of a R-MAT calculated by $2^{scale} \times (2 \times edge - factor)$. Now, bigger graph brings higher cache misses because of the limitation of the memory size. So, if a graph shows higher average degree and size of the graph accommodate by the system memory then vectorized ONPL Label propagation will show a tremendous performance compare to scalar version. Figure 7 also provide similar evidence that smaller size (vertices) graph with higher edge-factor provide huge spike in performance gain.

7.3.3 Louvain Method

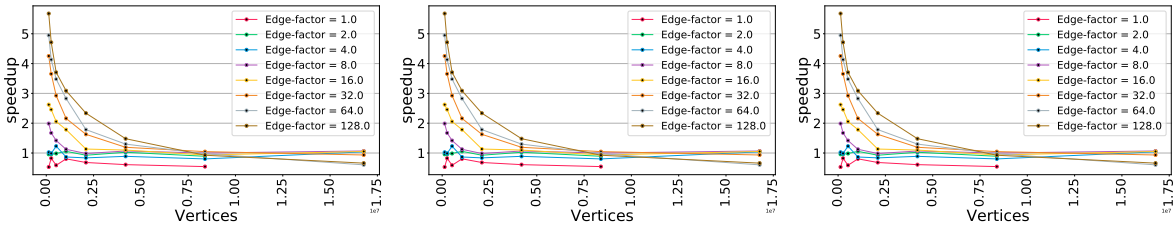
Figure 8 and 9 show the performance of the Louvain method on the R-MAT graph. Louvain method shows the similar nature but the performance gain lower than the Label Propagation. One of the main reasons, the calculation of the Louvain method way much complex than Label Propagation. Memory usages is also higher in Louvain method which led more cache misses. But experiments on the R-MAT graph follow the main argument of the work that one should choose vectorized version ONPL Label propagation and Louvain for graphs with higher averages edges. If a graph shows lower average degree then scalar version is well suitable.

7.4 Louvain Method on NetworKit

7.4.1 Modified Parallel Louvain Method (MPLM)

We noticed some performance deficiencies in PLM, like threads reallocation of the memory needed for the affinity computation for each vertex that it encounters. To be able to study the impact of vector processing, we needed to make sure that the performance difference was rooted in vectorization rather than in memory management. The Modified Parallel Louvain Method (MPLM) is the code that contains various performance fixes for PLM.

Figure 10(a) presents the improvement of MPLM compared to PLM for 48 threads on *Cascade Lake* for all studied graphs. Similar results observe on SkylakeX (not shown for brevity). We will use MPLM as the comparison point to see the impact of vector processing in community detection codes.

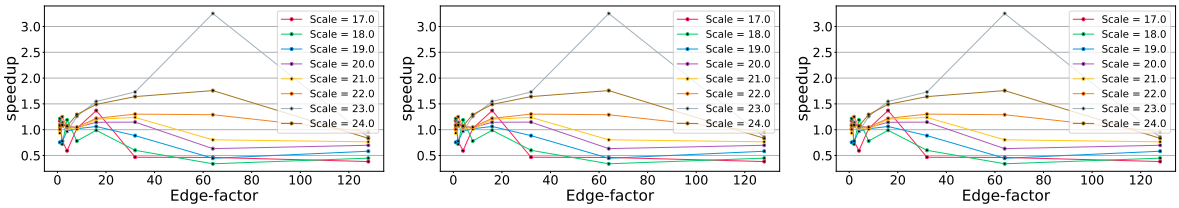


(a) RMAT parameters a=33%, b=33%, c=33%, and d=1%

(b) RMAT parameters a=40%, b=30%, c=20%, and d=10%

(c) RMAT parameters a=57%, b=19%, c=19%, and d=5%

Fig. 7 Performance gain of the ONPL Label propagation against scalar on the RMAT graph with different different number of vertices on Cascade Lake processor.



(a) RMAT parameters a=33%, b=33%, c=33%, and d=1%

(b) RMAT parameters a=40%, b=30%, c=20%, and d=10%

(c) RMAT parameters a=57%, b=19%, c=19%, and d=5%

Fig. 8 Performance gain of the ONPL Louvain Method against scalar on the RMAT graph with different edge-factor on Cascade Lake processor.

7.4.2 Modularity

Since the algorithm has significant race conditions, any change of timings could affect the quality of the communities detected. Modularity is one of the standard metrics to evaluate the quality of the communities and is the metric optimized by MPLM. Figure 10(b) shows the modularity of the implementations of MPLM, ONPL, and OVPL on the *Cascade Lake* architecture using 48 threads. All methods achieve almost the same modularity which confirms the quality of the vectorized communities has not been significantly impacted.

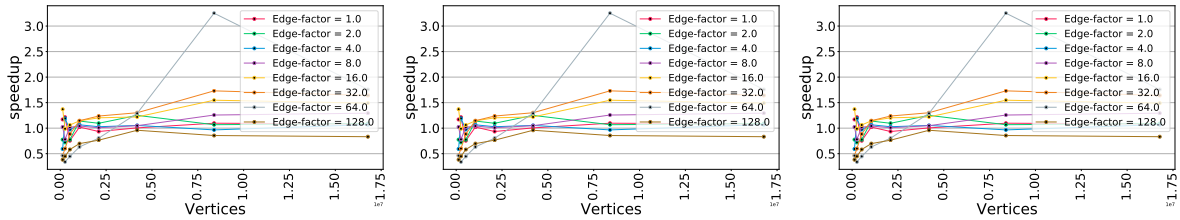
7.4.3 ONPL

is a vectorized algorithm with the same memory consumption as the scalar algorithm MPLM and similar memory access patterns. Figure 11(a) shows the performance of ONPL compared to MPLM on the Cascade Lake for 48 threads: ONPL shows performance improvement for most of the selected graphs and at most a factor of 2.5 performance gain compared to *MPLM*. Figure 11(b) shows the *ONPL* performance in the NetworKit on the *SkylakeX* architecture. ONPL performs better than its scalar counterpart for almost all the graphs. The best performance of *ONPL* is recorded on the *SkylakeX* processor is around a factor of 1.8 compared to *MPLM*.

7.4.4 OVPL

is an algorithm that consumes a lot more memory than the scalar algorithm due to having to store community affinity information for an entire block of vertices. Figure 11(a) presents the results of OVPL on the *Cascade Lake* architecture relative to the scalar implementation. For the graphs that were completed (some graphs ran out of memory), the performance derived is much better than the scalar implementation. Figure 11(b) shows the performance of OVPL on SkylakeX. We can see a factor of 9.0 and 6.5 performance gain for *OVPL* on the *Cascade Lake* and *SkylakeX* processors respectively compared to *MPLM*.

From the algorithm perspective, *OVPL* performs vectorization on a block of vertices, more specifically proper vectorization applies on the iteration only the minimum degree of vertices from the block. The rest of the iterations need more branching and also some lanes of the vectorization always remain unused. Our experimental

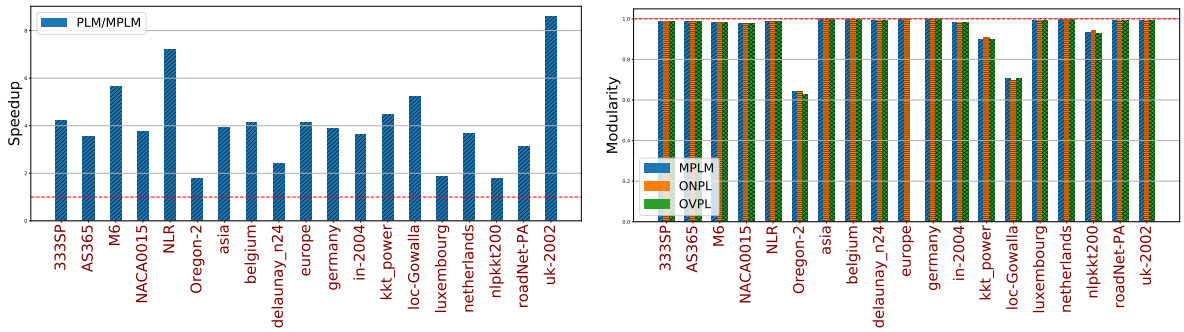


(a) RMAT parameters a=33%, b=33%, c=33%, and d=1%

(b) RMAT parameters a=40%, b=30%, c=20%, and d=10%

(c) RMAT parameters a=57%, b=19%, c=19%, and d=5%

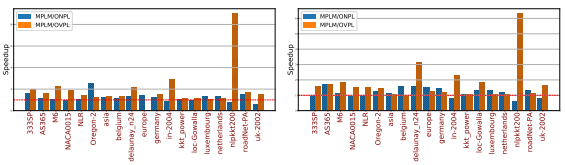
Fig. 9 Performance gain of the ONPL Louvain Method against scalar on the RMAT graph with different different number of vertices on Cascade Lake processor.



(a) PLM vs MPLM speedup on the Cascade Lake(48 threads).

(b) Modularity of MPLM, ONPL, and OVPL on Cascade Lake(48 threads).

Fig. 10 [Louvain Method] Performance and quality of the Modified PLM (MPLM) over PLM.



(a) On Cascade Lake Processor(48 threads).

(b) On SkyLakeX Processor(36 threads).

Fig. 11 [Louvain Method] Speedup of ONPL and OVPL over MPLM.

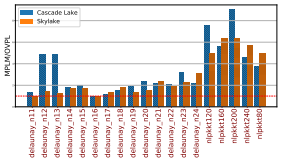
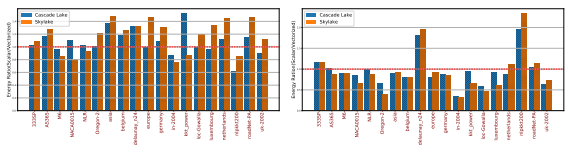


Fig. 12 [Louvain Method] Speedup of *OVPL* over *MPLM* for the selected graphs where many vertices have degrees close to the average on both architectures.

results also reflect the scenario. Figure 12 shows only the performance of the selected graphs where most of the vertices have the same degree or very

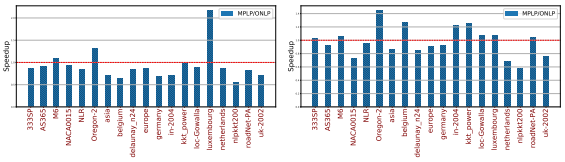


(a) MPLM vs ONPL.

(b) MPLM vs OVPL.

Fig. 13 [Louvain Method] Energy consumption comparison of ONPL and OVPL over MPLM on Skylake(36 threads) and Cascade Lake(48 threads).

small variations. It shows a great performance gain. Graphs like `Delaunay`(average Degree 5) triangulations of random points or sparse matrix `nlpkkt`(average Degree 26) have most vertices with degrees close to the average. Every vertex in *OVPL*'s block is in sorted order and properly distributed by their degree, which also brings great load balancing.



(a) On Cascade Lake Processor(48 threads).

(b) On SkyLakeX Processor(36 threads).

Fig. 14 [Label Propagation] Speedup of vectorized Label Propagation(ONLP) over the parallel Label Propagation(MPLP).

7.4.5 Energy Consumption

Figure 13 shows the overall energy consumption of different Louvain methods. The energy consumption by Louvain methods is calculated from RAPL (Running Average Power Limit) energy usages. Figures 13(a) and 13(b) show the energy consumption of ONPL and OVPL over MPLM. Any bar above 1 represents ONPL or OVPL consuming less energy than MPLM. Using SIMD operations helps to reduce the number of instructions in the execution. So, the expectation is that it can give better run time as well as better energy usage.

OVPL consumes more energy compared to ONPL and MPLM. Indeed, OVPL needs extra pre-processing, so it is adds work to enable vectorization. Also, because the vectorization pads the graph representation to fit the vector lanes, there are cases where vector lanes are actively used to perform no useful computation, which raises energy consumption. Since OVPL adds work and wastes vector lane, it makes sense that it raises energy consumption.

ONPL shows decent energy efficiency for both architectures (Cascade Lake and Skylake). Most graph tested have a better energy consumption with ONPL than with MPLM. If we compare Figures 11 and 13, we can see that some graphs see better energy gains than speedup. For instance, *uk-2002* see a slowdown from ONPL but a factor of 1.2 of gain in energy efficiency. That means vectorization can help graph algorithms by not only making them faster but also energy efficient. We conjecture that while vector instruction consume more power, they decrease the number of instruction that need to be decoded which can translate in energy gains.

7.5 Label Propagation on NetworKit

Figure 14 shows the performance of label propagation(LP) on Cascade Lake and SkylakeX processors. The parallel and vectorized one neighbor per-lane label propagations represent by MPLP and ONLP. A couple of graphs get moderate performance gain for ONLP on Cascade Lake 14(a) processor; the highest performance gain is reported around 2.0 times over MPLP. Graphs on SkylakeX processor 14(b) also show moderate performance.

It is possible to vectorize the Label Propagation, but it has limited benefits. In the Louvain method, we vectorize the affinity calculation and modularity calculation code sections. Both of the code sections are required a good amount of instructions to assign vertices to their respective community. So while gather and scatter provide limited performance benefits, they enable the rest of the affinity and modularity calculation to be vectorized which improves performance. However, the vectorization of the Label Propagation does not lead to many more instructions to be vectorized.

8 Related Work

Label propagation is one of the most popular community detection algorithm proposed by Raghavan *et al.* [2]. The algorithm iteratively refines labeling of vertices to communities by finding for each vertex the label that most frequently appears in its neighborhood and migrating the vertex to that label. PLM [20] is the shared-memory parallelization of the Louvain Method [13] we use as a reference. Halappanavar *et al.* [23] presented community detection for static and dynamic networks using Grappolo.

Cheong *et al.* [30] proposed a parallel Louvain method for GPUs using three levels of parallelism for the single and multi-GPU architectures. Later, Naim and Manne *et al.* [31] proposed a highly scalable GPU algorithm for the Louvain method, which parallelizes the access to individual edges. There are other recent works like Sanders *et al.* [32] proposed Louvain method for the python; the main objective of their work is the simplicity to implement the algorithm in python language. Gheibi *et al.* [33] proposed a cache efficient Louvain method for Intel Knight Landing(KNL) and Haswell architecture.

Both GPUs and CPUs are SIMD systems, at least in spirit. Taking the analogy of a GPU warp as a core and a thread inside a warp as a lane, algorithms for GPUs can be re-envisioned as vectorized CPU algorithm. At a very high level, the distinction between vertex-based algorithms (such as OVPL) and edge-based algorithms (such as ONPL) appears in GPUs. However, there are still many differences between the architectures which cause engineering and algorithmic decisions for CPU and GPU systems very different.

9 Conclusion

We considered the impact of AVX-512 instructions on graph partitioning problems. We investigated, in particular, the *Cascade Lake* and the *SkylakeX* architectures and how to use them to perform speculative greedy graph coloring, the Louvain method, and Label Propagation.

Using different SIMD lanes for different vertices only makes sense for the Louvain Method as this vectorization requires a pre-processing overhead. The vectorization forces to process blocks of vertices with the same number of neighbors, which induces some work overhead. It proved to be particularly efficient for graphs with balanced degrees and high average degrees.

The vectorization strategy that processes multiple neighbors of a single vertex at once also shows performance improvement for many graphs. That strategy is only possible thanks to scatter instructions and other various new instructions in `AVX-512` that are critical to partitioning problems. The reduce and scatter pattern is critical in implementing these vectorizations.

Reduce-scatter as a concept can be implemented in multiple ways with vector instructions. We implemented both in our software environment by using intrinsic operations. In future works, we want to investigate compiler techniques to enable us to deploy these techniques on more graph partitioning kernels without requiring low-level programming expert.

Acknowledgement

This work is supported by grant from the National Science Foundation CCF-1652442 and was made possible by a computing allocation given by TACC through XSEDE.

References

[1] S. Porta, V. Latora, F. Wang, E. Strano, A. Cardillo, S. Scellato, V. Iacoviello, and Messora R. Street centrality and densities of retail and services in Bologna, Italy. *Environment and Planning B: Planning and Design*, 36(3):450–465, 2009.

[2] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007.

[3] Ullas Gargi, Wenjun Lu, Vahab S Mirrokni, and Sangho Yoon. Large-scale community detection on youtube for topic discovery and exploration. In *ICWSM*, 2011.

[4] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *PNAS*, 99(12):7821–7826, 2002.

[5] Pall F Jonsson, Tamara Cavanna, Daniel Zicha, and Paul A Bates. Cluster analysis of networks generated through homology: automatic identification of important protein communities involved in cancer metastasis. *BMC bioinformatics*, 7(1):2, 2006.

[6] Christian Staudt, Andrea Schumm, Henning Meyerhenke, Robert Gorke, and Dorothea Wagner. Static and dynamic aspects of scientific collaboration networks. In *Proc. of ASONAM*, pages 522–526, 2012.

[7] V. Krebs. Mapping networks of terrorist cells. *Connections*, 24, 2002.

[8] Mehmet Deveci, Erik G Boman, Karen D Devine, and Sivasankaran Rajamanickam. Parallel graph coloring for manycore architectures. In *Proc. IPDPS*, pages 892–901, 2016.

[9] David W Matula, George Marble, and Joel D Isaacson. Graph coloring algorithms. In *Graph theory and computing*, pages 109–122. Elsevier, 1972.

[10] Gary C Lewandowski. Practical implementations and applications of graph coloring. 1995.

[11] Satu Elisa Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.

[12] George Karypis and Vipin Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM SISC*, 20(1):359–392, 1999.

[13] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[14] Chun Yew Cheong, Huynh Phung Huynh, David Lo, and Rick Siow Mong Goh. Hierarchical parallel algorithm for modularity-based community detection using gpus. In *Proc EuroPar*, pages 775–787, 2013.

[15] Md Maruf Hossain and Erik Saule. Impact of avx-512 instructions on graph partitioning problems. In *50th International Conference on Parallel Processing Workshop*, pages 1–9, 2021.

[16] Jierui Xie, Stephen Kelley, and Boleslaw K Szymanski. Overlapping community detection in networks: The state-of-the-art and comparative study. *ACM CSUR*, 45(4):43, 2013.

[17] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Freeman, 1979.

[18] Ümit V. Çatalyürek, John Feo, Assefaw H. Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. Graph coloring algorithms for multi-core and massively multithreaded architectures. *Parallel Computing*, 38(10-11):576–594, Oct-Nov 2012.

[19] Erik Saule and Ümit V Çatalyürek. An early evaluation of the scalability of graph algorithms on the intel mic architecture. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1629–1639. IEEE, 2012.

[20] Christian Staudt and Henning Meyerhenke. Engineering parallel algorithms for community detection in massive networks. *IEEE TPDS*, 27:171–184, 2016.

[21] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016.

[22] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, 2015.

[23] Mahantesh Halappanavar, Hao Lu, Ananth Kalyanaraman, and Antonino Tumeo. Scalable static and dynamic community detection using grappolo. In *Proc. IEEE HPEC*, pages 1–6, 2017.

[24] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *Proc. HiPEAC*, pages 111–125, 2010.

[25] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling many-core performance portability through polymorphic memory access patterns. *JPDC*, 74(12):3202–3216, 2014.

[26] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[27] Peter Sanders, Christian Schulz, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. 2014.

[28] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. *Graph partitioning and graph clustering*, volume 588. American Mathematical Society Providence, RI, 2013.

[29] Bradley Efron and Robert Tibshirani. Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical science*, pages 54–75, 1986.

[30] Chun Yew Cheong, Huynh Phung Huynh, David Lo, and Rick Siow Mong Goh. Hierarchical parallel algorithm for modularity-based community detection using gpus. In *Proc. EuroPar*, pages 775–787, 2013.

[31] M. Naim, F. Manne, M. Halappanavar, and A. Tumeo. Community detection on the gpu. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 625–634, May 2017.

[32] Tze Meng Low, Daniele G Spampinato, Scott McMillan, and Michel Pelletier. Linear algebraic louvain method in python. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 223–226. IEEE, 2020.

[33] Sanaz Gheibi, Tania Banerjee, Sanjay Ranka, and Sartaj Sahni. Cache efficient louvain with local rcm. In *2020 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6. IEEE, 2020.