2022

Exploring Multi-Level Parallelism For Graph-Based Applications Via Algorithm And System Co-Design

Zhen Peng
College of William and Mary - Arts & Sciences, hi.pengzhen@gmail.com

Exploring Multi-Level Parallelism for Graph-Based Applications via Algorithm and System Co-Design

Zhen Peng

Loudi, Hunan, China

Master of Engineering, Huaqiao University, China, 2016
Bachelor of Engineering, Huaqiao University, China, 2013

A Dissertation presented to the Graduate Faculty of
The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
January 2023

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Zhen Peng

Zhen Peng

Approved by the Committee, January 2023

Bin Ren

Committee Chair
Bin Ren, Associate Professor, Computer Science
College of William & Mary

Adwait Jog

Adwait Jog, Associate Professor, Computer Science
College of William & Mary

Andreas Stathopoulos, Professor, Computer Science
College of William & Mary

Pieter Peers, Associate Professor, Computer Science
College of William & Mary

Ruoming Jin

Ruoming Jin, Professor, Computer Science
Kent State University

ABSTRACT

Graph processing is at the heart of many modern applications where graphs are used as the basic data structure to represent the entities of interest and the relationships between them. Improving the performance of graph-based applications, especially using parallelism techniques, has drawn significant interest both in academia and industry. On the one hand, modern CPU architectures are able to provide massive computational power by using sophisticated memory hierarchy and multi-level parallelism, including thread-level parallelism, data-level parallelism, etc. On the other hand, graph processing workloads are notoriously challenging for achieving high performance due to their irregular computation pattern and unpredictable control flow. Therefore, how to accelerate the performance of graph-based applications using parallelism is still an open question.

This dissertation focuses on providing high performance for graph-based applications. To take full advantage of multi-level parallelism resources provided by CPUs, this dissertation studies the characteristics of graph-based applications and matches their parallel solutions with the underlying hardware via algorithm and system co-design.

This dissertation divides graph-based applications into three categories: typical graph algorithms, sequential graph-based applications, and applications with graph-based solutions. The first category comprises typical graph algorithms with available parallel solutions. This dissertation proposes GraphPhi as a new approach to graph processing on emerging Intel Xeon Phi-like architectures. The second category includes specialized graph applications without nontrivial parallel solutions. This dissertation studies a state-of-the-art 2-hop labeling approach named Pruned Landmark Labeling (PLL). This dissertation proposes Batched Vertex-Centric PLL (BVC-PLL), which breaks PLL's inherent dependencies and parallelizes it in a scalable way. The third category includes applications that rely on graph-based solutions. This dissertation studies the sequential search algorithm for the graph-based indexing methods used for the Approximate Nearest Neighbor Search (ANNS) problem. This dissertation proposes *Speed-ANN*, a parallel similarity search algorithm that reveals hidden intra-query parallelism to accelerate the search speed while fulfilling the high accuracy requirement. Moreover, this dissertation further explores the optimization opportunities for computational graph-based deep neural network inference running on tiny devices, specifically microcontrollers (MCUs).

Altogether, this dissertation studies graph-based applications and improves their performance by providing solutions of multi-level parallelism via algorithm and system co-design to match them with the underlying multi-core CPU architectures.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

The following dissertation is the combination of supervision, encouragement, and inspiration from many people throughout my time at William & Mary. First, I would like to thank my research advisor, Professor Bin Ren, for his thoughtful, patient, and considerate mentoring. He always encourages me to have deep thought about research questions, a comprehensive research view, an analysis ability of experimental outcomes, as well as a clear and sharp perspective of solutions. Most importantly, he told me the importance of critical thinking in research. These lessons will be essential tools for me in future work.

I sincerely thank my co-authors Professor Bo Wu, Doctor Tekin Bicer, Professor Ruoming Jin, Professor Gagan Agrawal, and Doctor Minjia Zhang. I am very thankful to have them as my collaborators who provide me with so much knowledge, guidance, support, and time. I would also like to thank all my research collaborators and colleagues with whom I have interacted.

I extend my gratitude to my dissertation committee members, Professor Adwait Jog, Professor Andreas Stathopoulos, Professor Pieter Peers, and Professor Ruoming Jin, for their generous support and attentive feedback. Without them, I could not finish my dissertation.

I would like to thank Doctor Xin Chen as my internship mentor at Kuaishou U.S. R&D Center, along with other colleagues. He introduced me to the new field of machine learning and deep neural networks. I also would like to thank Doctor Gekcen Kestor as my internship leader at PNNL, along with other group members who taught me a lot about compiler techniques.

I would like to thank all other faculty members in the Department of Computer Science at William & Mary. I am so grateful to have this opportunity to come to study here. I have learned a lot in their classes and received tremendous help and advice not only for my study and research but also for my life. I would also like to thank the entire administrative staff — Vanessa Godwin, Jacqulyn Johnson, and Dale Hayes — for being so caring, efficient, and professional.

I have spent memorable time with my group members Ruiqin Tian, Qihan Wang, Wei Niu, Yu Chen, Jiexiong Guan, Zhenqing Hu, and Jiaze E. They are not only my lab mates but also my friends who offer me so much support and company. I would also like to thank all my friends I have met throughout my Ph.D. journey.

Last but not least, I would like to thank my family members, who have always been caring, supportive, and encouraging. Without them, I would never be able to go this far.

To people.

LIST OF TABLES

LIST OF FIGURES

Exploring Multi-Level Parallelism for Graph-Based Applications via Algorithm and System Co-Design

Chapter 1

Introduction

A graph is a fundamental structure to represent data in various real-world applications. A graph consists of vertices and edges. A vertex represents an entity of interest, and an edge between two vertices represents a relationship or interaction between the two corresponding entities. As a result, graph processing is an important building block of many modern applications, such as map services [209], social network analytics [30], recommendation engines [63], scientific computations [38], and even machine learning [203]. Currently, those applications usually need to process very large graphs (e.g., with millions or even billions of vertices or edges) with strict latency requirements, which necessitates high-performance parallel computing.

Modern CPU architecture is equipped with massive parallel computing resources, including many-core processors [132], SIMD (Single Instruction, Multiple Data) processing units [41], and high-bandwidth memory [32]. For example, Knights Landing (KNL) is the second generation of Intel Xeon Phi processors [178]. It has up to 72 cores with four threads per core. Each core has two 512-bit vector units and supports AVX-512 SIMD instructions. It also has up to 16 GB of high-bandwidth memory providing about 400 GB/s bandwidth compared to 90 GB/s of normal memory. Meanwhile, Graphics Processing Units (GPUs) have become increasingly popular for general-purpose computing [29]. GPUs have more computing power and memory bandwidth than CPUs and also exploit

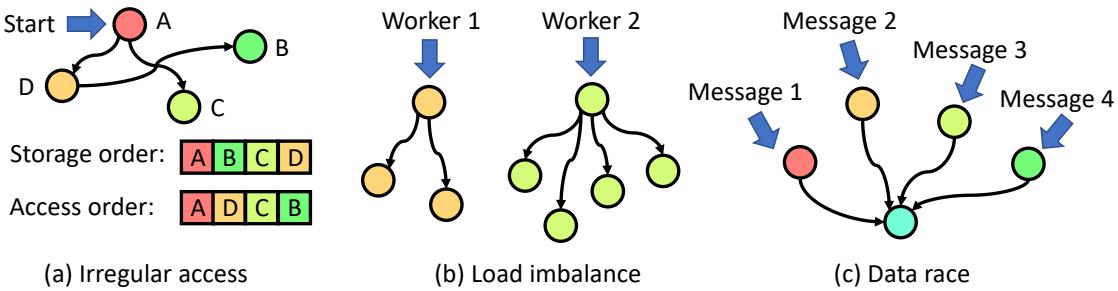


Figure 1.1: Typical performance challenges for graph-based applications.

data-level parallelism, which makes them well-suited to accelerate graph processing workloads. Compared to GPUs, CPUs have larger memories and caches, which enables larger graphs to be handled, making CPUs more scalable to very large datasets.

Even though modern machines are able to provide considerable computing power, graph-based applications are notoriously challenging to achieve high performance. A common issue of graph processing workload is its irregular computation pattern and unpredictable control flow [191]. Take the example of a typical graph algorithm Breadth-First Search (BFS). Even if there were only a small set of vertices active right now, all the rest vertices could be potentially active in the next computing iteration. Due to the irregularity, graph-based applications usually suffer from typical performance challenges such as frequent irregular access, serious load imbalance, and heavy data race conditions, as shown in Figure 1.1.

- **Irregular access.** When exploring the vertices in a graph via its edges, those visited vertices in order are usually not stored in contiguous memory locations. This misalignment results in pool data locality of memory accesses.

- **Load imbalance.** In practical applications, a graph is usually a scale-free network in that vertices' degree distribution follows a power law. When assigning vertices with huge degree differences to multiple processors, they are likely to have an imbalanced workload because high-degree vertices require much more computation than low-degree ones.

- **Data race.** Data races are not uncommon in a graph processing scenario, as multiple edges may connect to the same vertex. When multiple processors attempt to update the same destination, a data race occurs. Guaranteeing the correctness after the update usually requires extra synchronization overhead.

Moreover, some graph-based applications use well-designed sequential solutions. These solutions are carefully composed to reduce algorithmic complexity or computation time. However, many of them inevitably incur control dependency or data dependency that impedes efficient parallelization. Consequently, how to optimize the performance of graph-based applications remains an open question.

1.1 Problem Statement

This dissertation studies how to explore multi-level parallelism for graph-based applications via algorithm and system co-design. Modern CPU architectures can provide powerful computational resources for multi-level parallelism, including thread-level parallelism, data-level parallelism, etc. The goal of this research is to provide high-performance solutions for graph-based applications by considering and exploiting the underlying hardware. On the one hand, for traditional graph algorithms, this dissertation presents the insight that the whole graph processing system stack, including data representation, the execution model, and job scheduling, should match the features of the hardware. On the other hand, for inherently sequential methods, it analyzes the characteristics of the applications and then provides parallel algorithms to break their dependency. Meanwhile, through algorithm and system co-design, the proposed optimizations aim to mitigate any potential issues of the introduced parallelism, such as extra computation and synchronization overhead.

This dissertation divides graph-based applications into three categories. The first category comprises typical graph algorithms with available parallel solutions. As mentioned above, modern parallel architecture design has increasingly turned to throughput-oriented

devices to address concerns about energy efficiency and power consumption. However, graph applications cannot tap into the full potential of such architectures because of highly unstructured computations and irregular memory accesses.

The second category includes specialized graph applications without nontrivial parallel solutions. In this dissertation, we perform a case study on such an algorithm: the Pruned Landmark Labeling (PLL) method, which is a state-of-the-art solution of the 2-hop labeling approach to solving the shortest path distance problem for large graphs. The original PLL algorithm is inherently sequential. The algorithm operates on one vertex at a time to label the entire graph, and the labeling of a vertex depends on the partial labeling results from earlier processed vertices.

The third category includes applications that rely on graph-based solutions. In this dissertation, we study an example of such an application: the approximate nearest neighbor search (ANNS), which searches for the nearest neighbors in high-dimensional datasets given a query point. The interest is fueled by the success of neural embedding, where deep learning models transform unstructured data into semantically correlated feature vectors for data analysis, e.g., recommend popular items. Among several categories of methods for fast ANNS, similarity graphs are a popular choice to base such solutions on. It employs best-first traversal along the underlying graph indices to search for nearest neighbors. Maximizing the performance of the search is essential for many tasks, especially at the large-scale and high-recall regime. Compared to inter-query parallelism, intra-query parallelism is able to shorten the query latency. However, the sequential search algorithm cannot efficiently leverage the full capabilities of multi-core processors.

Moreover, there are graph-based applications running on resource-constraint devices. Specifically, deep neural network inference can be regarded as a graph-based application because of its computational graph. In this graph, vertices correspond to operations, and the edges correspond to the dependencies between individual operations. The inference proceeds by evaluating operations in their topological order, which requires computational resources to process and hold the data in a device's memory. Meanwhile, tiny devices such

as microcontrollers (MCUs) are widely used and integrated into daily life. They usually have low clock speed processors and very small memory capacity in the order of hundreds of kilobytes. How to enable and improve the inference on those tiny devices is still an open question for both academia and industry.

1.2 Contributions

This dissertation aims to improve the performance of graph-based applications via algorithm and system co-design.

First, this dissertation presents GraphPhi, a new approach to graph processing on emerging Intel Xeon Phi-like architectures, by addressing the restrictions of migrating existing graph processing frameworks on shared-memory multi-core CPUs to this new architecture. Specifically, GraphPhi consists of 1) an optimized hierarchically blocked graph representation to enhance the data locality for both edges and vertices within and among threads, 2) a hybrid vertex-centric and edge-centric execution to efficiently find and process active edges, and 3) a uniform MIMD-SIMD scheduler integrated with lock-free update support to achieve both good thread-level load balance and SIMD-level utilization. Moreover, our efficient MIMD-SIMD execution is capable of hiding memory latency by increasing the number of concurrent memory access requests, thus benefiting more from the latest High-Bandwidth Memory techniques. We evaluate our GraphPhi on six graph processing applications. Compared to two state-of-the-art shared-memory graph processing frameworks [148, 175], it results in speedups up to 4X and 35X, respectively.

Second, this dissertation demonstrates the first scalable parallel implementation of the PLL algorithm that produces the same results as the sequential algorithm. Based on theoretical analysis, it shows how computations on each vertex can be performed in parallel while maintaining correctness, resulting in the Vertex-Centrix PLL (VC-PLL) algorithm. It also shows a formulation of this algorithm based on linear algebra and argues why the use of a library based on linear algebra operations will not produce an

efficient implementation. Next, this dissertation introduces a batched VC-PLL (BVC-PLL) algorithm to reduce the computational inefficiency in VC-PLL. This dissertation has carried out a parallel implementation of this method for modern clusters, combining shared memory and distributed memory parallelism, that can efficiently execute on graphs with more than a billion edges. It also demonstrates how the BVC-PLL algorithm can be extended to handle directed graphs and weighted graphs and how the version for weighted graphs can benefit from SIMD parallelization. In the results, the sequential BVC-PLL can run more than 2× faster than the original PLL (both using one single thread). And the parallel BVC-PLL shows an average speedup of 6.6× over sequential BVC-PLL on a 20-core shared memory machine and up to 11.8× on a 16-node cluster.

Third, this dissertation provides an in-depth examination of the challenges of state-of-the-art similarity search algorithms, revealing its challenges in leveraging multi-core processors to speed up the search efficiency, i.e., the query latency. It also explores whether similarity graph search is robust to deviation from maintaining strict order by allowing multiple walkers to simultaneously advance the search frontier. Based on the insights, this dissertation proposes *Speed-ANN*, a parallel similarity search algorithm that exploits hidden intra-query parallelism and memory hierarchy that allows similarity search to take advantage of multiple CPU cores to significantly accelerate search speed while achieving high accuracy. The results show that *Speed-ANN* reduces query latency by 13× and 17.8× on average of million-scale datasets than two state-of-the-art graph-based solutions at 0.999 recall target, respectively. It also offers up to 16.0× speedup on two billion-scale datasets.

Fourth, this dissertation explores the optimization opportunities for deep neural network inference on MCUs. First, for the neural network that can be fitted in the device memory, it tests how lightweight quantization and loop unrolling influence the inference latency. Preliminary results show about 1.30× total speedup on a small MobileNetV2 model. Second, for the larger neural network that can not reside in the on-chip memory, it tests the effects of using off-chip memory, including Quad-SPI NOR Flash and SDRAM.

It demonstrated the possibility of doing inference of some large models by taking full advantage of available hardware resources, although causing some degree of performance declines.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 presents GraphPhi as a new approach to graph processing on emerging Intel Xeon Phi-like architectures. Chapter 3 demonstrates how BVC-PLL breaks the inherent dependence in the PLL algorithm and provides the opportunity for parallelism to speed up the solution. Chapter 4 proposes *Speed-ANN* which is a parallel similarity search algorithm that exploits intra-query parallelism to accelerate query latency. Chapter 5 presents the ideas to optimize neural network inference on MCUs and also shows some preliminary results. Finally, Chapter 6 concludes this dissertation and discusses future research directions.

Chapter 2

GraphPhi: Efficient Parallel Graph Processing on Emerging Throughput-oriented Architectures

2.1 Introduction

Throughput-oriented architectures equip many-core processors with wide SIMD (Single Instruction, Multiple Data) processing units to provide massive parallelism and high energy efficiency. For instance, seven of the top ten supercomputers around the world as of June 2018 [1] rely on the tremendous throughput offered by either GPUs or Xeon Phis. This trend is expected to continue through the whole post-Moore's law era to build future exascale systems.

Among the throughput-oriented architectures, Intel Xeon Phi is particularly attractive due to two reasons. First, the x86-compatible Xeon Phi-like processors allow running operating systems natively, and support various parallelization tools, libraries, and pro-

[1] https://www.top500.org/lists/2018/06/

gramming models, including OpenMP [50], MPI [81], CilkPlus [27] and Thread Building Blocks [164], making them friendly to programmers [95]. Second, the latest Xeon Phi, Knights Landing [178], can be used as the host processor, which automatically benefits from a large host memory hierarchy and eliminates data communication overhead through the PCIe bus.

It is promising to use the Intel Xeon Phi architecture to accelerate graph analytics, which plays a critical role in various domains, including bioinformatics, social networks, machine learning, and data mining. Unfortunately, it is not straightforward to map such graph analysis applications onto throughput-oriented architectures due to the severe mismatch between the significant irregularity shown by graph algorithms and the underlying many-core and vector-based processing.

Specifically, efficient graph processing on modern throughput-oriented architectures brings forward three challenges. First, different processing units may process uneven amounts of workloads while traversing different portions of the graph. Second, the SIMD unit cannot automatically resolve write conflicts (i.e., multiple SIMD lanes by executing the same instruction write to the same memory location), but the compiler or programmer cannot efficiently eliminate write conflicts due to the irregular accesses. A typical conservative approach which adds locks to synchronize the threads substantially degrades program scalability. Finally, the random memory access significantly decreases cache performance and memory throughput, hence underutilizing the tremendous computational resources.

There exist several graph processing frameworks and libraries based on popular many-core processors such as GPUs [109, 194] and early versions of Xeon Phis [42, 100, 135]. In addition, many other graph processing frameworks [148, 175] designed for shared-memory multi-core CPUs are also capable of running on Xeon Phis owing to their x86-compatibility. However, merely applying these techniques to emerging Xeon Phi architectures directly without any further optimization results in suboptimal performance. First, most of these efforts target either MIMD (Multiple Instruction, Multiple Data) or SIMD executions but

not both by assuming a uniform computation capability of different processing units. However, appropriately combining coarse-grained MIMD parallelism with fine-grained SIMD parallelism is critical to achieving optimal performance for Xeon Phi-like architectures. Second, the GPU-based frameworks assume hardware support to handle SIMD divergence and SIMD level conflicts but lack global synchronization due to the limitation of the hardware. However, Xeon Phi architectures have limited SIMD divergence support, no SIMD level locking, but cheap global synchronization. Third, none of the previous work has studied the interplay between optimized graph processing and the newly introduced High-Bandwidth Memory (HBM).

This work presents the GraphPhi optimizing framework to bridge graph processing and Intel Xeon Phi-like many-core processors. Our insight is that the whole graph processing system stack, from data representation to the execution model to job scheduling, should match the unique features of the hardware. Specifically, GraphPhi employs a hierarchical data organization for improving locality and simplifying SIMD processing. It exploits both coarse-grained MIMD and fine-grained SIMD parallelism in a *cache-aware*, *lock-free*, *load-balanced*, and *SIMD-efficient* manner for irregular graph processing. Moreover, since GraphPhi's hybrid MIMD and SIMD execution significantly increases the number of concurrent memory access requests, it changes latency-bound graph applications to bandwidth-bound ones, hence benefiting more from the emerging HBM techniques.

Overall, this work has the following contributions:

- Describing an optimized hierarchical blocked graph representation (with tiles/stripes/-groups) to enhance the edges' spatial data locality, and the vertices' temporal and spatial data locality within and among threads;
- Presenting a hybrid graph processing to efficiently find active edges by a vertex-centric approach and process edges in an edge-centric manner;
- Designing a uniform MIMD-SIMD scheduler to improve both thread-level load balance and SIMD-level utilization and lock-free update support on both thread and

SIMD levels.

We implement GraphPhi and evaluate it on six graph applications with six input data sets, achieving speedups up to $4X$ and $35X$ comparing to two state-of-the-art shared-memory graph processing frameworks, respectively. Moreover, we empirically prove that our efficient MIMD-SIMD execution is capable of benefiting more from the latest HBM techniques.

2.2 Background and Motivation

Recently, many graph processing frameworks and libraries have been developed to improve the performance of graph applications on various platforms. This section introduces the basic graph processing models used in these efforts and describes our focused architecture—latest Xeon Phi. Then it identifies the significant challenges that are rooted in the mismatch between graph processing and the hardware features. It also explains the difference between our work and previous work in GPU-based frameworks.

2.2.1 Graph Processing

Among the various proposed graph processing model, the two most relevant ones are *vertex-centric* and *edge-centric* models. We hence discuss them in detail.

The **Vertex-centric** model, also known as the "think-like-a-vertex" model, has been broadly adopted by many parallel graph processing frameworks [115, 175, 109]. Its original implementation in Google Pregel [136] demonstrates the model's simplicity, productivity, and strong scalability. It models parallel graph processing as an iterative process, which in each iteration traverses the *active* vertices in the frontier, processes their incoming and/or out-going edges, and updates the frontier. The parallelization typically uses the Bulk Synchronous Parallel (BSP) execution [190] and demands a global synchronization at the end of each iteration. The whole process terminates once the frontier becomes empty.

The **edge-centric** model was first proposed in X-Stream [166]. It keeps streaming edge partitions, the processing of which involves a gather stage and a scatter stage. The gather stage generates updates out of the active edges; the scatter stage applies the updates to the corresponding vertices. Similar to the vertex-centric model, a global synchronization is needed after each round of edge partition streaming to make sure that in the next round the gather stage can see only the updates generated in the current round.

Vertex-centric and edge-centric models present different benefits. Edge-centric processing avoids random accesses to edges by streaming on them sequentially, thus resulting in better disk I/O and memory performance. However, when only a small portion of the edges generate updates, streaming all the edges incurs substantial overhead. Such problems can be resolved by vertex-centric execution, which only processes the edges of active vertices but may degrade edge loading performance. Therefore, some systems like Mosaic [135] adopt a hybrid model to take advantage of both worlds.

2.2.2 Intel Xeon Phi Architectures

The latest Xeon Phi, Knights Landing (KNL), leverages a new tile design, which consists of two cores, two vector-processing units (VPUs) per core, and a 1M of shared L2 cache. A KNL chip has 32 (active) tiles (i.e., 64 cores and 128 VPUs) connected by a 2-D mesh interconnect. The cores are out-of-order and support all legacy x86 and x86-64 instructions. In addition, KNL has a High-Bandwidth Memory (MCDRAM) that can offer up to 400+ GB/s bandwidth in addition to the normal DDR4 main memory with $>$ 90 GB/s bandwidth according to the test on Stream Triad benchmark[2]. *We use the KNL as the central processor*, which directly connects to the main memory hierarchy rather than the PCIe bus.

A Large Number of Concurrent Threads: Each KNL core runs up to 4 hyper-threads, so the whole chip executes as many as 256 hardware threads that share the same DDR4 and MCDRAM memory. On the one hand, the massive thread-level parallelism

[2] `https://www.cs.virginia.edu/stream/`

has the potential to result in high processing throughput to take advantage of the HBM. On the other hand, the architecture is highly sensitive to latency-bounded workloads.

Powerful Vector Processing Units (VPUs): The VPU on Knights Landing is even more sophisticated than the one on Knights Corner (the previous version of the Xeon Phis architecture). Besides the *gather/scatter* and *mask* operations support in AVX-512 Foundation instructions (AVX-512F), the VPU implements more kinds of operations. For example, the Intel AVX-512 Conflict Detection Instructions (CDI) are able to detect the existence of write conflicts efficiently during SIMD execution, thus offering us a good opportunity of exploiting SIMD data parallelism to handle irregular memory write operations.

2.2.3 Our Challenges

Although there exist many GPU-based optimization techniques for irregular workloads [205, 198, 109], we notice significant architectural differences between Xeon Phi and GPUs, which make graph processing on Xeon Phi architectures uniquely challenging.

Xeon Phi vs. GPU: First, their memory hierarchies are different, e.g., GPU shared memory is a software-managed cache designed to reduce memory latency, while Xeon Phis are equipped with larger hardware-controlled caches and a separate High-Bandwidth Memory to increase memory bandwidth. Second, different from GPU's SIMT (Single Instruction, Multiple Threads) execution that assumes all threads have the same computation capability, in Xeon Phi's hybrid MIMD and SIMD execution, CPU and SIMD threads have different computation powers. Very importantly, Xeon Phi supports efficient global synchronization, but GPU programs have to be implemented in a particular format to enable global synchronization [83]. Third, although Knights Landing is more flexible compared to previous Xeon Phis, many lock-step features of original SIMD intrinsics (SSE) remain, *e.g.*, lack of atomic support within and among SIMD operations, and limited hardware support of SIMD divergences.

Considering the hardware differences introduced above, we highlight the following

challenges of optimizing graph processing on Xeon Phi architectures:

Data Locality: Due to the irregular data structure, it is notoriously challenging to layout graphs in modern memory hierarchy for good data locality, especially in the Xeon Phi architectures that have a small cache size per core. The memory access pattern to refer to active vertices and edges depends on the graph topology, algorithms, and processing models, which is hence unpredictable and leads to substantial cache misses. Therefore, many graph applications are latency bounded [196].

MIMD-SIMD Load Balance: Vertex-centric processing does not naturally match SIMD execution, because the workload assigned to the SIMD lanes vary substantially due to the skewed degree distribution. Since the SIMD lanes execute instructions in lock-step, the ones that process low-degree vertices may experience significant idleness. Edge-centric processing mitigates this problem but cannot eliminate it, as the same SIMD unit may process a hybrid workload of active and inactive edges.

Update Conflict: Graph processing usually involves reading attributes in the source vertices and writing attributes in the destination vertices. An update conflict happens if multiple SIMD lanes in the same unit update the same destination vertex. As aforementioned, the SIMD unit (i.e., the VPU) does not support atomic operations. Hence, when an update conflict happens, only the value produced by one SIMD lane can be successfully stored. One approach is to carefully map data to SIMD lanes to avoid write conflicts in the first place, which requires the knowledge of the whole access sequence at the very beginning of the execution. Another possible approach is to detect conflicts once the mapping is established, which requires careful overhead control.

2.3 Overview of Our Approach

In this section, we present the overview of the GraphPhi framework as shown in Figure 2.1. It consists of four major components (❶ to ❹) as follows:

The preprocessing (❶) transforms the input graph into a hierarchically tiled format

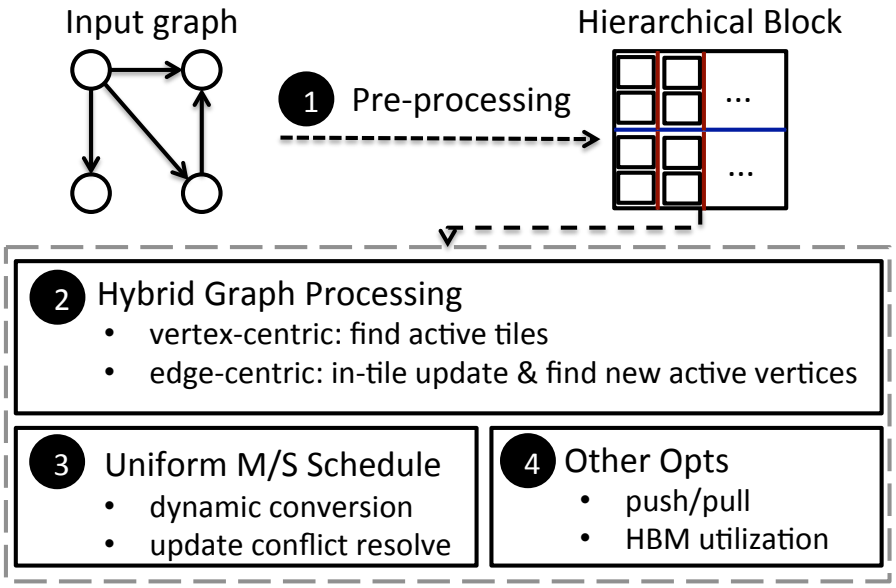


Figure 2.1: Overview of our approach.

based on the well-known COO representation, including tiles, stripes, and groups from small to large. This format is able to achieve multiple objectives, including improving temporal data locality by reducing reuse distance within the same thread and across threads, supporting later optimized hybrid vertex-centric and edge-centric graph processing, and enabling co-schedule of MIMD and SIMD tasks.

The graph processing model (❷) is a hybrid vertex-centric and edge-centric graph processing model to take advantage of their appealing features. GraphPhi leverages vertex-centric processing to identify which tiles to process but computes updates and explores new active vertices within a tile in an edge-centric way. There are several specific considerations in this design. First, an active tile that contains active source vertices should be processed. Vertex-centric processing in GraphPhi records which vertex tiles are active and only processes those tiles, hence reducing wasted computation; Second, compared to vertex-centric data storage (e.g., Compressed Sparse Row, or CSR for short), edge-centric in-tile storage has potential to substantially save storage space, especially for sparse graphs (details in Section 2.4). Finally, edge-centric in-tile storage is more friendly for SIMD processing because it leads to sequential memory accesses to edge data.

The MIMD/SIMD-aware scheduler (❸) addresses load imbalance through dynamic task conversion, i.e., if the SIMD utilization is low while extra tasks exist in MIMD level, it *merges* these tasks and explores better SIMD parallelism; if some threads are idle in the MIMD level while there are too many tasks for others, these tasks will be *split* and assigned to the idle threads. Moreover, the scheduler takes care of possible update-conflict complications caused by the conversions.

The final component (❹) represents several extra optimizations to improve further the overall performance, including the hybrid pull and push execution and the use of HBM offered by the latest Xeon Phi architectures.

2.4 Data Format and Execution Design

This section introduces the design of our GraphPhi framework and explains the design basis and more details of our hierarchical blocked graph representation, hybrid graph processing, and uniform MIMD-SIMD task scheduler. These designs aim at achieving improved intra- and inter-thread data locality, efficient lock-free graph processing, and both good MIMD load balance and high SIMD utilization.

2.4.1 Hierarchical-blocked Organization

GraphPhi decomposes the adjacency matrix into 2-D disjoint edge tiles. A certain number of rows of tiles compose a group, which consists of many columns of tiles, and each called a stripe. The edges in a tile are stored in COO (coordinate list) format, i.e., each edge is in the form of (*row*, *column*, *value*). The edges in the same tile are stored continuously in row major, achieved by sorting the edges by column IDs (destination vertices), and then by row ids (source vertices). All tiles in the same stripe are stored contiguously in column major order, which share the same subset of destination vertices. All stripes in the same group are stored contiguously.

Figure 2.2 shows an example of this hierarchical-blocked graph data format. On the

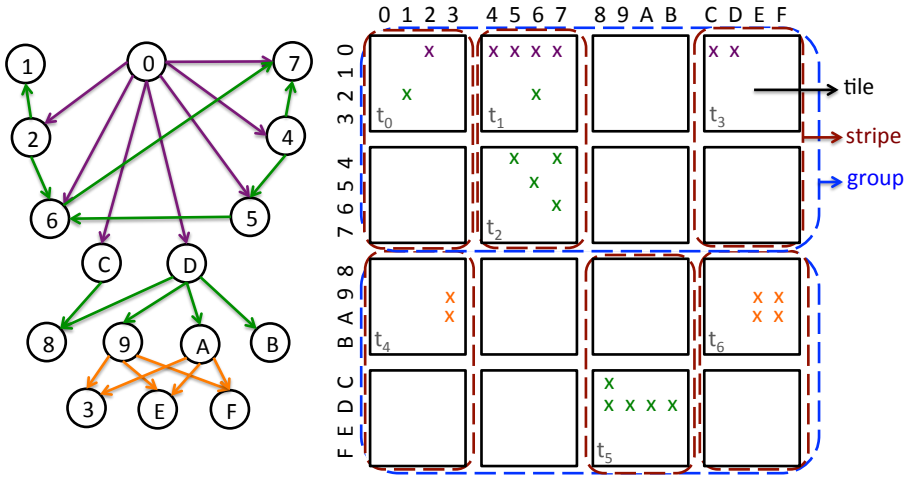


Figure 2.2: A hierarchical-blocked format example.

left-hand side, we show the topology of a graph that consists of 16 vertices and 24 edges; while on the right-hand side, we represent this graph in our hierarchical-blocked format. The representation has 16 tiles in total with 7 non-empty tiles with edges (t_0 to t_6), 6 stripes (marked with red-dot rectangles) and 2 groups (marked with blue-dot rectangles). Each edge is in COO format, i.e., the row ID is the source vertex ID while the column ID is the destination vertex ID, and the edge value is omitted. This example shows a directed graph, while for undirected graphs, each edge is treated as a pair of directed edges, one in each direction.

This hierarchical-block data format serves as the foundation of GraphPhi, supporting all the other optimizations. To help to understand this format design, we explain it from the following aspects: the COO format basis and the hierarchical design advantages.

2.4.1.1 Tile COO format design basis

The use of the COO data format for in-tile edges is an important design choice, which affects the other parts of GraphPhi. We adopt this design mainly from two considerations.

On the one hand, although CSR usually results in compressed data storage, it does not satisfy our requirement well. We explain the reason by an example in Figure 2.3. In this

example, we convert a tile (t_0) from Figure 2.2 to a standard CSR format. Although we have only two edges, we still need to maintain a `row pointers` array of length $width_{tile}+1$. Assuming there is no empty tile, we need up to $(\#vertices/width_{tile})^2 \times (width_{tile}+1)$, i.e., $O(\#vertices^2/width_{tile})$ space to store the `row pointers` arrays for the whole graph which is unacceptably large. An alternative solution is to change the `row pointers` array into another indirection array or a hash table structure, however, with the increased irregularity of the computation.

On the other hand, the COO format results in better storage efficiency when there exist many empty rows as shown in the above example. Moreover, it stores edge data continuously, leading to sequential memory load operations, so it is more suitable for streaming SIMD processing.

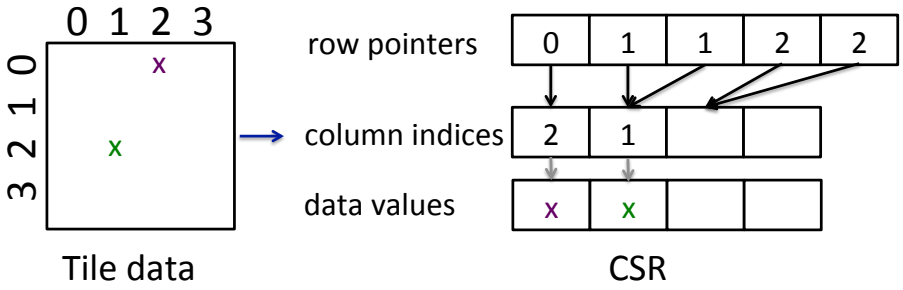


Figure 2.3: An example to show CSR problem.

2.4.1.2 Hierarchical design basis and advantages

We partition and hierarchically organize the whole graph into three levels due to the following reasons.

Why tile—intra-thread data locality: First, there is potential to improve the temporal data locality of memory accesses to *destination vertices* by organizing and processing the edges in the unit of a tile because the *reuse distance* of the same destination vertex is significantly reduced. For example, in tile t_1 of Figure 2.2, before tiling, the reuse distance of the destination vertex 6 is 5; while after tiling, it is 2. In addition, for sparse computation like our graph processing, tiling is also capable of improving the spatial data

locality by restricting updates within a specific range of destination vertices even when there is no data reuse. Because an individual thread processes a tile, these data locality enhancements are treated as intra-thread.

Why stripe—inter-thread update conflict: We design stripes for thread-level task scheduling. By default, a stripe is mapped to a distinct thread. Because stripes in the same group correspond disjoint sets of destination vertices, inter-thread update conflicts are naturally avoided. Such a design is similar to the design of *Shards* in GraphChi [115], and *G-Shards* in CuSha [109]. However, in contrast with *Shards* or *G-Shards*, our stripes contain only a subset of edges that share the same destination vertices, i.e., we further partition *Shards* or *G-Shards* by their source vertices as shown in Figure 2.2.

Why group—inter-thread data locality: Stripes that share the same source vertices are organized into the same group. Between the execution of two groups, there exists a global synchronization barrier. This design aims to improve inter-thread data locality for accesses to the source vertices due to two reasons. First, due to the global synchronization, the aggregate working set in terms of source vertices is limited and thus may fit in the cache. Second, the threads that process different stripes may prefetch source vertices to the cache for each other.

In addition, our graph preprocessing cost is low because it is convenient to convert between our hierarchical blocked representation and traditional COO/CSR (or CSC, Compressed Sparse Column) formats.

2.4.2 Hybrid Graph Processing

Based on our hierarchical blocked graph representation, we design a hybrid *vertex-centric* and *edge-centric* processing model. The basic idea is as follows: we use *vertex-centric* processing to find all *tiles* containing active source vertices for the current frontier, and use *edge-centric* processing to work through these active tiles, updating destination vertices and generating new active frontier for the next iteration.

Algorithm 1 shows more details of the hybrid processing. For each iteration, we main-

tain an *active vertex map* (frontier), indicating which source vertices are active in the current iteration. If it is not empty (line 1), we iterate through the graph hierarchically from group to stripe to tile (line 2 to line 4), in which multiple threads process stripes in parallel in a lock-free manner. If there exists a tile that contains at least one active source vertex (line 5), we work through all edges in this tile by finding active edges, computing destination vertices with source vertices, and generating an *active vertex map* for next iteration (line 6 to line 9). We place a global barrier between two groups' processing, aiming to further improve the inter-thread data locality as aforementioned (line 10). In the end, we prepare the *active vertex map* for the next iteration (line 11).

Algorithm 1: Hyb_Process (*block_graph*, *active_map*)

```
1  while active_map is not empty do
2      forall group ∈ block_graph do
           /* parallel processed by threads                          */
3          forall stripe ∈ group in parallel do
4              forall tile ∈ stripe do
5                  if ∃ vertex ∈ tile.src_vertices is active then
                       /* parallel processed by SIMD                 */
6                      forall edge ∈ tile in parallel do
7                          if edge.src is active then
8                              update_vertex(edge.dest, edge.src)
9                              next_active_map.add(edge.dest)
           /* global barrier between two groups processing           */
10         __synchthreads ()
11     swap(active_map, next_active_map)
```

Figure 2.2 illustrates the hybrid processing of Breadth-First Search (`BFS`) by marking the active edges in different iterations with different colors. In the first iteration, the *active vertex map* contains only one vertex (v_0) and activates three tiles (t_0, t_1, and t_3) in three stripes that belong to the same group. All edges in these tiles are processed, and active ones are colored *purple*. After such processing, seven vertices (v_2, v_4, v_5, v_6, v_7, v_C, and v_D) are marked with active and processed in the second iteration. Four active tiles are belonging to three stripes in two groups in the second iteration (t_0, t_1, t_2, and t_5),

in which, active edges are colored *green*. We keep such processing until all vertices are processed.

Our hybrid graph processing is able to take advantages of both vertex-centric and edge-centric models, *i.e.*, on the one hand, we avoid processing inactive edges by skipping those tiles efficiently (benefits from vertex-centric); on the other hand, we decrease the difficulty of performing a lock-free and load-balanced execution and increase the SIMD efficiency (benefits from edge-centric).

Discussion—*mixed-tile problem and related optimization:* For graph traversal applications, we notice an important performance issue that we may have to unnecessarily run through many inactive edges due to the *edge-centric* tile processing. For instance, in our `BFS` example in Figure 2.2, we need to access two edges when we process t_0 in the first iteration— the active purple one and the inactive green one respectively, although the inactive edge only requires checking its source vertex status. This problem causes duplicated checking for the same edge, incurring non-neglectable overheads. We call it the *mixed-tile problem*.

We identify a significant source of this problem, i.e., there are too few active vertices in a frontier for some iterations. Correspondingly, we leverage an existing optimization to mitigate this problem, i.e., incorporating a push-based execution to our hybrid graph processing similar to Ligra [175]. More details of this optimization will be elaborated in Section 2.5. To the end, we would like to achieve that when a tile is active, most of its edges are active; otherwise, this tile is inactive.

2.4.3 Uniform MIMD-SIMD Scheduler

Based on basic hybrid graph processing, we establish a uniform MIMD-SIMD task scheduler to achieve both good thread load balance and SIMD utilization.

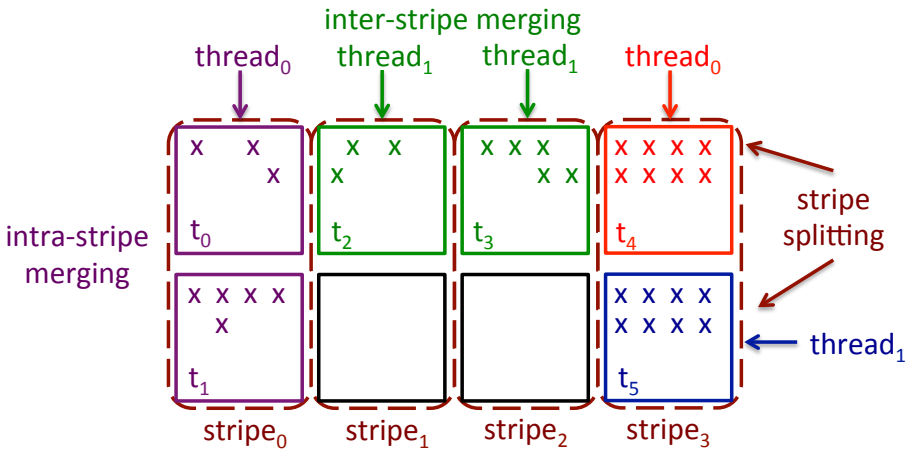


Figure 2.4: A MIMD-SIMD schedule example.

2.4.3.1 Dynamic Conversion of MIMD and SIMD Tasks

Our uniform MIMD-SIMD task scheduler is able to selectively execute stripes either in MIMD+SIMD or MIMD-only to dynamically maintain a high SIMD utilization and MIMD load balance. In particular, we have three execution modes as follows:

- **Basic Execution:** By default, each stripe is assigned to an individual thread with each tile executed in a SIMD manner. Meanwhile, we explore coarse-grained MIMD parallelism among multiple stripes with dynamic scheduling.

- **Stripe Merging:** To maintain a high SIMD utilization, we design two levels of stripe merging—intra-stripe and inter-stripe. If a tile in a stripe contains too few tasks ($< merging\_threshold$), resulting in a low SIMD utilization, an intra-stripe (inter-tile) merging operation happens. Similarly, if a stripe contains too few tasks, an inter-stripe merging is activated to consolidate current stripe with next one, guaranteeing a certain SIMD utilization.

- **Stripe Splitting:** To enhance MIMD load balance, we design a stripe splitting mode as follows. If a stripe has too many tasks ($>> merging\_threshold$) while there exist more than one idle threads, this stripe will be assigned to multiple threads, with a tile as the minimal assignment unit. After stripe splitting, there may exist

update conflicts among threads, so we have to process these tasks in a MIMD-only way with *atomic* update support. Because such execution is inefficient, we restrict our stripe splitting to a condition that more than half threads are idle. This case may only happen if there are any highly skewed inputs and at the end of a group processing.

We show an execution example with all three scheduling modes in Figure 2.4. It contains four stripes ($stripe_0$ to $stripe_3$) in the same group and two MIMD threads ($thread_0$ and $thread_1$). We set the $merging\_threshold$ as 8, i.e., when the number of edges is less than 8, a merging operation will happen to guarantee a good SIMD utilization. In this example, $thread_0$ performs an *intra-stripe merging* while $thread_1$ performs an *inter-stripe merging*. Assume after completing $stripe_0$, $thread_0$ gets $stripe_3$ and finds it has too many edges, a *stripe splitting* operation will happen, in which t_5 is assigned to (stolen by) $thread_1$ to achieve a better thread load balance.

In summary, our uniform MIMD-SIMD execution dynamically toggles among these modes. If we treat a *group* as a 2-D space with the row representing the number of stripes and the column representing the length of stripes, our MIMD execution is a *row-major* (or *row-preferred*) task schedule, i.e., an idle thread seeks for unprocessed tasks in the row direction preferentially. Such design is aimed to maximize the benefit of SIMD execution while minimizing the performance degradation caused by the *atomic* operation involved execution, and also to achieve a MIMD load balance in case the input is highly skewed.

2.4.3.2 Update Conflict Resolving

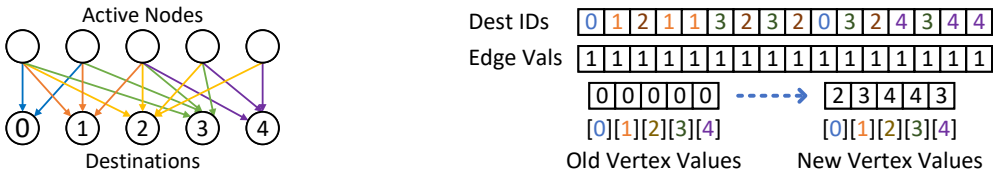
Our MIMD-SIMD scheduler requires resolving possible update conflicts. In MIMD-level, for *basic execution* and *stripe merging* modes, the update conflicts have been handled by the stripe data organization. For *stripe splitting* mode, we need to run a lock-based code for tiles in the same stripe as we mentioned before. In SIMD-level, as opposed to previous efforts based on a heavy data reorganization preprocessing [42], we rely on the built-in

conflicts detection intrinsics provided by Xeon Phi to dynamically address the possible update conflicts.

2.5 Implementation

GraphPhi includes several optimizations in its implementation to support efficient MIMD-SIMD execution.

2.5.1 Dynamic Write-conflicts Processing



(a) Connection. (b) Update vertex by sum edge vals.

Figure 2.5: A write-conflict example.

Our hierarchical blocked data representation naturally resolves the coarse-grained thread-level write conflicts. However, write conflicts still happen in SIMD level when multiple lanes of a SIMD instruction attempt to update the same memory location, Figure 2.5 shows an example, in which multiple edges update the same destination vertices. If one SIMD write performs these updates, write conflicts occur. In particular, every vertex sums up its in-edges' values in this example. If any conflicts happen, the sum result will not be guaranteed.

We handle SIMD-level update conflicts by a dynamic fine-grained solution based on emerging SIMD conflict detection intrinsics. In a SIMD operation, the conflict lanes with the same destination can be grouped. We call it a *conflict group*. For example, we mark such conflict groups by different colors in Figure 2.5b. On the latest Xeon Phi, only the *last value* in a conflict group will be stored into the memory in a SIMD write. Hence, we accumulate all edge values in a conflict group to its last element and then perform a

scatter operation on all conflict groups to update destinations.

We show our implementation in Algorithm 2. At first, it uses AVX-512 conflict detection (CD) intrinsic to detect update conflicts (line 1). If any conflicts exist, we get the position of the *immediately previous conflict* for each conflict starting from the second one in each group (line 3 to line 4). For example, in Figure 2.5b, for conflicts in the *yellow* group with the common destination index of 1, the second yellow 1's *immediately previous conflict* position is the position of the first yellow 1. We maintain an array ($pIDs$) holding such *immediately previous conflict* positions for all conflicts. With this information, it is possible for us to accumulate the update values with common destinations from the first to the last iteratively, beginning with adding the first value to the second (in the position of $pos2$) (line 6 to line 9). Such accumulations among different conflict groups are performed in parallel, and the largest group decides the number of repeated iterations. Eventually, the element at the end of each conflict group holds the cumulative sum of all conflicted updates of that group, to be stored into memory by a SIMD scatter operation (omitted in our algorithm). The conflict detection at line 1, the for-loop at line 3 and line 6 are all implemented in AVX-512 intrinsics. Moreover, we may also apply this algorithm to other update scenarios besides cumulative addition by changing the operation at line 8 accordingly.

2.5.2 Push/Pull Execution

This technique was first proposed by Beamer et al. [155] as a direction-optimization for accelerating BFS, and generalized in Ligra [175] and a later study [26]. Its basic idea is a hybrid execution consisting of a push (top-down) stage, where the vertices in the current frontier explore their neighbors, pushing updates to them and adding unvisited neighbors to the next frontier, and a pull (bottom-up) stage, where the unvisited vertices search for their parents in the active frontier, pulling updates from their active parents and adding themselves to the next frontier.

These two stages show different strengths. The push stage works better when the

Algorithm 2: cumulative_sum (*data*, *indices*)

```
   /* detect if there exist conflicts in indices                          */
1  cd_mask = conflict_detect(indices)
2  if cd_mask not all false then
       /* get the immediately-before conflict position in the same group for
          all conflicts starting from the second one in each conflict group */
3      forall conflict index i ∈ cd_mask do
4          pIDs[i] = indx immediate before i in conflict group
       /* in each group, keep merging the value in the first conflict position
          to the value in the second, and set the first conflict position as
          false until all conflict values merged to the value in the last
          position                                                          */
5      repeat
6          forall conflict group do
7              pos2 = the index of 2nd conflict in this group
8              data[pos2] += data[pIDs[pos2]]
9              cd_mask[pos2] = false
10     until cd_mask all false
```

current frontier is small, while the pull stage works better when the current frontier is large. This is because as the frontier gets larger, there exist too many edges connected to the same unvisited vertex, causing redundant checking or update conflicts when push is used. Usually, such redundancy and conflicts can be avoided from another direction by asking the unvisited vertices to pull the updates. However, the pull approach requires visiting all vertices, which is too expensive when the frontier is small.

Our *Mixed-tile* problem makes our hybrid processing prefer large active frontiers, similar to the bottom-up pull stage. Therefore, to accelerate the small frontier situation for applications like `BFS`, we combine our approach with a top-down push execution like Ligra. Correspondingly, we keep an untiled CSR format of graph data in addition to our hierarchical blocked graph. As shown in section 2.6, we can significantly reduce the unnecessary edge checking with this solution.

2.5.3 High-Bandwidth Memory

Our efficient MIMD-SIMD execution thoroughly explores the massive system parallelism, thus naturally increasing the number of concurrent memory access requests. In many

cases, GraphPhi becomes memory bandwidth bound. As we mentioned before, KNL is equipped with a High-Bandwidth Memory (HBM) with more than $4X$ bandwidth than traditional DDR4. Our GraphPhi is capable of taking advantage of this feature, i.e., allowing users to place the graph (or a portion of the graph) on HBM. This is important because modern memory hierarchies are becoming increasingly heterogeneous.

In our implementation, we configure our machine into a flat mode with DRAM as node 0 and HBM as node 1. We use `numactl` (NUMA control utility) with "`-m 1`" option to bind our application to HBM. However, for some large graphs, 16 GB HBM is not big enough to hold the whole graph. Thus, we also specify a "`-p 1`" option that states a preference for HBM, i.e., part of it will be stored in regular DRAM if the graph is too big.

Table 2.1: User APIs and Pre-defined Functions.

User Defined Functions	**Description**
`bool compute_cond (Edge e, Frontier f)`	Decide if the edge is active in this iteration.
`void compute (Edge e)`	Perform computation for active edges.
`bool update_cond (Vertex v)`	Decide if the vertex is active in next iteration.
Pre-defined Functions	**Description**
`void scheduler (Frontier f, Group g)`	Traverse a group and process its stripes in parallel; run `kernel` for every tile.
`void kernel (Frontier f, Tile t)`	Process edges in a tile according to user-defined `compute_cond` and `compute`
`Frontier update()`	Traverse vertices and mark active vertices for next iteration according to user-defined `update_cond`.

2.5.4 Application Programming Interface

Our GraphPhi framework provides a set of user APIs and pre-defined functions (Table 2.1) to assist users in developing new applications. The pre-defined functions, including `scheduler`, `kernel`, and `update`, are aimed to offer a basic execution skeleton for graph processing, while the user APIs, including `compute_cond`, `compute`, and `update_cond`, are designed for users to specify the applications' computation logic.

In general, the `scheduler` function traverses groups of the input graph. For each group,

it assigns stripes to threads, and each thread calls the `kernel` function to process all the tiles in its assigned stripe. The stripes in a group are processed in parallel. In the `kernel` function, the user needs to specify the user-defined functions `compute_cond` and `compute` to perform specific computations according to the application's requirement. Specifically, the `kernel` function traverses all edges in each tile and runs the `compute` function upon those edges whose return values from `compute_cond` function are true. After the whole graph is processed, the `update` function traverses all vertices and marks them as active for the next iteration if their return values from `update_cond` function are true. It also resets the update condition for all vertices. The iterative process stops if there are no active vertices anymore.

2.6 Experimental Evaluation

In this section, we evaluate the performance of our GraphPhi approach by comparing it with the other two popular graph processing frameworks for shared-memory CPUs, Galois[3] and Ligra[4] on six graph applications and six graph datasets. There are four objectives in our evaluation: first, demonstrating that our GraphPhi outperforms other graph processing systems that are not optimized specifically for Xeon Phi; second, confirming that our uniform MIMD-SIMD execution results in good scalability and SIMD speedup; third, studying some key underlying reasons for our performance benefits, such as the effect of our hierarchical blocking, push and pull optimization, and SIMD utilization improved with stripe merging; and finally, empirically proving that GraphPhi can perform even better by leveraging the High-Bandwidth Memory.

2.6.1 Platform and Benchmarks

Platform: We evaluate our GraphPhi approach on the latest version of Intel Xeon Phi, Knights Landing. It is a 64-core Xeon Phi 7210 processor with up to 256 hyper-threads,

[3] `http://iss.ices.utexas.edu/?p=projects/galois`
[4] `https://github.com/jshun/ligra`

running at 1.30 GHz, supporting efficient 512-bit AVX-512 intrinsics, with 1M L2 cache shared between every two cores. We use it as a CPU host with 96 GB DRAM and 16 GB HBM (MCDRAM) as main memory. We configure the DRAM and HBM in a flat mode.

Benchmark Applications: We evaluate our GraphPhi on six graph applications. These benchmark applications are written in C++ and compiled with the icc-17.0.1 compiler with `-O3`. We use the default setting for prefetching because the `-qopt-prefetch=5` option does not yield noticeable performance difference. Here are more details:

- **Breadth-First Search** (`bfs`) traverses graphs in frontiers and calculates the minimal hop distance from the source to all other vertices. Ligra adopts a push/pull hybrid execution to switch between sparse and dense frontiers. Our implementation follows Ligra and uses the same threshold for the switch. We choose `barrierWithCas` as Galois' algorithm option.

- **PageRank** (`pagerank`) approximates the impact of every vertex by calculating its rank based on its neighbors' ranks. Our implementation accesses all edges in a data-driven manner. All implementations run 1 iteration. Galois uses a *pull* model and requires a weighted graph. We use its graph conversion tool to add weights randomly to edges for our graphs. Ligra and GraphPhi use unweighted graphs.

- **Single-Source Shortest Path** (`sssp`) computes the shortest distance from a source vertex to others. Ligra and Galois implement a frontier-based modified Bellman-Ford algorithm. Our implementation follows Ligra's algorithm. All Ligra, Galois, and GraphPhi require a weighted graph as input. We choose `asyncPP` as Galois' algorithm option.

- **Connected Components** (`cc`) finds a maximal set of vertices reachable from each other. We implement it based on label propagation in GraphChi[5]. We choose `async` as Galois' algorithm option.

[5] `https://github.com/GraphChi/graphchi-cpp/blob/master/example_apps/connectedcomponents.cpp`

Table 2.2: Character and configuration of graphs.

Datasets	# Vertices	# Edges	Tile Width	Stripe Length
Pokec [4]	1.6M	30.6M	8192 (4096)	64 (16)
LiveJournal [3]	4.8M	68.5M	16384 (16384)	64 (128)
RMAT24	16.8M	268.4M	16384 (32768)	128 (256)
RMAT27	134.2M	2.1B	16384 (16384)	4096 (2048)
Twitter [5]	41.7M	1.5B	16384 (32768)	1024 (512)
Friendster [1]	68.3M	2.1B	65536 (131072)	512 (128)

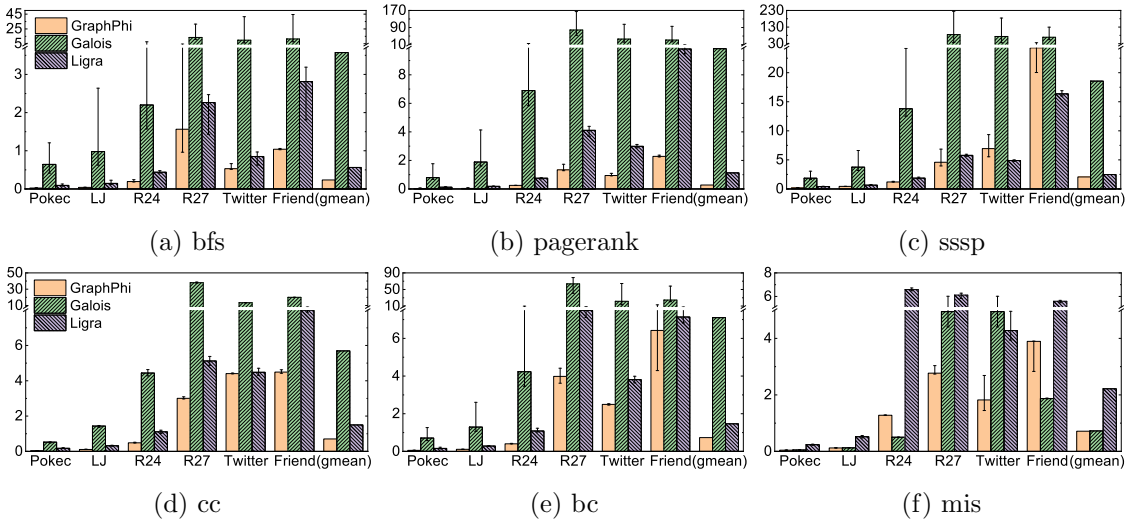
- **Betweenness Centrality** (`bc`) calculates the betweenness centrality index for every vertex. It contains two phases where the first phase is very similar to `bfs` and the second is a reversed traversal of the first phase. Our implementation follows Ligra that the estimated betweenness centrality value is based on only one traverse of BFS. We choose `async` as Galois' algorithm option and set `-t=1`.

- **Maximal Independent Set** (`mis`) finds a maximal set of vertices that form an independent set (and not a subset of any other independent set). Our implementation follows Ligra. We choose `nondet` as Galois' algorithm option.

Input Graph Datasets: We evaluate GraphPhi on six graphs as shown in Table 2.2. They are all downloaded from their website. The synthetic scale-free graphs RMAT24 and RMAT27 were generated from the RMAT generator in Ligra, following the same configuration used by the Graph500 benchmark [2]. The RMAT24 has parameters $a = 0.5, b = c = 0.1, d = 0.3$. The RMAT27 has parameters $a = 0.57, b = c = 0.19, d = 0.05$. We also show graphs' hierarchical block configuration in Table 2.2 where the number in parenthesis is for `pagerank` and another one is for all other applications. Particularly, for `sssp`, we use randomly weighted edges, a tile width of 65536, and a stripe length of 128.

2.6.2 Overall Performance

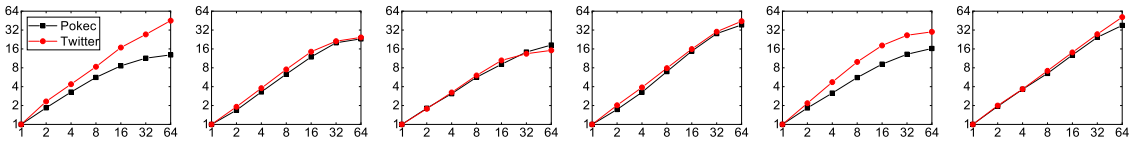
Figure 2.6 shows the overall performance of GraphPhi compared to Galois and Ligra on all benchmarks and input graphs. We run all tests for 10 times with 64 threads in parallel. We report minimum, maximum, and average execution time. We also report the geometric



(a) bfs (b) pagerank (c) sssp

(d) cc (e) bc (f) mis

Figure 2.6: Overall performance. x-axis: graph datasets; y-axis: execution time (s).

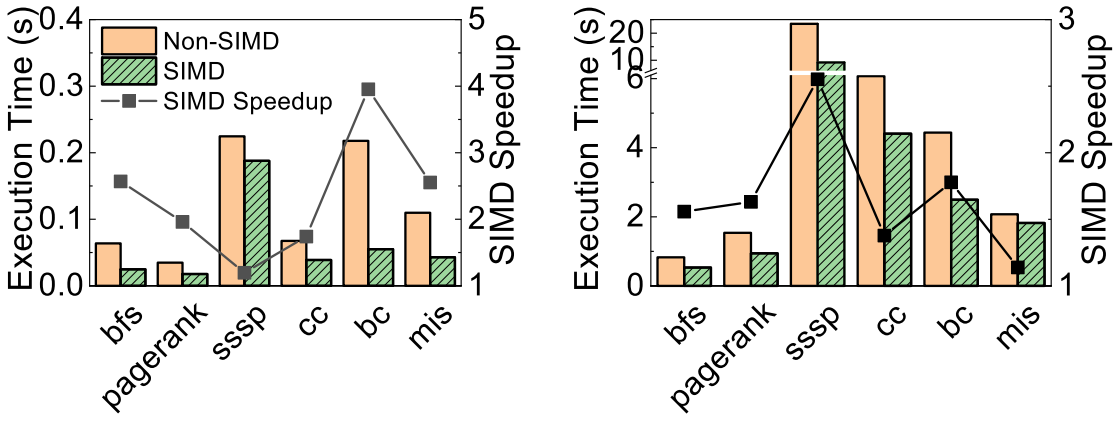


(a) bfs (b) pagerank (c) sssp (d) cc (e) bc (f) mis

Figure 2.7: Scalability. x-axis: number of threads; y-axis: speedup over 1-thread.

mean (`gmean`) of the average execution time for each benchmark on all input graphs. In addition, to ensure a fair comparison, all tests are performed on DRAM only. We will perform an extra performance study for HBM later.

GraphPhi outperforms Galois and Ligra for most cases, except `mis` on `R24` (an abbreviation for `RMAT24`) and `Friend` (an abbreviation for `Friendster`), when Galois works better than GraphPhi. For `pagerank`, GraphPhi shows the best *geometric mean* speedup over Galois and Ligra ($35.4X$ and $4.0X$, respectively), because our `pagerank` implementation traverses all edges tile by tile without any redundant accesses, benefiting most from both data locality improvement and SIMD execution. For other applications, GraphPhi's geometric mean speedups over Galois range from $1.2X$ to $16.5X$, and over Ligra range from $1.2X$ to $3.4X$. In terms of different datasets, GraphPhi obtains average speedups over Galois from $5.8X$ to $19.3X$, and over Ligra from $1.6X$ to $4.3X$.



(a) Pokec (b) Twitter

Figure 2.8: Performance: non-SIMD vs SIMD.

2.6.3 MIMD-SIMD Scheduler Efficacy

To further study the efficacy of our MIMD-SIMD scheduler, we report our scalability and SIMD speedup results in Figure 2.7 and Figure 2.8, respectively. Constraint by our space, we only show results on two representative graphs; one is small (`Pokec`), and another one is large (`Twitter`). All tests were run 10 times, too; however, we only show the average execution time to make the figures more readable.

2.6.3.1 Scalability

Figure 2.7 illustrates that all our six benchmarks scale well on both `Pokec` and `Twitter`. Particularly, tests on `Twitter` exhibit better scalability than on `Pokec`, because `Pokec` has fewer workloads, rendering the parallel execution overhead more noticeable. Moreover, some benchmarks cannot scale perfectly from 32 threads to 64, e.g., `pagerank` and `bc`. This is because they are increasingly memory bandwidth bound as the concurrency increases since these tests are on DRAM. Later tests further prove that these applications are more likely to benefit from the High-Bandwidth Memory.

2.6.3.2 SIMD Speedup

Figure 2.8 compares the performance of SIMD codes and non-SIMD codes with 64 threads and reports the speedup of SIMD over non-SIMD. For all six benchmarks, SIMD execution results in an average speedup of 2.3 and 1.7 on `Pokec` and `Twitter`, respectively. We do notice that the execution time fluctuates more for the relatively small graphs like `Pokec`, while running large graphs like `Twitter` produces more stable speedups. For different kinds of applications, the speedup stems from different reasons. On the one hand, most topology-driven graph applications like `bfs` and `bc` are memory latency bound with MIMD-only execution, and an extra SIMD execution is able to increase the number of concurrent memory access requests, thus hiding memory latency. We confirm this with a memory throughput comparison test that is omitted due to our space constraint. The later HBM study can prove this from another perspective. On the other hand, data-driven graph applications like `pagerank` are computation bound with MIMD-only execution, and SIMD execution can accelerate their kernel computations.

2.6.4 Understanding the Performance

We now explore several important optimizations that significantly affect our performance.

2.6.4.1 Effect of Hierarchical Blocking

Our hierarchical blocking, specifically, the design of partitioning a graph into multiple groups, is aimed to improve both intra- and inter-thread data locality. However, the introduction of groups also requires additional global barriers, thus incurring synchronization overhead. Therefore, the overall performance stems from a combination of both data locality and synchronization overhead. We show the study on `pagerank` application running on the `Twitter` graph (tile width 4096) with 64 threads and report the results in Figure 2.9. This study shows that as we increase the group size (stripe length), the L2 miss rate will increase while the synchronization overhead will decrease. The former is

because of the increasingly worse data locality, while the latter is owing to the decrease of global barrier counts. Eventually, our best performance is a trade-off between these two factors and achieved with the tile width of 64.

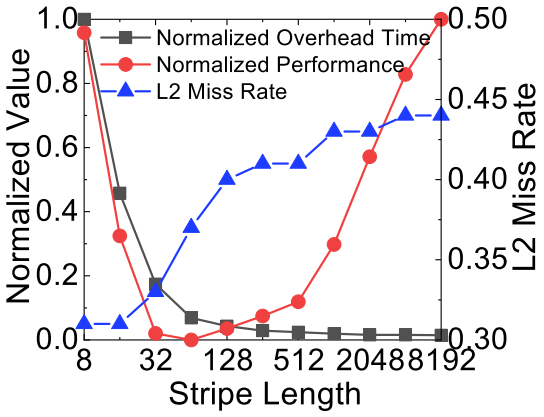


Figure 2.9: Trade-off of cache and synchronize.

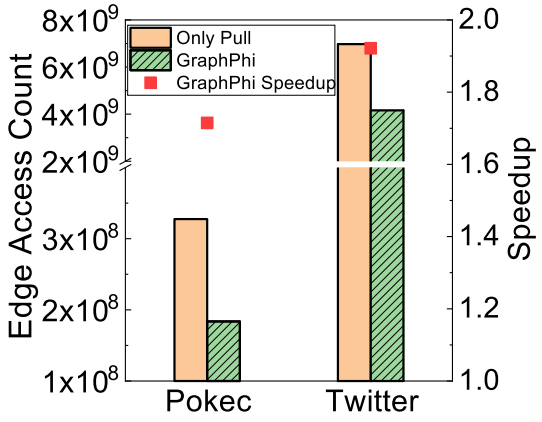
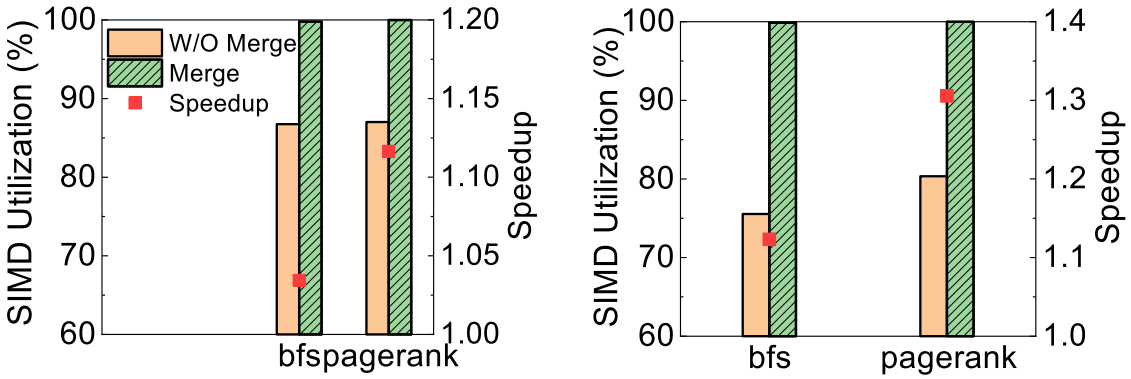


Figure 2.10: Pull-only and push/pull comparison.

2.6.4.2 Effect of Push-based Execution

The aforementioned *mixed-tile* issue significantly affects our overall performance for topology-driven applications like `bfs`, thus we also evaluate the efficacy of our *push-based* execution. Our evaluation is performed on `bfs` application with both `Pokec` and `Twitter` graphs, running with 64 threads. We compare both the number of accessed edges and the execution time between a pull-only version (`Only Pull`) and a hybrid push/pull version (`GraphPhi`) of GraphPhi, and report the results in Figure 2.10. The results demonstrate that with *push-based* execution, GraphPhi is able to reduce around 43% and 40% total edge processing, resulting in $1.7X$ and $1.9X$ speedup, respectively.

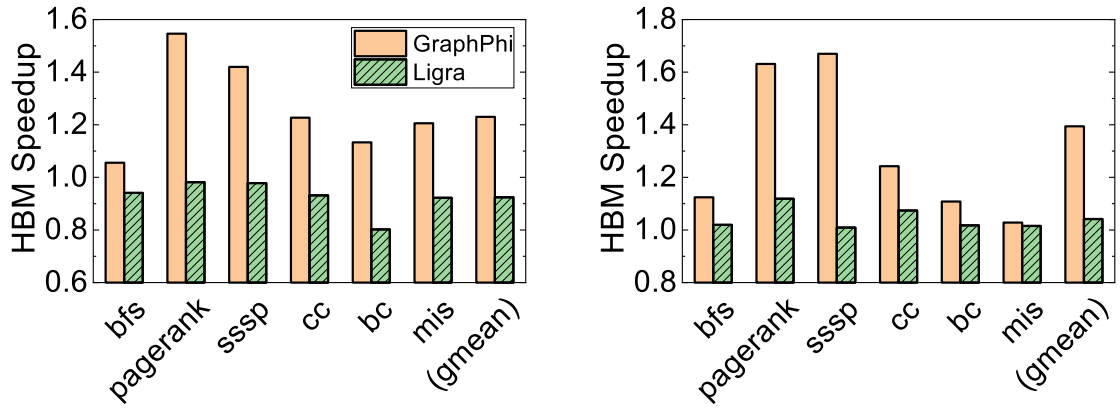


(a) Pokec. (b) Twitter.

Figure 2.11: SIMD utilization: merge vs w/o merge.

2.6.4.3 SIMD Utilization Study

We also perform a SIMD utilization[6] study to help our understanding of the efficacy of the *stripe merging* execution in our scheduler. We evaluate two representative applications, `bfs` and `pagerank` with `Pokec` and `Twitter` graphs, and compare the SIMD utilization before and after this optimization. The result in Figure 2.11 shows that *stripe merging* optimization is able to improve the SIMD utilization by up to 25% and provide a speedup up to $1.3X$.



(a) Pokec. (b) Twitter.

Figure 2.12: HBM speedup for GraphPhi and Ligra.

[6]SIMD utilization is defined as the ratio of SIMD executed workloads over all workloads.

2.6.5 Extra HBM Benefit

Previous studies are all performed on DRAM. We also test GraphPhi on HBM and compare the performance with Ligra. This evaluation is conducted on all benchmarks with `Pokec` and `Twitter` graphs. We use 128 threads (i.e., 2 hyper threads per core) because each KNL core has 2 VPUs, and 128 threads result in the best performance for both Ligra and GraphPhi. We calculate the speedups of GraphPhi and Ligra with HBM configuration over without HBM and report them in Figure 2.12. While the execution time fluctuates for `Pokec`, a relatively small graph, the running on `Twitter` produces a more stable performance. For GraphPhi, the use of HBM yields additional $1.2X$ and $1.45X$ geometric mean speedups on these two graphs, respectively; while for Ligra, it results is either slightly slowdown or unnoticeable speedup. The reason is as follows. Ligra implementation does not seek the help of SIMD. Thus, the memory stall still dominates the overall execution time for these applications. However, Xeon Phi HBM only optimizes the memory bandwidth, and its memory latency is similar to or even worse than DRAM. Alternatively, our GraphPhi leverages extra SIMD parallelism to hide memory latency by more concurrent memory requests (like GPUs), thus more thirsty to memory bandwidth. Such results empirically prove that our GraphPhi is more suitable for future HBM architectures.

2.7 Related Work

Due to the significant importance of graph analytics, there have been many efforts to efficiently parallelize graph processing on modern multi-core or many-core architectures in recent years. We describe some of them closely related to our work.

Graph processing on CPUs: There exist many popular graph processing engines and frameworks on CPUs nowadays, however, they are designed for different scenarios, thus facing very different problems. Inspired by the BSP model [190], Google Pregel [136] was proposed as the first *vertex-centric* graph processing model mainly for large-scale

distributed clusters. Such a model has been adopted by many other distributed graph processing engines since then, such as GraphLab [133] and PowerGraph [78]. The primary challenge for these distributed graph processing systems is how to efficiently partition graphs, store partitions on multiple machines, and perform low-cost communications. Out-of-core graph execution engines, such as GraphChi [115] and X-Stream [166], focus on reducing disk traffic when processing large-scale graphs which do not fit in the main memory of a single-machine. In-memory single-machine graph processing frameworks, such as Polymer [207], Galois [148], and Ligra [175], are similar to GraphPhi. Polymer focuses on optimizing graph processing on multi-CPU NUMA machines rather than our platform. Although Galois and Ligra either offer more general APIs or perform an efficient hybrid execution on CPU platforms, they do not mainly match the Xeon Phi-like architectures, either, because their current design does not consider the emerging hardware features, such as wide SIMD units and the HBM. In contrast, GraphPhi is carefully designed and implemented to take advantage of those features.

Graph processing on GPU and Xeon Phi: There are also many graph processing frameworks on GPUs [143, 214, 109, 172, 194, 152, 86, 90, 40]. However, they also concern different problems compared to our work. For example, efforts like GraphReduce [172] and Graphie [86] aim at reducing CPU-GPU traffic for the processing of large graphs which do not fit in the GPU memory, while works like CuSha [109] and Gunrock [194] optimize for load balance and memory coalescing. GraphPhi focuses on a different throughput-oriented architecture and explores many unique features that are not shown on GPUs. There are also some optimization techniques for Xeon Phi [42, 100, 25]. Although they comprehensively explore advanced SIMD execution, none of them offer a general graph processing framework by effectively exploiting both MIMD and SIMD execution, or emerging HBM techniques. For example, Chen et al.'s effort [42] requires a relatively heavy preprocessing to resolve update conflicts and includes a basic MIMD scheduling method that groups all tiles in the same column together, while our work dynamically resolves conflicts through careful data organization and computation schedule with an optimized

MIMD-SIMD scheduling technique. In addition, several graph processing frameworks are designed for hybrid CPU and coprocessors [91, 76, 41, 135], but their main focus is on workload partition instead of exploiting the SIMD units. In particular, Chen et al. [41] employ a vertex-centric message passing model and resolves update conflicts by a costly reordering process. Mosaic [135], a heterogeneous processing engine uses both the CPU and multiple Xeon Phi cards as co-processors to perform graph computation without supporting SIMD execution, while our work focuses on a single Xeon Phi card used as the host processor with SIMD support.

Specific graph algorithms and other important algorithms: Besides these general graph processing systems, there are also many efforts on parallelizing specific graph applications or graph related operations on modern parallel architectures that are related to our work, *e.g.*, BFS [143, 155, 131, 129], Connected-Components [179], Betweenness Centrality [140], Single Source Shortest Path [55], and SpMV [201, 128]. In contrast, our goal is to provide a more general graph processing framework on an emerging throughput-oriented architecture. There also exist some research efforts using Xeon Phi for algorithms other than graph processing, such as FFT [153], an important scientific computing kernel. However, scaling FFT and graph computation on KNL are different. The challenge for scaling graph computation arises from irregular parallelism and memory accesses.

2.8 Chapter Summary

This work presented GraphPhi, a new optimization framework to process graphs efficiently on Xeon Phi architectures. It consists of an optimized graph representation, a hybrid vertex-centric and edge-centric execution design, and an efficient MIMD-SIMD scheduler with lock-free update support. Inherited from edge-centric processing, GraphPhi may process redundant edges. However, compensated by our advanced SIMD acceleration together with a push-based execution for sparse frontiers, GraphPhi produces better performance than state-of-the-art graph execution frameworks, such as Galois and Ligra. In addition,

we showed that GraphPhi, by efficiently utilize the SIMD units, converts latency-bound graph applications into bandwidth-bounded ones, hence taking advantage of the HBM.

Chapter 3

Parallelizing Pruned Landmark Labeling: Dealing with Dependencies in Graph Algorithms

3.1 Introduction

Computing the shortest path distance between any two vertices stands out as one of the most fundamental graph operations, with applications ranging from transportation systems (for distance related navigation) [59], social networks/WWW/semantic web (for recommendations and ranking) [37], to knowledge graphs (for concept detection) [174], among others. This operation also serves as the basis for more complex graph analytics and mining operations, such as graph pattern matching [47, 217], distance join processing [169], and centrality computation [28].

However, computing shortest path distances over scale-free complex networks (e.g., massive social and web graphs) remains a challenging problem [9, 103, 151]. To help answer shortest path distance queries on demand, the 2-*hop labeling approach* [48] has

emerged as an effective tool. Given a graph, it aims to assign each vertex v a label set $L(v)$, which comprises a list of vertices and their distances to v. Subsequently, given any two vertices u and v, we only need to use their respective label information, $L(u)$ and $L(v)$, to rapidly compute their exact distance.

This approach was made scalable by the *Pruned Landmark Labeling (PLL)* method [9]. This labeling approach adopts a fast greedy process to iteratively consider one vertex at a time (according to certain vertex order) and potentially assigns it to the label sets of other vertices, using a *distance check criterion*. Once the labeling process is done, the results are guaranteed to be *minimum* (or *canonical*) with respect to the given vertex order. In the past few years, a number of studies [57, 123, 161, 8] have further validated and confirmed the scalability of this approach.

Parallel PLL: The original PLL algorithm is inherently sequential; i.e., the algorithm operates one vertex at a time to label the entire graph, and the labeling of a vertex depends on the partial labeling results from earlier processed vertices. In view of this, on the one hand, the original PLL [9] suggested to simply parallelize the BFS labeling of each vertex instead of dealing with inter-vertex labeling dependency, thus severely limiting parallelism. On the other hand, two recent attempts [65, 162] allow multiple vertices to be simultaneously processed, but do not produce the same compact label as the original PLL.

This work studies how PLL can be parallelized in a scalable and exact fashion. We make the following **contributions**:

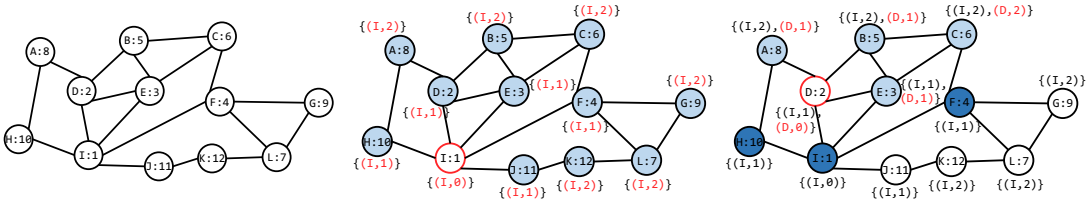
- **Linear Algebra Formulation (Section 3.2):**. Motivated by developments like GraphBLAS(T) [107] and GraphMat [189] that support graph computations based on a set of linear algebra operators, we show how the original PLL algorithm can be expressed as a series of matrix (and vector) computations. However, we also observe that because of a *masking* operation, the more efficient implementation will be the one based on a vertex-centric approach.

- **Parallel PLL Algorithm (Section 3.3):** To solve the mismatch between the inherent sequential/dependence of PLL algorithms and the goal of allowing independent operations on each vertex, we present a theoretical result and use it to develop a new VC-PLL algorithm that utilizes VC to parallelize PLL and is guaranteed to produce the same labels as original PLL.

- **Batched Vertex-Centric PLL (Section 3.4):** To deal with the limitations of VC-PLL, we introduce a batched VC-PLL (BVC-PLL) algorithm that largely preserves the same vertex computation function while reducing the costs of message (label) broadcasting and remote memory access.

- **Efficient Parallelization and Generalization (Section 3.5):** We combine intra-node (shared memory) and inter-node (distributed memory) parallelism, resulting in an implementation that can handle graphs with more than 1 billion edges. In addition, we show how BVC-PLL can be extended to handle directed graphs and weighted graphs, and how the extension for weighted graphs can benefit from SIMD parallelism.

In our **experimental study (Section 3.6)**, we show that the sequential BVC-PLL can run more than 2× faster than the original PLL (both using one single thread). The parallel BVC-PLL also demonstrates good scalability and obtains an average speedup of 6.68× over sequential BVC-PLL on a 20-core shared memory machine and a speedup up to 11.85× on a 16-node distributed cluster over 1-node version. We also demonstrate that the shared memory and distributed memory combined BVC-PLL gains good scalability for large graphs with over 1 billion edges. We finally extend BVC-PLL to process weighted graph with the help of SIMD parallelism, achieving up to 1.92× speedup over PLL (both with a single thread), and achieving up to 15.68× speedup in going from 1 to 20 threads.



(a) Original graph with ranks. (b) L after vertex I spreading. (c) L after vertex D spreading.

Figure 3.1: 2-Hop Labeling and PLL Example: spreading vertex is marked with red; light shadow vertices have already received the label spread (colored in red); pruning happens in dark shadow vertices; white vertices are not accessed by the vertex being spread because of pruning.

3.2 2-hop Labeling and PLL

This section further describes the 2-hop labeling problem and the PLL algorithm that forms the basis for our work on parallelization. We also describe how this algorithm can be viewed as a series of linear algebra operations.

3.2.1 2-Hop Labeling

The 2-hop labeling algorithm [48], which was pioneered by Cohen *et al.* [48], provides an efficient scheme to answer on-demand shortest distance queries. It assigns each vertex u in an (undirected) graph a label set $L(u)$ such that for any two vertices u and v, their distance can be computed using only their respective label sets. Formally, we compute $L(u)$ and for each $h \in L(u)$, the corresponding distance from u, i.e, $d(h, u)$. Table 3.1 illustrates a 2-hop labeling of the undirected graph G that is shown in Figure 3.1a.

Formally, the shortest path distance query $Dis(\cdot, \cdot)$ between any two vertices u and v can be answered as:

$$Dis(u, v) = \min_{h \in L(u) \cap L(v)} \{d(u, h) + d(h, v)\}$$

Thus, 2-hop labeling can answer distance queries efficiently by traversing two lists of vertices, with an operation similar to merge sort. As an example, in Table 3.1, the distance

between nodes A and B can be computed as 2 by first identifying that D and I are common vertices in their label sets, and subsequently that the distance through D is the shortest (1+1).

Over a decade, numerous efforts [170, 45, 101, 46, 7, 74, 168] largely failed in making 2-hop labeling practical on large real-world graphs. The pruned landmark labeling algorithms [9, 104] are considered a major breakthrough in solving this problem and are the focus of our work.

3.2.2 Complete Definition of Hierarchical Hub Labeling (HHL) and Canonical Hierarchical Hub Labeling (CHHL)

An important direction to make 2-hop labeling feasible and scalable for a large graph is to restrict the choices of labeling (by imposing some special properties on what can be added to the labels).

Definition 1 *(Hierarchical Hub Labeling) Given two distinct vertices u and v, we say $u \succeq v$ if $u \in L(v)$ (u is a hub of v). A hub (2-hop) labeling is* hierarchical *if $\succeq$ forms a partial order.*

In fact, any partial order can be extended to a total order (the order-extension principle) and for a set of vertices V, the total order is defined as a bijection $\pi : V \to 1, \cdots, |V|$ ($\pi(v)$ is the rank of v). Given this, we can say that a label is hierarchical if there is a total order π which satisfies: $u \in L(v)$ then $\pi(u) < \pi(v)$ (u ranks higher than v).

Definition 2 *(Canonical Hierarchical Hub Labeling) Let the shortest path vertex set P_{uv} consist of all vertices on shortest paths between u and v (including u and v). Given a total order π on V, its canonical hub labeling is defined as follows: $u \in L(v)$ if u has the highest order in P_{uv}, i.e., no other vertex w in P_{uv} such that $\pi(w) < \pi(u)$.*

An important implication of canonical hierarchical hub labeling is that *it produces the minimal hierarchical hub labeling* for a given order [19]. Thus, the optimal HHL problem

can be transformed into two sub-problems: 1) finding the optimal order that minimizes the label size; 2) computing the canonical HHL with respect to a given vertex order.

A main breakthrough enabling efficient 2-hop labeling is the discovery of a simple, yet elegant algorithm called pruned landmark labeling (PLL) [9]. It computes the canonical HHL (the second subproblem) for a given vertex order efficiently. Independently, essentially the same style algorithm was discovered for 2-hop reachability labeling, and is called *distribution labeling* [104]. In the past few years, a number of studies [57, 123] have further validated and confirmed the efficiency and effectiveness of PLL style algorithms for distance labeling.

Theoretically, the optimal hierarchical hub labeling (HHL) as well as the original 2-hop labeling have recently been proved to be NP-hard [19], which implies that the optimal order sub-problem (the first sub-problem listed above) is NP-hard as well. A few heuristics, such as the ranking by degree and betweenness, have been developed for addressing this sub-problem [123]. The second sub-problem (labeling generation) typically dominates the overall labeling computation and is thus the focus of this study.

3.2.3 Pruned Landmark Labeling (PLL)

The main idea of this algorithm is to use a total ordering of all vertices (finding such an order optimally is NP-hard [197], but heuristics are feasible) for labeling. Given such a total order π of vertices, the pruned landmark labeling (PLL) [9] assigns each vertex, based on the order $(\pi(v_1) < \pi(v_2) < \cdots < \pi(v_n))$, to the labels of other vertices in the graph following a BFS process. Note that when $\pi(v_i) < \pi(v_j)$, we say that v_i has the higher rank. As PLL assigns the vertex u with the rank $\pi(u)$ as a label to a lower ranked vertex v, it needs to check if u is the highest ranked vertex in the shortest paths between u and v (P_{uv}). This can be done by checking

$$d(u, v) < d(v, h) + d(h, u), \text{ for all } h \in L(u) \cap L(v).$$

Algorithm 3: PLL for $G = (V, E)$ with Order π

```
1 forall u ∈ V do /* following order π from high to low                    */
    /* BFS process to use u for labeling                                   */
2   Queue Q = {(u, 0)}
3   while Q is not empty do
4     (v, d(u, v)) ← Q.pop()
5     if d(u, v) < min_{h∈L(u)∩L(v)}{d(u, h) + d(h, v)} then
6       Add (u, d(u, v)) into L(v)
7       forall v' ∈ v's neighbor do
8         if v' is unvisited by u and π(u) < π(v') then
9           Add (v', d(u, v) + 1) to Q
```

Table 3.1: 2-hop labeling for Graph G.

Vertex	Labels
A	$\{(A, 0), (D, 1), (I, 2)\}$
B	$\{(B, 0), (D, 1), (E, 1), (F, 2), (I, 2)\}$
C	$\{(C, 0), (B, 1), (E, 1), (F, 1), (I, 2), (D, 2)\}$
D	$\{(D, 0), (I, 1)\}$
E	$\{(E, 0), (D, 1), (I, 1)\}$
F	$\{(F, 0), (I, 1)\}$
G	$\{(G, 0), (F, 1), (L, 1), (I, 2)\}$
H	$\{(H, 0), (A, 1), (I, 1)\}$
I	$\{(I, 0)\}$
J	$\{(J, 0), (I, 1), (L, 2)\}$
K	$\{(K, 0), (J, 1), (L, 1), (F, 2), (I, 2)\}$
L	$\{(L, 0), (F, 1), (I, 2)\}$

Intuitively, this is ensuring that the distance between u and v cannot be recovered by a certain higher ranked vertex. When the condition above does not hold, u will be *pruned* by v (i.e., is not added into the label of v and will not be further expanded from v) during the labeling process.

Figure 3.1 illustrates the processing associated with the two highest ranked vertices (I and E) for the graph in Figure 3.1a. Rank (or order) of each vertex is explicitly shown following the vertex ID.

Algorithm 3 sketches the labeling process for an undirected graph. Note that $d(u, v)$ in

the algorithm is the distance computed by the BFS process, which may not be the exact distance between u and v (due to the pruning effect). But the recorded distance in the label (Line 6) is always exact (since it can travel through all the shortest paths starting from u reaching to v). It can be proved that the results are guaranteed to be *minimum* (or *canonical*) with respect to the given vertex order.

3.2.4 A Linear Algebra View of PLL

It has been well known that a large number of graph algorithms can be stated as operations on (sparse) matrices [108, 75]. Multiplication of a sparse matrix with a vector (SpMV) and multiplication of two sparse matrices (SpGEMM) are the operations most commonly used. Multiple recent efforts have focused on using the work on optimizing (and parallelizing) SpMV and SpGEMM operations as the basis for graph processing [107, 189].

We have examined how PLL can also be viewed as a series of linear algebra operations. Our examination, however, shows that PLL requires more than SpMV and SpGEMM and therefore is better parallelized by taking a vertex-centric view of the computations.

Let A be the adjacency matrix for graph G, where $A[u, v] = Dis(u, v)$. Let I be the identity matrix and let I_i be the i-th column of identity matrix I.

First, it is well known (see, for example [108]) that the following equation computes the shortest distances from a given vertex i, denoted as a vector $\vec{y_i}$:

$$\vec{y_i} = (I_i^T(I + A + A^2 + \cdots + A^d))^T = (I + A + A^2 + \cdots + A^d)^T I_i$$

where d is the diameter of the graph. We also denote

$$A^* = (I + A + A^2 + \cdots + A^d)^T.$$

Note that for any matrix M, M_i is the i-th column of M ($M_i = MI_i$).

Now, we represent the original PLL algorithm as a series of linear algebra operations.

Following the order used in the PLL algorithm, let $\vec{x}_i$ be the vector recording the distance of all vertices to the i-th ordered vertex in the distance labeling:

$$\vec{x}_i[j] = d(j,i), v_i \in L(v_j); \quad \vec{x}_i[j] = +\infty, v_i \notin L(v_j).$$

Let $\ominus$ be the *generalized element-wise masking*: $\vec{x}[i] \ominus \vec{y}[i] = \infty$ if $\vec{x}[i]$ is larger than or equal to $\vec{y}[i]$ and $\vec{x}[i] \ominus \vec{y}[i] = \vec{x}[i]$ if $\vec{x}[i]$ is smaller than $\vec{y}[i]$. Given this, for two column vectors $\vec{x} = \{x_1, \cdots, x_n\}^T$ and $\vec{y} = \{y_1, \cdots, y_n\}^T$, $\vec{x} \ominus \vec{y} = \{x_1 \ominus y_1, \cdots, x_n \ominus y_1\}^T$. Then the labeling PLL essentially utilizes the following equation to assign the labeling:

$$\begin{array}{lcl} \vec{x}_1 & = & A^* I_1 \\ \vec{x}_2 & = & A^* I_2 \ominus (\vec{x}_1 \vec{x}_1^T) I_2 \\ \vec{x}_3 & = & A^* I_3 \ominus (\vec{x}_1 \vec{x}_1^T + \vec{x}_2 \vec{x}_2^T) I_3 \\ \cdots & & \\ \vec{x}_n & = & A^* I_n \ominus (\sum_{i=1}^{n-1} \vec{x}_i \vec{x}_i^T) I_n \end{array}$$

Here, $\vec{x}_i \vec{x}_i^T$ generates the matrix recording the shortest distance between any two vertices via vertex i. Thus, $(\vec{x}_i \vec{x}_i^T) I_j$ corresponds to the shortest distance from any vertex to vertex j via vertex i.

In Algorithm 3, line 2–9 basically provides an efficient procedure to generate the distance label $\vec{x}_i$. Especially, Line 5 can be considered as the on-demand implementation of the generalized masking operation (without explicitly producing the masking vector, and testing the condition as needed).

As we can see above, though we have been able to map PLL to a set of linear algebra operations, the formulation involves more than SpMV and SpGEMM algorithms. Particularly, the use of linear algebra libraries will require materialization of the expression $(\sum_{i=1}^{k-1} \vec{x}_i \vec{x}_i^T) I_k$ during the k-th step, which can be much more expensive as compared to performing the masking operation (or pruning) on demand in Line 5 of Algorithm 3. As a result, we examine other models for parallelizing PLL.

3.3 Parallelization of PLL

This section first gives background on vertex-centric model. It then states the challenges in parallelizing PLL using this model. We state an important theoretical result and then proceed to develop an initial (basic) parallel algorithm for PLL.

3.3.1 Vertex-Centric (and Other) Models

Graph algorithms have been frequently parallelized by thinking of independent computations on each vertex. This was the basis for the seminal vertex-centric programming model proposed by the Pregel paper [137], and many other parallel graph processing system research efforts [78, 115, 148, 175, 133, 109, 207, 172, 194, 152, 135, 86, 53]. Though other models have been used, including the recent projects that use (sparse) linear algebra operations (for example, GraphBLAS(T) [107] and GraphMat [189]), we find vertex-centric model to be a good fit for approaching parallelization of PLL.

In the vertex-centric model, parallel graph processing is viewed as an iterative process, where each iteration processes the set of *active* vertices. For each vertex in this set, we perform computations based on the data from incoming and/or outgoing edges together with the local vertex data, and then update the values/state associated with the vertex. The vertices that record a change in their local state become the active vertices for the next iteration. The parallelization typically uses Bulk Synchronous Parallel (BSP) execution [190] and requires a global synchronization at the end of each iteration. The entire process terminates once the set of active vertices becomes empty.

A high level abstraction of the vertex-centric computation based on a scatter-gather model [137, 166] is sketched in Algorithm 4. Each vertex computation is described through two functions: 1) the *Scatter* function, which describes how each vertex uses its vertex value and edge value to propagate a message to its neighbors; and 2) *Gather* function, which describes how each vertex computes a new value based on its original value and all the new messages it received.

Algorithm 4: Vertex-Centric (Scatter-Gather) (G=(V,E))

```
1 Initialize ActiveVertices ⊆ V
2 while ActiveVertices is not empty do
      /* Scatter Phase:                                          */
3     forall a ∈ ActiveVertices do
          /* for each edge e = (a, v) of a, send message(a, e, v) to v   */
4         a.Scatter(a.edges)
5     ActiveVertices ← ∅
      /* Gather Phase:                                           */
6     forall v Received Message do
          /* vertex compute using received messages and update its value
             */
7         v.Gather(v.messages)
8         ActiveVertices ← {v : v.value is updated}
```

Various more advanced parallel graph programming models are proposed to further refine the vertex-centric model. This includes GAS (Gather-Apply-Scatter) [78] and push and pull models [155, 175, 26], where the goal is to better fit the computational and communication patterns of graph processing. There is also work on generalizing the model to finer granularity, such as the edge-centric model [166], or to coarser granularity, such as path- or subgraph-centric [163], and k-step neighborhood [33, 111] models.

Finally, as we stated earlier, there has been recent interest in the use of linear algebra libraries (and thus using existing methods for parallelizing them). However, because of the masking (or pruning) operation in PLL, a linear algebra based implementation is unlikely to be efficient.

In this work, we focus our efforts on the vertex-centric model, with resulting code implemented efficiently through the use of MPI and OpenMP. Exploration of the use of other models and whether there can be a programmability or performance benefit is a subject for future work.

3.3.2 Vertex-Centric Approach and PLL

Recall that the PLL algorithm (Alg. 3) iterates following the vertex rank (order): at the i^{th} round, the vertex u with rank $\pi(u) = i$ will be distributed to all other vertices in the graph using a BFS process. The key condition to add u into the label of v is that the distance between u and v cannot be recovered by earlier processed vertices. The main challenge in parallelizing PLL is that adding a vertex u of rank $\pi(u)$ to another vertex v in the BFS traversal seems to be *dependent* on the completion of labeling of all higher ranked vertices in order to apply the distance check. In comparison, for parallelization with the vertex-centric model, we would like to distribute all vertices to their neighbors simultaneously for vertex labeling. This requirement seems to be in conflict as there is no guarantee that the higher vertices can finish the distribution before the lower-rank ones. Indeed, as we mentioned earlier, all the existing attempts have all failed to parallelize inter-vertex labeling while preserving the canonical labeling criterion [9, 65, 162].

We address this problem through the following important result.

Theorem 1 *Assume we spread all vertices simultaneously into the graph (starting by sending each vertex to its neighbors), and we do the spreading iteration by iteration following the vertex-centric programming model. Let us consider a vertex u with the rank $\pi(u)$ that reaches the vertex v at the j-th iteration. Then if there is a vertex w with the following properties: 1) with a higher rank than u ($\pi(w) < \pi(u)$); 2) with a shorter distance to v and u ($d(w, v) < d(u, v)$ and $d(w, u) < d(u, v)$), and 3) being recorded as a label of v ($w \in L(v)$) and u ($w \in L(u)$), then, w must be able to reach both u and v before the j-th iteration ($d(u, v)$ steps).*

Proof Sketch: We first note that conditions 1 and 3 ensure w cannot be pruned by other vertices with higher ranks between w and u (and v). Then vertex w can reach u and v in less than j iterations as $d(w, u) < d(u, v) = j$ and $d(w, v) < d(u, v) = j$. (By way of contradiction, if we assume w cannot reach u (or v) in j iterations, it either has a distance longer than j or is pruned, i.e, there is another vertex w' with a higher rank and located

on the shortest path between w and u. In the latter case, w cannot be recorded as a label in u (or v).) $\square$

The Theorem implies that even when the spreading process is parallelized across the node, we can correctly determine if u should be added to $L(v)$ by testing if there is any other vertex, say w, with a higher rank than u ($\pi(w) < \pi(u)$), which can produce an equal or shorter distance, i.e., $d(u,v) \geq d(u,w) + d(w,v)$.

Recall that the distance check condition for canonical labeling criterion requires not only the labeling of higher ranked vertices h to be completed before the distance check between u and v, but also their distances $d(u,h)$ and $d(h,v)$ to be smaller than $d(u,v)$. The latter condition is the key to the parallelization of PLL.

The main result above can also be stated (or derived) through the linear algebra formulation. Recall from the last section that the original PLL computes the distance labels one vertex at a time $(\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_n)$. To parallelize it, we will generate labels for all vertices with the same distance in the same batch. The details of the process are as follows. At the iteration 0 (initialization): for any vertex i, let $\vec{z}_i = I_i$ and $\vec{y}_i = z_i$ ($\vec{z}_i$ is the newly generated distance vector recording all vertices' distance to vertex i if they record v_i at the latest iteration. $\vec{y}_i$ is the "accumulated" vector for all vertices recording their distance to vertex i if they record v_i up to the latest iteration).

Next, at iteration 1, we have

$$\begin{array}{rcl} \vec{z}_1 & = & A^T\vec{z}_1 \ominus \{\vec{y}_1\} \\ \vec{z}_2 & = & A^T\vec{z}_2 \ominus \{\vec{y}_2 + \vec{y}_1\vec{y}_1^T I_2\} \\ \vec{z}_3 & = & A^T\vec{z}_3 \ominus \{\vec{y}_3 + (\vec{y}_1\vec{y}_1^T + \vec{y}_2\vec{y}_2^T)I_3\} \\ \cdots & & \\ \vec{z}_n & = & A^T\vec{z}_n \ominus \{\vec{y}_n + (\sum_{i=1}^{n-1} \vec{y}_i\vec{y}_i^T)I_n\} \end{array}$$

and then for all i, $\vec{y}_i = \vec{y}_i + \vec{z}_i$. Here, $\vec{y}_i\vec{y}_i^T$ generates the matrix recording the distance between any two vertices via vertex i (as their distance label), and $(\vec{y}_i\vec{y}_i^T)I_j$ corresponds to the distance from any vertex to vertex j via vertex i (they all use vertex j in their distance label). Using these values, we will repeat the above computations until no new label is generated.

We can easily prove the following result:

Theorem 2 *When the above algorithm stops, we have*

$$\vec{y}_i = \vec{x}_i, \textit{for all } i.$$

Proof Sketch: We will prove this by induction. First, let us define both $\vec{x}_i$ and $\vec{y}_i$ by iteration. Let $A^{*k} = (I + A + A^2 + \cdots + A^k)^T$ for k-th iteration ($k \leq d$, d is the diameter of the graph). We first observe that

$$\begin{aligned}
\vec{x}_1^k =& A^{*k} I_1 \\
\vec{x}_2^k =& A^{*k} I_2 \ominus (\vec{x}_1^k \vec{x}_1^{kT}) I_2 \\
=& A^{*k} I_2 \ominus (\vec{x}_1^{k-1} \vec{x}_1^{k-1T}) I_2 \\
\vec{x}_3^k =& A^{*k} I_3 \ominus (\vec{x}_1^k \vec{x}_1^{kT} + \vec{x}_2^k \vec{x}_2^{kT}) I_3 \\
=& A^{*k} I_3 \ominus (\vec{x}_1^{k-1} \vec{x}_1^{k-1T} + \vec{x}_2^{k-1} \vec{x}_2^{k-1T}) I_3 \\
& \cdots \\
\vec{x}_n^k =& A^{*k} I_n \ominus (\sum_{i=1}^{n-1} \vec{x}_i^k \vec{x}_i^{kT}) I_n \\
=& A^{*k} I_n \ominus (\sum_{i=1}^{n-1} \vec{x}_i^{k-1} \vec{x}_i^{k-1T}) I_n
\end{aligned}$$

Here, the above equation can be observed as the set $\vec{x}_i^k \ominus \vec{x}_i^{k-1}$ records what the vertex i can reach in exactly k steps, and this set will not help prune $\vec{x}_j^k$ (what vertex j can reach within k steps) for $j > i$.

Next, let us look at $\vec{y}_i^k$:

$$\begin{aligned}
\vec{y}_1^k =& \vec{y}_1^{k-1} + A^T \vec{z}_1^{k-1} \ominus \{\vec{y}_1^{k-1}\} \\
\vec{y}_2^k =& \vec{y}_2^{k-1} + A^T \vec{z}_2^{k-1} \ominus \{\vec{y}_2^{k-1} + \vec{y}_1^{k-1}(\vec{y}_1^{k-1})^T I_2\} \\
\vec{y}_3^k =& \vec{y}_3^{k-1} + A^T \vec{z}_3^{k-1} \ominus \{\vec{y}_3^{k-1} + (\vec{y}_1^{k-1}(\vec{y}_1^{k-1})^T + \vec{y}_2^{k-1}(\vec{y}_2^{k-1})^T) I_3\} \\
& \cdots \\
\vec{y}_n^k =& \vec{y}_n^{k-1} + A^T \vec{z}_n^{k-1} \ominus \{\vec{y}_n^{k-1} + (\sum_{i=1}^{n-1} \vec{y}_i^{k-1}(\vec{y}_i^{k-1})^T) I_n\}
\end{aligned}$$

Now, when $k = 0$ (initialization), the $\vec{y}_i^0 = \vec{x}_i^0$, for all i trivially holds. Now, we consider when k holds to be true, we will derive $k+1$ to be true. To prove this, again, we can start with $i = 1$, and we can easily observe: $\vec{y}_1^{k+1} = \vec{x}_1^{k+1}$. Now, assume $j \leq i$ are all true, then, let us consider $i + 1$:

$$\begin{aligned}
\vec{y}_{i+1}^{k+1} =& \vec{y}_{i+1}^k + A^T \vec{z}_{i+1}^k \ominus \{\vec{y}_{i+1}^k + (\sum_{l=1}^{i} \vec{y}_l^k (\vec{y}_l^k)^T) I_l\} \\
=& \vec{x}_{i+1}^k + A^T(\vec{x}_{i+1}^k \ominus \vec{x}_{i+1}^{k-1}) \ominus \{\vec{x}_{i+1}^k + (\sum_{l=1}^{i} \vec{x}_l^k (\vec{x}_l^k)^T) I_l\} \\
=& A^{*k} I_{i+1} \ominus \{(\sum_{l=1}^{i} \vec{x}_l^k (\vec{x}_l^k)^T) I_l\} \\
=& \vec{x}_{i+1}^{k+1}.
\end{aligned}$$

□

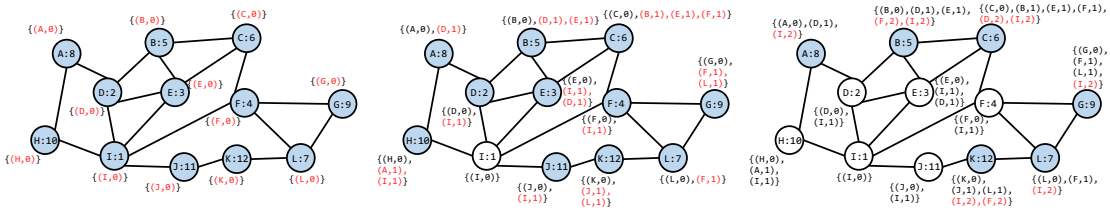
We note that in each iteration, all $\vec{z}_i$ can be computed simultaneously (as $\vec{z}_i$ now only depends on its own state and $\vec{y}_i$, which is computed from earlier iteration). Thus, we can parallelize PLL. However, we still do not have an efficient implementation based on linear algebra. More specifically, we have the requirement of materializing $(\sum_{i=1}^{k-1} \vec{y}_i \vec{y}_i^T) I_k$ in the

Algorithm 5: VC-PLL for $G = (V, E)$ with Order π

```
   /* Init.: (L(v): label; δL(v): new label from each iteration)   */
1  ActiveVertices← V; ∀v ∈ V, δL(v) ← {(v, 0)}, L(v) ← δL(v) ;
2  while ActiveVertices ≠ ∅ do
       /* Scatter Phase:                                           */
3      forall a ∈ ActiveVertices do
           /* a.Scatter(a.edges):                                  */
4          forall (a, v) ∈ a.edges do
5              for all (u, d(u, a)) ∈ δL(a), when π(u) < π(v) ∧ u ∉ L(v): send
                 (u, d(u, a) + 1) to v.messages ;
6      ActiveVertices ← ∅;
       /* Gather Phase:                                            */
7      forall v ∈ V : v.messages ≠ ∅ do /* Received Message        */
           /* v.Gather(v.messages):                                */
8          δL(v) ← ∅ ;
9          forall unique (u, d(u, v)) ∈ v.messages do
10             if d(u, v) < min_{h∈L(u)∩L(v)}{d(u, h) + d(h, v)} then
11                 Add (u, d(u, v)) to δL(v);
12         If δL(v) ≠ ∅: L(v) ← L(v) ∪ δL(v); Add v to ActiveVertices ;
```

k-th step, which will be very expensive.

3.3.3 Vertex-Centric Parallel Implementation



(a) Initial L of Graph G. (b) L after the 1st iteration. (c) L after the 2nd iteration.

Figure 3.2: A VC-PLL Example: light shadow vertices got new labels (colored in red) in this iteration and will spread them in the next iteration.

Algorithm Description: Algorithm 5 sketches the main process of performing PLL based on the vertex-centric computation model (Algorithm 4). In the *Initialization* phase, all vertices are active initially ($ActiveVertices = V$). For each vertex v, $L(v)$ records the

partial label, and $\delta L(v)$ records the new label being generated at each iteration. Initially, every vertex v records itself and distance 0 (any vertex reaches itself in zero steps). The main computation alternates between the *Scatter* phase and *Gather* phase and will continue until no new active vertices exist (Lines 2 to 12):

1) *Scatter* phase (Lines 3 to 5, also referred to as the push model): all active vertices with new labels perform a vertex *Scatter* function (Lines 4 to 5): each sends their new labels with the updated distance: $(u, d(u, a)) \in \delta L(a) \rightarrow (u, d(u, a) + 1)$ to all their neighbors (Line 5) with two conditions: the rank of vertex u needs to higher than v (otherwise, it will be pruned) and it has never been added to the label of v.

2) *Gather* phase (Line 7-12): all vertices that receive a new message ($v.messages \neq \emptyset$) perform a vertex *Gather* function (Lines 7-12): For a vertex v, it traverses all its received messages (distance label from its neighbors), and for each *unique* vertex $(u, d(u, v))$ across the set of messages, it confirms the distance check for the canonical labeling criterion: for a distance label message $(u, d(u, v))$, $d(u, v)$ must be smaller than the distances via any existing labels (L), i.e., $d(u, v) < \min_{h \in L(u) \cap L(v)} d(u, h) + d(h, v)$ (Line 10). If this is true, it will be added into $\delta L(v)$. Once $\delta L(v)$ is computed, and it is not empty, we will add it into $L(v)$ and add v to ActiveVertices (Line 12). Note that we need to identify unique vertices in the step above, because two neighbors may send the same vertex u.

Running Example: Figures 3.2 illustrates the first 2 iterations of label spreading in VC-PLL, where the labels in red denote newly generated labels δL. At each iteration, $L(v)$ is simply the union of all $\delta L(v)$ from all earlier iterations.

3.3.4 Theoretical Properties

This section explains two key properties of the proposed VC-PLL algorithm.

Property I: Correctness. Theorem 3 proves that VC-PLL produces the same label as PLL.

Theorem 3 *VC-PLL (Algorithm 5) produces the minimum labeling size (or canonical*

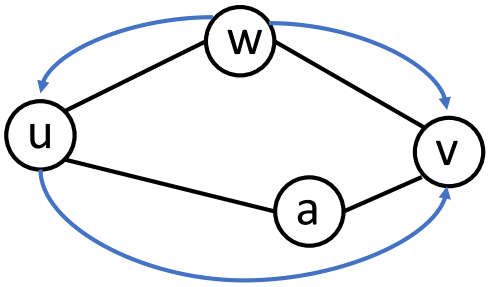


Figure 3.3: w reaches u and v before u reaches v

hub labeling) [19] given a vertex order π.

Proof Sketch: Recall the shortest path vertex set P_{uv} consists of all vertices on shortest paths between u and v (including u and v). Then, we need to prove $u \in L(v)$ *iff* *u has the highest order in P_{uv} (Definition 2).*

First ($\rightarrow$), we can see that if $u \in L(v)$, then we cannot find another vertex w with a rank higher than u, such that $d(u,v) \geq d(u,w) + d(w,v)$. Thus, u must have the highest order in P_{uv}. If not, assume we have another vertex $w \neq u$ that has the highest rank in P_{uv}. Then, based on our algorithm, w will be the highest ranked in P_{wu} and P_{wv}. Thus, w can always reach u and v before u reaches v (Figure 3.3) and it is in L_u and L_v when u reaches v.

Second ($\leftarrow$), assuming u has the highest order P_{uv}, then, based on the same argument, it can definitely go through the shortest path from u to v using Algorithm 5, and if it reaches v, no other vertices in L_v (and L_u) can prune it. $\square$

The following corollary can be immediately obtained.

Corollary 1 *In VC-PLL, when a distance label $(u, d(u,v))$ is added into $\delta L(v)$, $d(u,v)$ is the exact shortest path distance between u and v, and u has the highest rank in P_{uv}. Further, at any time $L(v) \subseteq \mathcal{L}(v)$, where $\mathcal{L}(v)$ is the final complete label of v.*

Property II: Time Complexity. Following the approach in PLL [9], we can obtain a theoretical upper bound of VC-PLL's time complexity.

Theorem 4 *Assuming graph G with a tree-decomposition [165] of tree-width w, then there is a vertex order π, in which the VC-PLL takes $O(w|E|\log|V|+w^2|V|(\log|V|)^2)$ time (the same as that of PLL [9]).*

Proof Sketch: The main idea is to utilize a recursive centroid extraction procedure to reorganize the tree decomposition: the centroid bag of the tree can break the tree decomposition into disjoint components where each disjoint component has no more than half of the tree bags. The centroid bag will become the root of the tree, and its children will be the centroid bag of the disjoint subtrees. Given this, the new tree's height is at most $\log|V|$. We then can *order the vertices in the graph based on their highest node in the tree, and if two vertices have the same height, they can break order arbitrarily.* Since each node in the tree corresponds to a graph separator, the maximal label size of any vertex is $w\log|V|$ (for a vertex in the leaf bag, all the vertex in its ancestors including its own bag of vertices can be added into the label). Then, the time complexity of generating distance labels is bounded by $O(w\log|V||E|)$. The time complexity of distance check is $O(w^2|V|\log^2|V|)$. □

3.3.5 Limitations of VC-PLL

Sequential Performance Comparison: We implemented Algorithm 5 (VC-PLL) and tested its performance on the `DBLP` graph (that is introduced in Section 3.6) against PLL using a single thread. We found that it has poor performance with a total execution time of $13,583$ seconds compared to less than 100 seconds for PLL! It does not fare well against PLL in other graphs either.

Basic algorithm and performance analysis reveals that VC-PLL introduces additional computational costs due to extra labeling spreading and distance testing. For a given vertex u, PLL will send it to a vertex v only once. In BFS, PLL will flag v after one distance label $(u, d(u, v))$ is passed through (Line 7 in Algorithm 3 is sequentially executed). But VC-PLL can send multiple $(u, d(u, v))$ messages to the same v in two consecutive iterations.

Therefore, VC-PLL introduces redundant distance labeling messages, which may also lead to redundant distance checks. Furthermore, individual distance checks in PLL can be much faster due to the reuse of $L(u)$ in an array or hash-table representation.

Additional Cost of Distance Label Generation: For a given vertex u, PLL will send it to a vertex v only once. In BFS, PLL will flag v after one distance label $(u, d(u, v))$ is passed through (Line 7 in Algorithm 3 is sequentially executed). But VC-PLL can send multiple $(u, d(u, v))$ messages to the same v at two consecutive iterations.

Lemma 3.1 *Given vertex u and vertex v, a distance label $(u : d(u, v))$ may reach v at exactly two possible and consecutive iterations: Let a be a neighbor of v, and $u \in L(a)$ (u is the highest rank vertex in P_{ua}), then it reaches v at $d(u, a)+1$ iteration, which is either: 1) equal to the shortest path distance between u and v, and u may or may not be added to $L(v)$; or 2) equal to $d(u, v) + 1$, i.e., the path from u to a to v is one step longer than the shortest path between u and v, and u will be pruned.*

Proof Sketch: To see this, we first need to prove that the shortest path distance $d(u, a)$ is smaller than or equal to $d(u, v)+1$, where a and v is the direct neighbor of one another. By way of contradiction, let us assume $d(u, a) \geq d(u, v) + 2$. Then, let w be the highest rank vertex in P_{ua}, then, we can find a path from u to w to v to a, which is $d(u, v) + 1$. This suggests $d(u, a) \leq d(u, v)+1$. Next, we show u indeed can reach v in two consecutive iterations. This happens when u reach v via a being the shortest path between u to v: $d(u, v) = d(u, a) + 1$; but u is not the highest rank one in P_{uv}. Thus, u is not added to $L(v)$ in $d(u, v)$ iteration. Now, assume u reach a' with $d(u, a') = d(u, v)$ and $u \in L(v')$ (added in $d(u, v)$ iteration. If a' is the neighbor of v, then u will be sent to v in $d(u, v)+1$ iteration as well. □

In addition, at each of these two iterations, if u has not been or is not in the label of v, then different neighbors of v may send the same $(u, d(u, v))$ messages to v.

Additional Cost of Distance Check:

Lemma 3.2 *The set consisting of all pairs (u, v) for distance checks is the same in PLL and VC-PLL.*

Proof Sketch: Let $reach(u)$ be the subset of vertices u reaches. In PLL, it corresponds to all the $(u, d(u, v))$ messages added into the Q (Line 7 in Algorithm 3). In VC-PLL, it corresponds to all the $(u, d(u, v))$ messages being sent to vertex v (Line 5 in Algorithm 5). Thus, $\bigcup_{u \in V} \{u\} \times reach(u)$ is the set consisting of all pairs (u, v) for distance check. In PLL and VC-PLL, for a vertex u, it is assigned to the same subset of vertices (Corollary 1). Also, it will also be sent to the same set of vertices that do not use u as a label. Thus, the set $\bigcup_{u \in V} \{u\} \times reach(u)$ is the same for both. □

However, *the number of distance checks in VC-PLL can be higher than PLL, as a vertex u can be sent to v in two consecutive iterations in VC-PLL.*

The computational cost of distance check

$$d(u, v) < \min_{h \in L(u) \cap L(v)} \{d(u, h) + d(h, v)\}$$

in VC-PLL is also higher than that in PLL. In VC-PLL, the cost is $O(|L(u)| + |L(v)|)$, where $L(u)$ and $L(v)$ are (partial) labels of u and v at the time of distance check for $d(u, v)$. Assuming $L(u)$ and $L(v)$ are not sorted, we can first map $L(u)$ into an array or hash table, and then check all the vertices in $L(v)$ against the above data structure. In PLL [9], since we process vertex u one at a time, and when we try to process u, its label $L(u)$ is already computed. Thus, we can first map $L(u)$ to an array only once at the beginning of the BFS iteration. Thus, the cost of $O(|L(u)|)$ can be practically saved for each distance check; thus the distance check for PLL is only $O(|L(v)|)$. For VC-PLL, we cannot do this directly as it is prohibitively expensive to map every $L(u)$ to an array or hash table at the same time.

To summarize, VC-PLL introduces redundant distance labeling messages, which may also lead to redundant distance checks $d(u, v)$. Furthermore, individual distance checks in PLL can be much faster due to the reuse of $L(u)$ in an array or hash-table representation.

In fact, these performance issues seem to challenge the capabilities of Vertex Centric (VC) computational in supporting: 1) effective message filtering and communication and 2) efficient memory access.

Thus, our question is: *can VC-PLL overcome its limitations and reduce those additional costs (message spreading and remote memory access)?*

3.4 Batched Vertex-Centric Algorithm

To deal with the performance inefficiency of VC-PLL discussed in the last section, we introduce a new algorithm. Specifically, here, the batches of the vertices are formed according to the rank of each vertex. For instance, the top $1K$ vertices form the first batch, and the next $1K$ vertices form the second batch, etc. Batches are processed in sequence with the vertices within each batch being processed in parallel. The reason to use batch is to 1) effectively and efficiently eliminate redundant message passing, and 2) significantly improve remote vertex memory access (as only the vertices in the batch need to be accessed remotely).

BVC-PLL Algorithm: Algorithm 6 sketches the batched Vertex-Centric algorithm for PLL, referred to as BVC-PLL. Specifically, here, the batches of the vertices are formed according to the rank of each vertex (Line 2). The earlier processed batch consists of the vertices with higher ranks (Line 3). BVC-PLL labels vertices one batch at a time and for assigning the labels in each batch, the vertex-centric computation in VC-PLL is followed (Lines 6-17) – more specifically, the Scatter Phase and Scatter function, Gather Phase and Gather function is preserved with only minor revisions for dealing with message passing and remote memory access. Each vertex v is associated with a candidate-bit vector $C(v)$. Its length is equal to the batch size. It will be initialized for each batch (Lines 1 and Line 18). During the Scatter phase, for any vertex a to send a message $(u, d(u, a) + 1)$ to its neighbor v, it will check if u is sent to v before ($u \notin C(v)$, Line 9). This corresponds to the *unvisited* flag in the original PLL. Due to the atomic compare-and-swap operation,

Algorithm 6: BVC-PLL for $G = (V, E)$ with Order π

```
   /* Init.: (L(v): label; δL(v): new label from each iteration)   */
1  ∀v ∈ V, L(v) ← ∅, C(v) ← ∅
2  Split V into equal-size batches: B_1, B_2, ··· B_T where B_i include the vertices with
     rank (i − 1) × |V|/T + 1 to i × |V|/T
3  forall B_i : i = 1 to T do /* Labeling in Batch                  */
4      ActiveVertices ← B_i
5      ∀u ∈ B_i, δL(u) ← {(u, 0)}, L(u) ← L(u) ∪ δL(u), and map L(u) to Hashtable
         H(u)
6      while ActiveVertices ≠ ∅ do
           /* Scatter Phase:                                       */
7          forall a ∈ ActiveVertices do
               /* a.Scatter(a.edges):                              */
8              forall (a, v) ∈ a.edges do
9                  for all (u, d(u, a)) ∈ δL(a), when π(u) < π(v) ∧ u ∉ C(v): flag u in
                     C(v) and send (u, d(u, a) + 1) to v.messages
10         ActiveVertices ← ∅
           /* Gather Phase:                                        */
11         forall v ∈ V : v.messages ≠ ∅ do /* Received Messages   */
               /* v.Gather(v.messages):                            */
12             δL(v) ← ∅
13             forall (u, d(u, v)) ∈ v.messages do
14                 if d(u, v) < min_{h∈L(u)∩L(v)}{d(u, h) + d(h, v)} then
15                     Add (u, d(u, v)) to δL(v)
16             If δL(v) ≠ ∅: L(v) ← L(v) ∪ δL(v); Add v to ActiveVertices
17             If v ∈ B_i: Add δL(v) to H(v)
18     ∀v ∈ V, C(v) ← ∅
```

it can guarantee only one message from u is being sent to v and thus help resolve the redundant distance labeling generation problem (in Subsection 3.3.5).

Each vertex u in the batch B_i will map its existing label $L(u)$ to a hash-table (or array) $H(u)$ at the beginning of vertex-centric computation (Line 5). Since the new label of u may be generated during the labeling process, we will map the new label $\delta L(v)$ to $H(v)$ when the update is available (Line 17). Given this, the distance check (in Line 14) only needs to go through $L(v)$, and thus has the same distance check cost as the original PLL.

Next, we discuss two key optimization techniques which leverage the batch processing

to reduce the additional computational costs from VC-PLL:

Using Bit Operation for Efficient Message Passing and Filtering: In each batch processing step, an active vertex only processes up to *batch_size* unique labels. Based on this important observation, we can use a compact bit-vector data structure called *candidate bit-vector* for efficient message filtering. The basic idea is as follows. Each active vertex maintains a candidate bit-vector with the length of *batch_size* bits, each bit corresponding to a vertex in the batch (e.g., if the *batch_size* is 1K, such candidate bit-vector is only 128 bytes). If a vertex u in the current batch is sent to a vertex v, then its corresponding bit in the candidate bit-vector of v is set. Note that the use of bit-vectors also allows atomic *compare-and-swap* operation in the shared memory setting. Note that without batch processing, we have to consider doing an expensive list merge for handling message passing and aggregation (as the scatter and gather functions in VC-PLL for distance label messaging and processing, respectively).

Improving Data Locality for Remote Vertex Memory Access: Simply speaking, only the vertices in the current processing batch can be accessed remotely during the vertex-centric computation. Because the number of vertices in each processing batch is limited, we can use a compact data structure such as an array or hash-table to store their labels for efficient $O(1)$ access (similar to what is done in PLL for each processed vertex in distance checks).

Correctness: It is easy to see that *BVC-PLL (Algorithm 6) produces the minimal labeling given a vertex order* π: the distance criterion ($u \in L(v)$ if u has the highest rank in P_{uv}) is maintained as BVC-PLL can assign u to $L(v)$ at u's batch correctly (Theorem 3) following the batch processing order. Another interesting property is that when the batch size reduces to one, i.e., when we process one vertex at a time, then BVC-PLL behaves exactly the same as the original PLL [9].

Complexity: We note that introducing and using bit-vector $C(v)$ for each vertex v and $H(u)$ for each processing batch vertex u does not introduce additional time complexity compared with PLL. PLL uses only one bit for each vertex v as the *visited* flag and one

$H(u)$ for distance check, whereas BVC-PLL simply utilizes a group of them at the same time. Thus, the time complexity results of Theorem 4 hold for BVC-PLL as well.

3.4.1 Complete Computation Cost Comparison between BVC-PLL and PLL

In the following, we provide an apple-to-apple computational cost analysis between BVC-PLL and PLL. We will focus on the cost of generating (sending) distance labels and distance checks.

Cost of Distance Label Generation: Since in BVC-PLL, each vertex u can be sent to v exactly once, together with Lemma 3.2 (the same set of u reaches v), we thus observe:

Lemma 3.3 *The time complexity of sending vertex label messages $(u, d(u, v))$ along the edges in graph G given an order π, is the same for PLL and BVC-PLL.*

Following Lemma 3.3, we obtain the following corollary.

Corollary 2 *The total number of distance checks (applying canonical labeling criterion) being invoked in PLL (Line 5 in Algorithm 3) is the same as those being invoked in BVC-PLL (Line 14 in Algorithm 6).*

This is because the number of distance checks is equivalent to the total number of generated distance label message: $\sum_{u \in V} |reach(u)|$ (following the algorithm logic).

Cost of Distance Check: Now, the cost of the same distance check on $d(u, v)$: $d(u, v) < \min_{h \in L(u) \cap L(v)} d(u, h) + d(h, v)$, in PLL and BVC-PLL, is $O(|L(v)|)$. However, $L(v)$ are different for PLL and BVC-PLL: In PLL, when u reaches v, $L(v)$ consists of all vertex labels which have higher rank than u; In BVC-PLL, assuming u in batch B_i, $L(v)$ consists of all the vertex labels in all the batches before B_i (those are the same as those in PLL) and the vertices in the current batch which are within the distance of $d(u, v)$.

Given this, let us focus on only those vertices being added at batch B_i for $L(v)$, and denote it as $L^i(v)$. Next, we break the distance check cost on $|L^i(v)|$ into two categories:

1) the *positive* distance check which will confirm the vertex u and can add it into the corresponding label of v; 2) the *negative* distance check will return false on the distance check and thus prune the vertex u.

Theorem 5 *(Positive Distance Check) The time complexity of all positive distance checks in BVC-PLL is lower than or equal to that of PLL.*

Proof Sketch: Let us consider any batch B_i. For the positive cases of distance check $d(u, v)$ here, given a vertex v and u, u will always be added to the label of v. For PLL, for a vertex v, let its complete $\mathcal{L}^i(v)$ consists of $u_1, u_2, \cdots, u_n \in B_i$, where $n = |\mathcal{L}^i(v)|$ and $\pi(u_1) < \pi(u_2) \cdots < \pi(u_n)$. Then the total cost of distance check with respect to $|L^i(v)|$ is simply $0 + 1 + \cdots + n - 1 = n(n-1)/2$,
because in PLL, when u_i arrives, $L^i(v)$ already consists of partial labels $\{u_1, \cdots, u_{i-1}\}$. For BVC-PLL, for a vertex v, we note that its distance label in B_i is arriving in group according to their distances. Let $g_1, g_2, \cdots g_k$ be the groups ordered by arriving (as well as distance), i.e., given any two vertices $x, y \in g_i$, $d(x, v) = d(y, v)$, and their distance is smaller than those in g_{i+1}. Note that *for any vertex $u \in g_i$, we only utilize $L^i(v) = g_1 \cup \cdots \cup g_{i-1}$ for distance check* (See Lines 7-12 in Algorithm 5, $L^i(v)$ will be updated until all the distance checks in a batch g_i are done). Let $n_i = |g_i|$ and $n = \sum_{i=1}^{k} n_i$, making the total cost of distance checks of vertex v with respect to $|L^i(v)|$ in BVC-PLL to be

$$
\begin{aligned}
&0 + n_1 \times n_2 + (n_1 + n_2) \times n_3 + \cdots + (\sum_{i=1}^{k-1} n_i) \times n_k \\
&= (n - n_1)n_1 + (n - (n_1 + n_2))n_2 + \cdots + (n - \sum_{i=1}^{k-1} n_{k-1}) \\
&= n(n-1)/2 - \sum_{i=1}^{k} n_i(n_i - 1)/2. \square
\end{aligned}
$$

Figure 3.4 illustrates the key idea in the proof of Theorem 5. Assuming 9 vertices $a, b, \cdots, i$ in one batch being added into $L(v)$ in PLL labeling, its total distance check

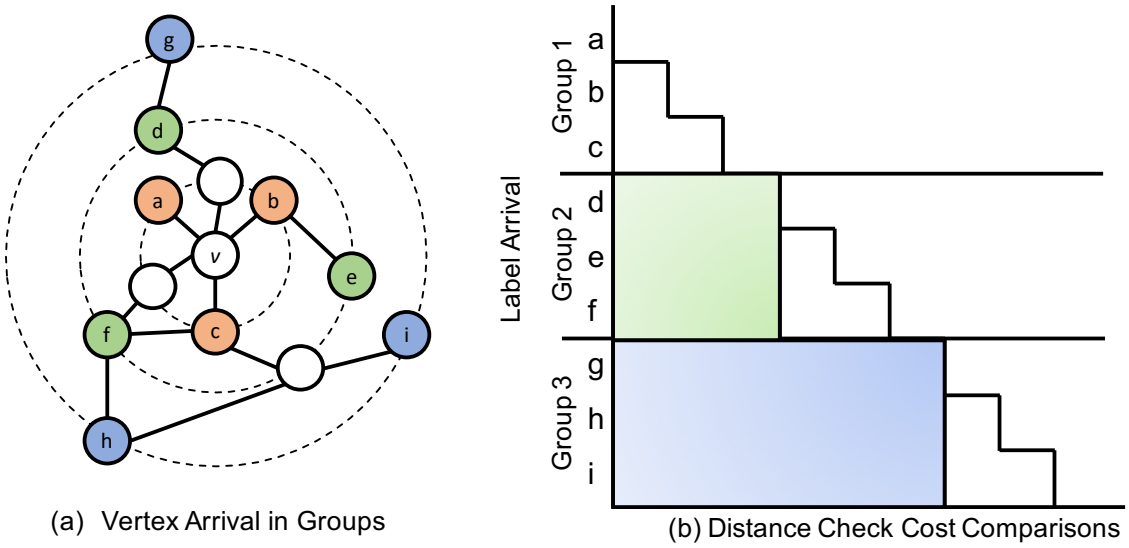


Figure 3.4: Theorem 5: (Positive Distance Check) The time complexity of all positive distance checks in BVC-PLL is lower than or equal to that of PLL.

cost is 36 no matter which order they are received in (visualized as the area under the diagonal stairs). Now assuming they arrive in three groups as shown in Figure 3.4(a), then in BVC-PLL, their total distance check cost is $3 + 3 \times 6 = 27$, a 25% reduction compared to PLL.

Theorem 5 essentially shows that BVC-PLL is able to save the intro-group cross-vertex comparison in each batch. Basically, if vertices arrive at the same time, they have the same distance to vertex v and cannot prune one another.

To compare the time complexity difference between PLL and BVC-PLL for the negative distance check, we introduce the following notation: for any vertex x, and one of its vertex label u ($u \in L^i(x)$), we denote $< x, u >$ to be a subset: $\{x \in$

$$\bigcup_{y \in N(x)} L^i(y) \setminus L^i(x) : \pi(u) < \pi(v) < \pi(x), d(x,u) > d(v,y) + 1\}$$

Similarly, we define $< y, v >$ for vertex y with its label v, $v \in L^i(y)$:

$$\{u \in \bigcup_{x \in N(y)} L^i(x) \setminus L^i(y) : \pi(u) < \pi(v), d(x,u) > d(v,y) + 1\}$$

Theorem 6 *(Negative Distance Check) In batch B_i, and on negative distance check, the time complexity saved by BVC-PLL compared with PLL is no higher than*

$$O(\sum_{x \in V} \sum_{u \in L^i(x)} |\langle x, u \rangle| - \sum_{y \in V} \sum_{v \in L^i(y)} |\langle y, v \rangle|).$$

The time complexity saved by PLL compared with BVC-PLL is no higher than

$$O(\sum_{y \in V} \sum_{v \in L^i(y)} |\langle y, v \rangle| - \sum_{x \in V} \sum_{u \in L^i(x)} |\langle x, u \rangle|).$$

Proof Sketch: To quantify the difference of the time complexities between two algorithms, we focus on the cases where one algorithm can save computational cost when the $L^i(v)$ will be different for distance check $d(u, v)$.

For the first case, let us consider vertex x, it has a vertex $u \in L^i(x)$. Now, consider any vertex $v \in B_i$ reaches vertex x for distance check and returns negative result. If v can reach x, it must be a label of neighbor y of x, i.e., $v \in L^i(y), y \in N(x)$, and $v \notin L^i(x)$ (false distance check). When v reaches x, it has also lower rank than u but higher than x: $\pi(u) < \pi(v) < \pi(x)$. Given this, for PLL, u is already in $L(x)$; however, for BVC-PLL, v can reach x before u reaches x. Thus, this case will introduce a gain for BVC-PLL; and such v is characterized and recorded in set $\langle x, u \rangle$.

For the second case, let us consider vertex y, and it has a vertex $v \in L^i(y)$. Now, consider any vertex $u \in B_i$ reaches vertex y for distance check and returns negative result. If u can reach y, it must be a label of neighbor x of y, i.e., $u \in L^i(x), x \in N(y)$, and $u \notin L^i(y)$ (false distance check). When u reaches y, it has also higher rank than v: $\pi(u) < \pi(v)$. Given this, for BVC-PLL, v is already in $L(y)$; however, for PLL, u can

reach x before v is added into $L(y)$. Thus, this case will introduce a gain for PLL; and such u is characterized and recorded in set $\langle y, v\rangle$. □

Theorem 6 does not provide a clear winner on the cost of negative check. However, from the symmetric expression of these two qualities, we conjecture they should be close to one another. In Section 3.6, we will experimentally confirm this. In addition, for negative distance check, we typically do not need to traverse through the entire $L(v)$ set. Indeed, the bit-parallel mechanism proposed in the original PLL paper [9] can help provide almost $O(1)$ pruning. Since the number of negative checks is the same for PLL and BVC-PLL, we expect their overall cost will be fairly close to each other.

Putting It Together: Assuming that PLL and BVC-PLL have a similar cost for negative distance checks, theoretically, *BVC-PLL may have smaller computational cost than that of PLL (due to positive distance check) since they have the same cost of generating/sending distance labeling!* Furthermore, BVC-PLL is guaranteed to have a smaller memory access cost for graph topology than PLL as it groups messages together for each edge access. Overall, it seems BVC-PLL, an unexpected marriage between PLL and VC computation, can run faster than the original PLL sequentially and can also enjoy the scalability of the VC model! Indeed, Section 3.6 shows that it can be more than two times faster than PLL (both using one thread) on real-world graphs.

3.5 Variants and Parallel Implementation

3.5.1 Generalization

Directed Graphs: For a directed graph, each vertex v is assigned with two labels $L_{in}(v)$ and $L_{out}(v)$. VC-PLL and BVC-PLL can be easily extended to handle directed graphs by considering these as separate computations. Specifically, in the Scatter function, the new labels δL_{in} and δL_{out} will be sent out along the outgoing edges and incoming edges, respectively. In the Gather function, there will be two message queues: one for candidate vertices in L_{in}, and another for those in L_{out}. The labels generated by this algorithm

will be canonical. The computational complexity analysis in Subsection 3.3.4 holds for directed graphs as well.

Weighted Graphs: The direct application of VC-PLL and BVC-PLL (by changing $d(u, v) + 1$ to $d(u, v) + w_e$ where w_e is the edge weight) on weighted graphs can produce a 2-hop labeling; but it may not be a canonical labeling. This is because unlike unweighted graphs, the iteration on the vertex-centric model will not be in sync with the distance between two vertices. For instance, when vertex u reaches v in two iterations, their distance may be larger than a path via vertex w with a higher rank, but w may take more than 2 iterations to reach v and u. Given this, we cannot use the partial label $L(u)$ at an arbitrary iteration to fully determine if vertex v is a true or final label for u anymore. Thus, adding vertex v into u's partial label $L(u)$ or $\delta L(u)$ (using the partial labels in the weighted graph) may lead to unnecessary vertices being spread in the networks. To deal with this problem, at the end of each batch processing (Line 18 in BVC-PLL), we can perform a distance recheck using only the labels from the batch. Since the hash tables of the labeling vertices in the batch are still in the memory, this recheck can be quite efficient.

3.5.2 Parallel Implementation Issues

Shared Memory Implementation: The BVC-PLL computation (as shown in Algorithm 6) can be easily parallelized with coarse-grained threads. The computation of each batch uses vertex-centric processing (line 6 to line 17) that consists of two parallel phases: (*Scatter* and *Gather*), with an implicit synchronization between them. In each phase, each thread processes a chunk of active vertices with dynamic scheduling to achieve load balance. In a shared memory implementation, we use OpenMP to parallelize each batch, manage the workload of each thread dynamically, and expand active vertices.

Extension to Distributed Platforms: BVC-PLL's shared memory parallel implementation was also extended to distributed platforms using MPI. Because BVC-PLL processes nodes based on an order, and this order is, in turn, based on degrees, we use the follow-

ing process to maintain load balance. After sorting vertices according to their rank (i.e., degrees) from high to low, vertices are assigned to nodes in a *zigzag round-robin* way. For instance, assuming there are p nodes in total, p consecutive vertices are assigned to node 0 to node $p-1$, respectively; then the next p consecutive vertices are assigned to node $p-1$ to node 0, and so on.

In maintaining information for each node, we follow the *master-mirror* notion (as also used by PowerGraph [78]). Every edge is placed on the computer node that owns its destination vertex (as a master) [187]. As described above, a vertex is assigned to a specific computer node on which the vertex is a *master*. If the vertex is the source of an edge that is assigned to a different computer node, the same vertex is a *mirror* on that computer node. Only the master contains all labels of that vertex.

The parallel implementation is based on MPI. In the scatter phase, every master sends its newly added labels to all of its mirrors. Then a mirror scatters those labels to its neighbors on that computing node. In the gather phase, every node maintains a hash-table to store all labels of vertices in the current batch. When conducting a distance check, a vertex only needs to look up its own labels and this hash-table. Thus, the distance check only requires local operations without any message passing to remote nodes. The hash-table is once established at the beginning of every batch and is synchronized at the end of the scatter phase. Note that the batch strategy is even more important for the distributed implementation than the shared memory one as it reduces the remote access overhead significantly.

SIMD Parallelization for Weighted Graphs: BVC-PLL is able to significantly increase the data locality for remote vertex memory access, thereby offering us extra opportunities to better exploit fine-grained data-level parallelism (i.e., SIMD parallelism or vectorization). Consider the *Gather Phase* in Algorithm 6 that involves an intensive label distance check kernel (line 13 to line 15). BVC-PLL can vectorize this kernel with the help of advanced SIMD gather/scatter and mask instructions in the latest AVX-512 intrinsic

set[1]. Unfortunately, such vectorization requires a dynamic expansion of the compact label structure that offsets most benefits of SIMD parallelization for unweighted graphs. However, vectorization turns to be useful for weighted graphs because the distance recheck operation incurs extra computation overhead. Efficient SIMD parallelization can significantly reduce such overhead.

3.5.3 Some Implementation Details

Integrated Bitmap and Queue: Much temporary data is generated for both labeling vertices and active vertices during each batch processing. These steps require a *clearance* (e.g., Algorithm 6, line 18). The cost of this clearance is significant as this operation occurs for each batch. Traditionally, we often use either a bitmap or a queue to handle the set of active vertices. However, they become inefficient or insufficient for supporting BVC-PLL. For a bitmap, each of its cleanings can take $O(|V|)$ where $|V|$ is the total number of vertices; for a queue, it cannot support efficient checks for whether a given vertex is active or not. Given this, we propose a new traversal control data structure by combining both the bitmap and the queue. The basic idea is that a bitmap supports fast recording and checking visited vertices and a queue supports fast finding and clearing the visited vertices. Each time a vertex is processed, we add it to both the bitmap and the queue. This approach is different from the bitmap and queue used in the push and pull strategy presented in [155, 175, 26] because we use both the bitmap and queue simultaneously rather than in different stages of processing.

Bit-parallel Adoption: Similar to PLL [9], *bit-parallel* is also adopted to accelerate the *distance checking* in the implementation of BVC-PLL for unweighted graphs. Its construction is similar to multi-source BFS traversals and can be easily expressed in the Vertex-Centric computing model.

[1] `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`

3.6 Evaluation

In this section, we perform a detailed evaluation of BVC-PLL, focusing on answering the following questions: 1) How does BVC-PLL algorithm perform against the original PLL in a sequential setting (single thread; no parallelism)? 2) In a shared memory setting, how does BVC-PLL scale as the number of threads increases? 3) How well does our distributed memory implementation scale (especially on large graphs with more than a billion edges)? 4) How does the weighted extension of BVC-PLL with SIMD perform and how does it fare against ParaPLL [162] (the state-of-the-art parallel weighted PLL algorithm)?

3.6.1 Experimental Setup

Table 3.2: Characterization of evaluation graphs.

Name	Graph	Category	$\|V\|$	$\|E\|$
GNUT	Gnutella	Social	63 K	148 K
DBLP	DBLP	Citation	317 K	1 M
WIKI	Wikipedia Talk	Comm.	2.4 M	5 M
YOUT	YouTube	Social	3.2 M	9 M
TREC	TREC WT10g	Hyperlink	1.6 M	8.0 M
SKIT	Skitter	Computer	1.7 M	11 M
CADO	Catster/Dogster	Social	62 K	15 M
FLIC	Flickr	Social	2.3 M	33 M
HOLY	hollywood-2009	Social	1.1 M	114 M
INDO	indochina-2004	Hyperlink	7.4 M	194 M
IT	it-2004	Hyperlink	41.3 M	1.1 B
GSH	gsh-2015-host	Computer	68.6 M	1.8 B

Platforms: Our *shared memory* scaling experiments were performed on an Intel Xeon Gold 6138 CPU. It is a Skylake processor with 20 cores running at 2.0 GHz supporting 512-bit AVX-512 intrinsics, with 27.5 MB L3 cache and 192 GB DDR4 memory. All code is compiled with an Intel icc compiler (version 19.0.2.187) with `-O3` optimization option. Hyper-threading is not used to simplify the analysis of experiment results. Our experiments for *distributed memory* scalability (while using one thread per node) were performed on a cluster with 16 nodes, each of which has an Intel Xeon E5-2680 CPU at

2.4 GHz with 35.8 MB L3 cache and 125 GB DDR4 memory. For evaluating the version that combines *shared and distributed memory* parallelism (and ability to process large graphs), we use another cluster with up to 224 cores (maximum of 28 cores per node and up to 8 nodes) and up to 512 GB memory per node.

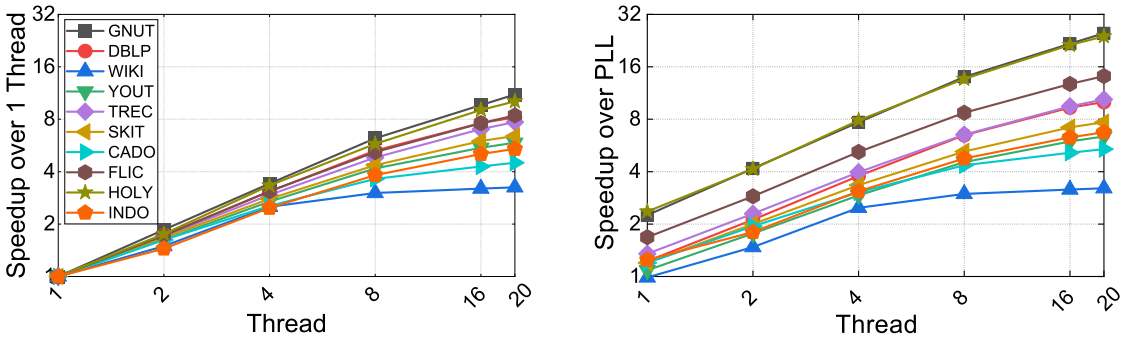
Graph Datasets: The 12 graphs used in our evaluation are summarized in Table 3.2. They are from 5 categories (Social, Citation, Communication, Hyperlink, and Computer) with varied numbers of vertices and edges – `GNUT`, and `WIKI` are from SNAP[2], `DBLP`, `YOUT`, `TREC`, `SKIT`, `CADO`, and `FLIC` are from KONET[3], `HOLY` and `INDO` are from SuiteSparse Matrix Collection[4], and `IT` and `GSH` are from WebGraph[5]. Particularly, `IT` and `GSH` are two large graphs with more than 1 billion edges. Because of large memory and computation time associated with these two massive datasets, experiments were limited to the version that combined shared and distributed memory parallelism, and used either 4, 6, or 8 nodes, with 4, 8, 16, or 28 threads on each node. For all other datasets, because of limited size (and thus parallelism), we either used only 1 node or 1 thread per node. These graphs are all unweighted. To test the performance of our BVC-PLL on weighted graphs, we randomly assign weights (from 1 to 7 with a uniform distribution) to their edges. Since we only evaluate algorithms for undirected graphs, we have transformed the edges in the directed graphs in Citation and Hyperlink as undirected edges.

Baselines: For the sequential performance comparison on unweighted graphs, we compare BVC-PLL against the PLL implementations by the original authors [9] and by [123]. We found these two implementations provide comparable performance with the former being slightly faster. Given this, we only compare against this version. For performance comparison on weighted graphs, we compare the weighted BVC-PLL against the implementation of ParaPLL [162], applying SIMD parallelism to both. Note that existing published work on parallelizing PLL (across threads or nodes) either has limited parallelism or does not

[2] `https://snap.stanford.edu/snap/`
[3] `http://konect.uni-koblenz.de/`
[4] `https://sparse.tamu.edu/`
[5] `http://law.di.unimi.it/datasets.php`

Table 3.3: **Unweighted Performance (sec.)**: BVC-PLL vs. PLL. LT denotes labeling time (sec.). $|L|$ denotes the average label size for each vertex. SP denotes speedup. BVC-PLL and PLL have the same label size. The same short names are used for Table 3.4.

Name	$\|L\|$	PLL	BVC-PLL	
		LT	LT	SP
GNUT	477	33	**13**	2.46
DBLP	214	61	**47**	1.30
WIKI	12	40	**32**	1.24
YOUT	70	285	**249**	1.15
TREC	269	462	**323**	1.43
SKIT	138	317	**242**	1.31
CADO	96	117	**92**	1.28
FLIC	442	1,624	**909**	1.79
HOLY	2,199	10,743	**4,368**	2.46
INDO	442	4,755	**3,508**	1.36



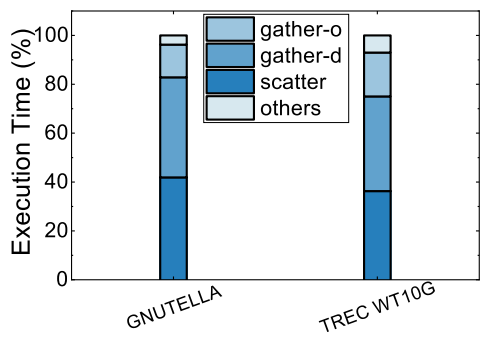
(a) Speedup over 1 thread.

(b) Speedup over PLL.

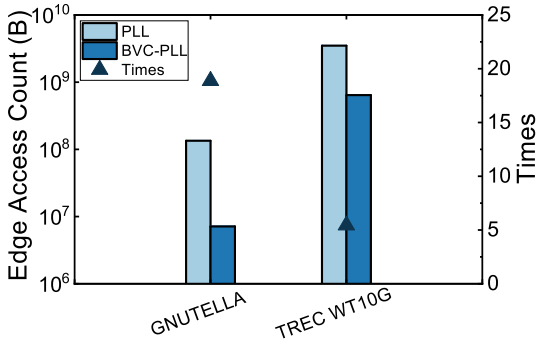
Figure 3.5: The scalability of BVC-PLL (unweighted).

produce the same results as a sequential implementation, and thus no comparison was performed. In all of our experiments, we determine the vertex order through the original and the most popular method where the vertices are ordered by their vertex degree [9, 123].

Throughout our experiments, we use 1024 as the batch size for unweighted graphs and use 512 for weighted graphs. In our experimental platform, we found those two are the optimal batch size. In general, we observe the larger the batch size the better the performance, as long as there is available memory. Due to the space limitation, we will not report the results that explore the sensitivity to batch size.

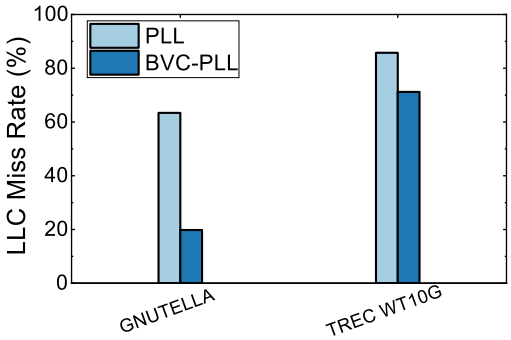


(a) BVC-PLL breakdown.

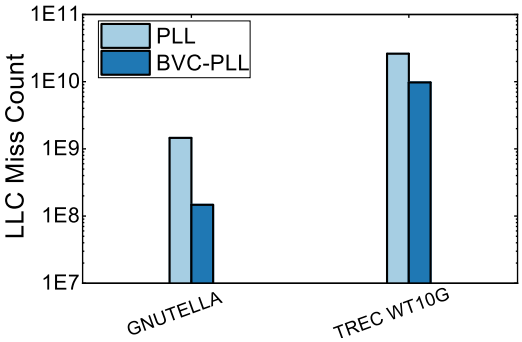


(b) Edge access count.

Figure 3.6: Performance analysis.



(a) LLC miss rate.



(b) LLC miss count.

Figure 3.7: Data locality: BVC-PLL vs. PLL.

3.6.2 BVC-PLL vs PLL and Shared Memory Scalability

Table 3.3 shows the performance comparison between BVC-PLL as a sequential algorithm and PLL (both using a single thread and no other parallelism, such as SIMD) on all graphs. Both algorithms use the same vertex order and produce the same label size, as expected. Interestingly, the BVC-PLL algorithm consistently outperforms PLL with the speedup ranging from 1.15× (`YOUT`) to 2.46× (`GNUT` and `HOLY`) with an average speedup 1.58× over PLL. Later in this section, we will report a more detailed cost breakdown and comparison.

Figure 3.5 shows the shared-memory scalability of BVC-PLL on all but the two largest graphs (1 node execution was not feasible for these graphs because of memory requirements). Figure 3.5a shows its speedup over 1-thread BVC-PLL, while Figure 3.5b shows

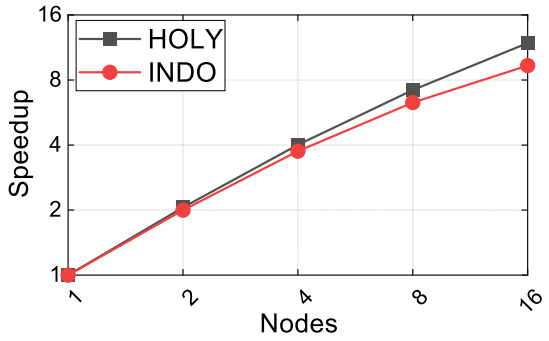


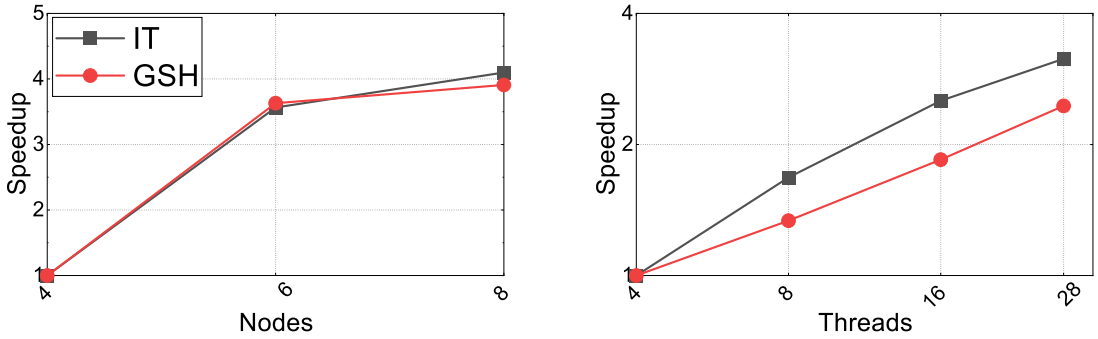
Figure 3.8: The scalability of distributed BVC-PLL (unweighted).

its speedup over the original sequential PLL. With 20 threads, BVC-PLL can achieve up to 11.08× and 24.95× speedup over its 1-thread version and PLL, respectively (with geometric mean 6.68× and 9.33×, respectively), demonstrating good scalability.

In addition, by comparing Figure 3.5 and the average label sizes in Table 3.3, we found that generally, BVC-PLL scales better as the average label size increases. For example, `GNUT` and `HOLY` with the largest average label sizes result in the best scalability while `WIKI` with the smallest results in lower scalability. The labeling size provides a good indication of the total computational costs (message passing and distance checks) involved for each vertex.

Figure 3.6a shows the overall running time breakdown on two graphs: `GNUT` and `TREC`. Due to space limitation, we only report results for these two – trends are similar in other graphs. We can see the Gather and the Scatter phases dominate the overall computational costs. In addition, within gather, the distance check time takes about $60\% - 80\%$ and $30\% - 40\%$ of the gather phase and overall time, respectively.

Figure 3.6b shows the total number of edge access for BVC-PLL and PLL on two graphs: `GNUT` and `TREC`. We can see that BVC-PLL has 5 and 18 times reduction for both graphs. Finally, Figure 3.7 shows the LLC (last level cache) miss rate and miss access count for the entire labeling process of BVC-PLL and PLL. We can see BVC-PLL has consistent lower LLC miss rate and access count than PLL.



(a) Speedup over 4 nodes w/ 28 threads.

(b) Speedup over 4 threads w/ 8 nodes.

Figure 3.9: Scalability of BVC-PLL (unweighted) on large graphs.

3.6.3 Distributed Memory Results

To further test scalability, we experimented with distributed memory parallelism. For the third and fourth largest datasets in our collection, `HOLY` and `INDO`, we used 1 thread per node. (Experiments on the two largest datasets, `IT` and `GSH`, combine both distributed memory and shared memory parallelism and are reported in the next subsection.) Figure 3.8 shows the distributed memory scalability of unweighted BVC-PLL on `HOLY` and `INDO`. For `HOLY`, BVC-PLL achieves 2.05×, 4.00×, 7.18×, and 11.85× speedup over 1 node as the number of nodes is 2, 4, 8, and 16, respectively. For `INDO`, the speedups over 1 node are 1.99× (on 2 nodes), 3.74× (on 4 nodes), 6.29× (on 8 nodes), and 9.32× (on 16 nodes).

When the number of nodes is large (e.g., 16 nodes), BVC-PLL cannot achieve near linear speedups mainly because of the communication overhead caused by the large volume of label data. This is particularly obvious for `INDO` that generates large label data.

3.6.4 Test on Large Graphs w/ Billions of Edges

BVC-PLL works on large graphs as well. We report results from `IT` and `GSH` that have over 1 billion edges. Because of computation time and memory requirements, experiments on these graphs involve the combination of distributed memory and shared memory parallelism, with 4, 6, or 8 nodes and 4, 8, 16, or 28 threads per node.

Figure 3.9 shows the results – Figure 3.9a shows the scalability from 4 to 8 nodes (each

Table 3.4: Weighted Performance (sec.): BVC-PLL vs. PLL (Dijkstra). "-S" denotes SIMD version. "-N" denotes non-SIMD version.

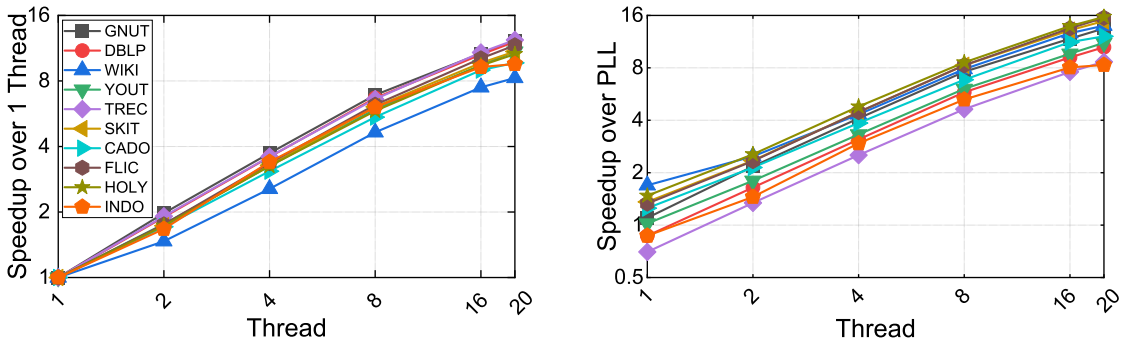
Name	$\|L\|$	PLL		BVC-PLL			
		LT-N	LT-S	LT-N	LT-S	SP-N	SP-S
GNUT	656	52	47	123	**37**	0.42	1.26
DBLP	387	152	**139**	293	146	0.52	0.95
WIKI	152	350	327	396	**171**	0.88	1.92
YOUT	147	652	625	763	**546**	0.85	1.14
TREC	304	632	**579**	1,140	662	0.55	0.87
SKIT	432	1,511	1,467	2,146	**921**	0.70	1.59
CADO	224	527	510	442	**347**	1.19	1.47
FLIC	653	3,879	3,826	4,189	**2,483**	0.93	1.54
HOLY	2,217	18,707	18,041	24,161	**10,399**	0.77	1.73
INDO	828	13,940	**13,105**	26,744	14,768	0.52	0.89

with 28 threads). From 4 to 6 nodes, `IT` shows super-linear speedup mainly due to the increase of available memory and cache capacity; while from 6 to 8 nodes, its speedup is sublinear because of the intensive label data communication. `GSH` shows a similar trend. Moreover, Figure 3.9b reports BVC-PLL's good scalability on 8 nodes with changing the OpenMP thread count from 4 to 28.

3.6.5 Extension to Weighted Graphs and SIMD

A similar performance study is conducted between PLL and BVC-PLL for weighted graphs. To evaluate weighted BVC-PLL's sequential performance against PLL, we have modified the original PLL implementation as suggested in [9], changing its BFS traversal to Dijkstra. We also extended BVC-PLL as described in Subsection 3.5.1. Please notice: both PLL and BVC-PLL are optimized with SIMD for the weighted version (and for unweighted, we also implemented them with SIMD however without obvious speedup change).

Table 3.4 shows the comparison results for 1-thread SIMD and non-SIMD versions of PLL and BVC-PLL. For all non-SIMD tests, PLL consistently performs better than BVC-PLL; while for most SIMD tests, BVC-PLL outperforms PLL. This is because the



(a) Speedup over 1 thread.

(b) Speedup (Dijkstra).

Figure 3.10: Scalability of BVC-PLL (weighted) on shared memory.

weighted BVC-PLL introduces additional distance check (due to additional message passing) and rechecks, which significantly increases the number of instructions for BVC-PLL, resulting in degraded performance. However, SIMD parallelism is a good remedy that can significantly reduce the number of instructions. It should be noted that BVC-PLL is able to effectively exploit SIMD parallelism because the data locality has been improved. (See the performance analysis in the last Subsection). In particular, for SIMD version, BVC-PLL outperforms PLL for 7 out of 10 graphs, resulting in 1.14× to 1.92× speedup with an average of 1.34×. For the slowdown cases, BVC-PLL's performance is only degraded up to around 10%. Our BVC-PLL is able to continue exploring hierarchical parallelism to further extract the most out of the massive parallelism of modern processors.

Figure 3.10 shows the scalability of BVC-PLL on all weighted graphs, in which, Figure 3.10a shows its speedup over 1-thread BVC-PLL while Figure 3.10b shows its speedup over PLL. With 20 threads, BVC-PLL can achieve up to 12.34× and 15.68× speedup over 1-thread and PLL, demonstrating good scalability.

Finally, we compare BVC-PLL with the state-of-the-art ParaPLL, which does weighted parallel PLL. Unfortunately, it can only run on small graphs (this is consistent on what being presented in their original paper [162]). In Figure 3.11 shows the performance comparison of BVC-PLL and ParaPLL on the graph `GNUT` (the only graph we are able to run for ParaPLL, as it throws a `Segmentation Fault` for the other graphs). For this

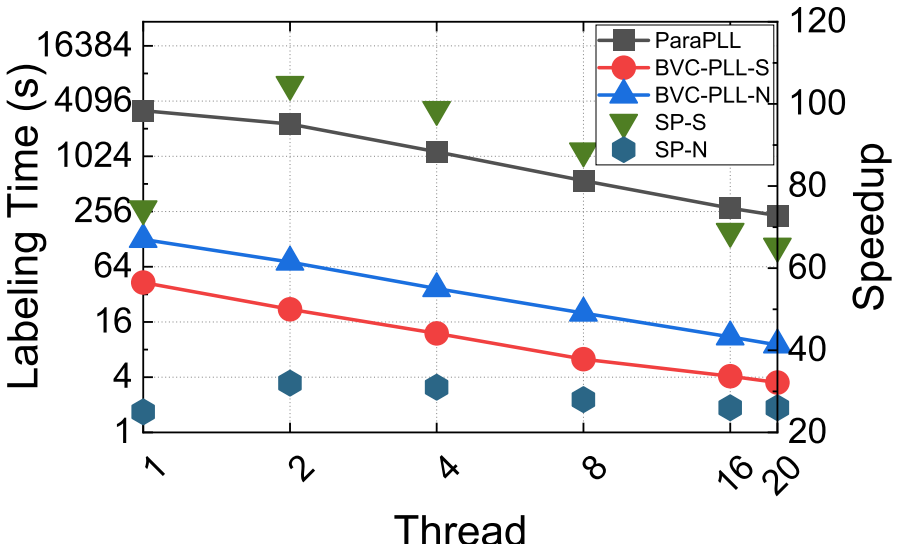


Figure 3.11: Parallel Weighted: BVC-PLL vs. ParaPLL on `GNUT`.

graph, we can see that BVC-PLL is, in general, more than one order of magnitude faster than ParaPLL (even for non-SIMD version).

3.7 Related Work

Many existing efforts aim to efficiently parallelize graph algorithms. Some of them most closely related to our work are discussed here.

Parallel PLL: Multiple parallel PLL approaches [65, 162, 60, 117] allow processing multiple vertices simultaneously. However, they cannot produce the same (and compact) label sets as the original PLL. They also cannot process large graphs as in our implementation. For instance, PLaNT [117] produces a superset of labels initially, so it needs label cleaning after construction. Dong *et al.* combine intra-node parallelism (as proposed in original PLL paper [9]) and inter-node parallelism – the latter not leading to the same results as the sequential computation. We are also made aware that Li *et al.* propose Parallel Shortest-distance Labeling (PSL) that replaces PLL's node-order dependency with a shortest-distance dependence [122]. Our basic vertex-centric algorithm, VC-PLL [102], is

discovered independently as PSL. Moreover, we provide linear algebra analysis and combine the vertex-centric model and batched design to guarantee a smaller memory access cost than PLL.

Graph Processing Paradigms: *Batch processing* is a general idea that is also explored within the context of graph processing [202]. However, this work was focused on addressing problems like skewed distributions and high density. The use of block processing to improve the 2-hop labeling approach is original to our work. From the theoretical side, we can prove that it reduces the number of distance check operations (as shown in supplementary materials aforementioned). It also significantly improves the data locality of remote distance check operations and restricts the overall memory usage.

Our BVC-PLL adopts a *synchronous* paradigm (Bulk Synchronous Parallel (BSP) execution [190]) because its key designs to accelerating the batched processing (like bit operation, and compact data structure) rely on this property. It can be beneficial to extend BVC-PLL to incorporate asynchronous execution ideas (like k-level asynchronous (KLA) [66], and synchronization-avoiding algorithms [67]) to further improve its scalability and performance in the future.

Finally, as we mentioned, new graph processing frameworks (like GraphBLAS(T) [107] and GraphMat [189]) exploit efficient implementations of SpMV and SpGEMM from HPC community. As we also explained, due to the complicated masking operator it appears inefficient to implement BVC-PLL in this linear algebra form.

Graph Framework Implementations: Many popular graph processing engines and frameworks have been developed in recent years. Some of them focus on processing in-memory datasets on one node (e.g., Galois [148], Ligra [175], Polymer [207], GraphGrind [188], etc.), or disk-resident datasets on one node, (Graphchi [115], X-Stream [166], etc.) or performing distributed memory processing (Pregel [136], GraphLab [133], PowerGraph [78], etc.). The in-memory frameworks focus on improving shared memory parallelism and addressing NUMA issues, the out-of-core ones aim to reduce disk traffic, and the distributed ones concern how to efficiently partition graphs, store partitions on multiple

machines, and perform low-cost communication. Many graph frameworks are also designed for GPUs [143, 214, 109, 172, 215, 194, 152, 86, 90, 99, 130, 71, 142]. For instance, works like CuSha [109] and Gunrock [194] target on load balance and memory coalescing optimizations, while works like GraphReduce [172] and Graphie [86] focus on reducing CPU-GPU traffic for the processing of large graphs not fitting in the GPU memory. Certain efforts were also specific to Xeon Phi [181, 42, 100, 25, 156]. In addition, certain graph processing frameworks are designed for hybrid CPU and coprocessors [91, 76, 41, 171, 135, 53]. Moreover, certain compiler-based efforts offer either high-level intermediate representations or domain-specific languages to support high-performance and high-productivity graph programming, such as GraphIt [213] on CPU (and, similarly, IrGL [152] on GPU). To the best of our knowledge, PLL has not been the target of any of these efforts, and there is no previous work supporting a scalable and exact parallel implementation of PLL.

3.8 Chapter Summary

In this work, we proposed VC-PLL, which, to the best of our knowledge, is the first scalable parallelization of Pruned Landmark Labeling (PLL) that is able to produce the same result as the sequential method. We have achieved this by developing new insights that enable mapping the algorithm to a vertex-centric model. We also introduced a new batched execution mechanism for VC-PLL to better support message filtering and remote memory access. The resulting BVC-PLL algorithm can even run faster than the original PLL sequentially. Our experimental results further demonstrate the parallel efficiency and scalability of BVC-PLL and shows its superiority over the most recent ParaPLL algorithms on weighted graphs (using a straightforward extension of BVC-PLL). In our future work, we plan to further investigate how to optimize BVC-PLL on weighted graphs and how to extend it for out-of-core graphs. We also plan to investigate the possibility of implementing the cost-saving mechanism in BVC-PLL for other graph algorithms.

Chapter 4

Speed-ANN: A Parallel Approximate Nearest Neighbor Search Algorithm for the Graph-Based Index

4.1 Introduction

Nearest neighbor search (NNS) is a fundamental building block for many applications within machine learning systems and database management systems, such as recommendation systems [51], large-scale image search and information retrieval [113, 134, 157], entity resolution [89], and sequence matching [24]. NNS has recently become the focus of intense research activity, due to its core role in semantic-based search of unstructured data such as images, texts, video, speech using neural embedding models. In semantic-based search, a neural embedding model transfers objects into *embeddings* in $\mathbb{R}^d$, where d often ranges from 100 to 1000 and N ranges from millions to billions. The task then is to find the K nearest embeddings for a given query. For example, major e-commerce players such as Amazon [149] and Alibaba [210] build semantic search engines, which em-

bed product catalog and the search query into the same high-dimensional space and then recommends products whose embeddings that are closest to the embedded search query; Youtube [49] embeds videos to vectors for video recommendation; Web-scale search engines embed text (e.g., word2vec [145], doc2vec [118]) and images (e.g., VGG [177]) for text/image retrieval [43, 186]. We expect applications built on top of the embedding-based search to continue growing in the future, due to the success and continual advancement of neural embedding techniques that can effectively capture the semantic relations of objects. We also expect the objects to embed will grow rapidly, due to ubiquitous data collections, e.g., through phones and IoT devices.

Since the search occurs for every query, the *latency* and the *accuracy* (*recall*) of the search engine critically depend on the ability to perform fast near neighbor search in the high-recall range. Various solutions for approximate nearest neighbor search (ANNS) have been proposed, including hashing-based methods[93, 52, 10, 11], quantization-based methods [96, 73, 199, 195], tree-based methods [176, 23, 192], and graph-based methods [138, 200, 69]. Among them, graph-based algorithms have emerged as a remarkably effective class of methods for high-dimensional ANNS, outperforming other approaches for very high recalls on a wide range of datasets [16]. As a result, these graph-based algorithms have been integrated with many large-scale production systems [69, 139], where optimizations for fast search and high recall are the focus of a highly active research area and have a clear practical impact.

To provide scalability, existing ANN search libraries often resort to coarse-grained inter-query parallelism, by dispatching each query to a core or even across different machines such that multiple queries can be processed simultaneously [69, 22]. Although inter-query parallelism obtains impressive throughput improvements, it does not help reduce query latency. In particular, online applications often process each query upon its arrival and have stringent latency service level agreements (e.g., a few milliseconds). As the size of datasets grows rapidly, the increased latency of current graph-based ANN algorithms has been restraining ANN-based search engines from growing to large-scale datasets, es-

pecially for high-recall regimes. To provide relevant results with consistently low latency, in this work, we investigate the possibility of intra-query parallelism on individual nodes to meet latency goals.

Although graph-based ANN consists of primarily graph operations, simply dividing the work of graph traversal into multiple threads is insufficient for supporting efficient ANN search, as it cannot efficiently leverage the underline multi-core processors due to complex interactions between graph operations and the hardware threads and memory hierarchy. In our studies, the intra-query parallelism may sometimes hurt search efficiency, because the communication and synchronization overhead increases as we increase the number of cores, making it especially harder to achieve high efficiency.

In this work, we provide an in-depth examination of the graph-based ANN algorithms with intra-query parallelism. Through a series of experiments, we have identified that an intrinsic challenge of the graph search process lies in its long convergence step — existing *best-first search* leads to long convergence steps and introduces heavy control dependencies that limit the upper bounds on speedup by using more cores, as predicted by Amdahl's Law. In our study, we find that, by enlarging the Best-First Search to *Speed-ANN*, the search process can converge in much fewer iterations, suggesting that the search process can achieve better overall performance by running individual queries with more hardware resources. However, exposing the *path-wise* parallelism also changes the search dynamics of queries, leading to additional challenges that may adversely affect search efficiency, which resides in the aspects of redundant computations, memory-bandwidth under-utilization, high synchronization overhead, and irregular accesses caused poor data locality.

Based on the insights from our analysis, we present *Speed-ANN*, a similarity search algorithm that combines a set of optimizations to address these challenges. *Speed-ANN* introduces three tailored optimizations to provide improved performance for graph-based ANN search. First, *Speed-ANN* uses *path-wise parallelism* to divide the search workload to multiple workers in coarse-grained parallelism. Among it, every worker performs its private best-first search in an asynchronous manner to avoid heavy global communication.

Second, *Speed-ANN* employs a *staged search* scheme, which reduces redundant computations caused by over-expansion during a parallel search. Third, *Speed-ANN* is characterized by *redundant-expansion aware synchronization* to lazily synchronize among workers while still providing fast search speed high recall. Finally, *Speed-ANN* provides additional optimizations such as loosely synchronized visiting maps and a cache-friendly neighbor grouping mechanism to improve cache locality during parallel search. In summary, this work makes the following contributions:

1. provides the first comprehensive experimental analysis of intra-query parallelism for ANN search on multi-core architecture and identifies several major bottlenecks to speedup graph-based approximate nearest neighbors in high recall regime;

2. studies how the characteristics of a query vary as the search moves forward from multiple aspects, e.g., by increasing the edge-wise parallelism degree and the dynamics in search queue update positions, which reveals the opportunities and challenges it brings;

3. introduces a search algorithm named *Speed-ANN* with novel optimizations such as *staged path-wise parallelism* and *redundant-expansion aware synchronization* that allow parallel search on graph-based ANN to achieve significantly lower latency with high recall on different multi-core hardwares.

4. conducts thorough evaluation on a wide range of real-world datasets ranging from million to **billion** data points to show that *Speed-ANN* speeds up the search by 1.3×–76.6× compared to highly optimized state-of-the-art CPU-based search algorithms NSG [69] and HNSW [139]. *Speed-ANN* sometimes achieves super-linear speedups in the high recall regime as the number of threads increases, obtaining up to 37.7× speedup over NSG and up to 76.6× speedup over HNSW when using 32 threads. *Speed-ANN* also outperforms a state-of-the-art GPU implementation and provides good scalability.

4.2 Preliminaries

4.2.1 Approximate Nearest Neighbors

Searching for nearest neighbors in high-dimensional space is fundamental in various applications of information retrieval and database management. In this work, the Euclidean space under the l_2 norm is denoted by E^d. The closeness of any two points p_1 and p_2 is defined by the l_2 distance $\delta(p_1, p_2)$ between them [69]. The *Nearest Neighbor Search (NNS)* can be defined as follows [77]:

Definition 1 (Nearest Neighbor Search) *Given a finite point set P of n points in the space E^d, preprocess P so as to answer a given query point q by finding the closest point $p \in P$.*

Please note that the query point q is not in the point set P, i.e. $q \notin P$. The above definition generalizes naturally to the *K Nearest Neighbor Search (K-NNS)* where we want to find $K > 1$ points in the database that are closest to the query point. A naïve solution is to linearly iterate all points in the dataset and evaluate their distance to the query. It is computationally demanding and only suitable for small datasets or queries without a time limit of response. Therefore, it is practical to relax the condition of the exact search by allowing some extent of approximation. The *Approximate Nearest Neighbor Search (ANNS)* problem can be defined as follows [77]:

Definition 2 (ϵ-Nearest Neighbor Search) *Given a finite point set P of n points in the space E^d, preprocess P so as to answer a given query point q by finding a point $p \in P$ such that $\delta(p, q) \leq (1 + \epsilon)\delta(r, q)$ where r is the closest point to q in P.*

Similarly, this definition can generalize to the *Approximate K Nearest Neighbor Search (AKNNS)* where we wish to find $K > 1$ points $p_1, \ldots, p_K$ such that $\forall i = 1, \ldots, K, \delta(p_i, q) \leq (1 + \epsilon)\delta(r_i, q)$ where r_i is the ith closest point to q.

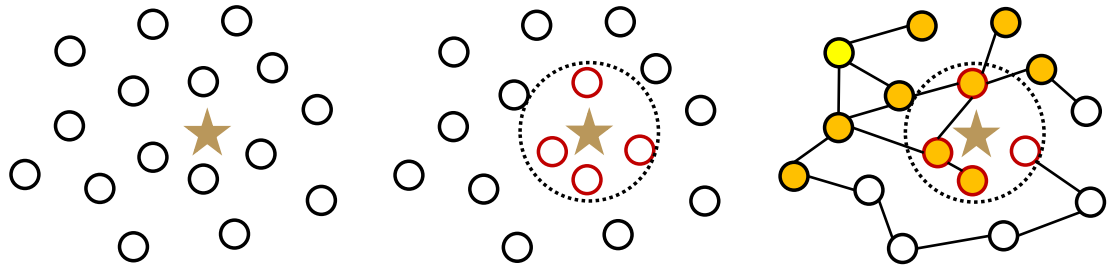
In practice, determining the exact value of ϵ requires high computational overhead. Instead, we use *recall* as the metric to evaluate the quality of the approximation. A high

recall implies a small ϵ, thus a good quality of the approximation. It is defined as the value of the recall. Suppose the approximate points set found for a given query q is R', and the true K nearest neighbor set of q is R, the *recall* is defined as follows [68]:

$$Recall(R') = \frac{|R' \cap R|}{|R'|} = \frac{|R' \cap R|}{K} \tag{4.1}$$

For a particular recall target, i.g. 0.990 or 0.995, our goal is to make the query latency as short as possible.

4.2.2 Graph-based ANN Search



(a) Data points and a query point.

(b) Nearest neighbors of the query.

(c) A graph index and search procedure.

Figure 4.1: An example of graph-based ANNS. Circles are data points. The golden star is query target (not in dataset). Four red circles are its nearest neighbors. Graph-based ANNS builds a graph index on the dataset in 4.1c. The yellow circle is the starting point. Orange circles are visited vertices during the search via Algorithm 7.

Various ANNS solutions have been proposed over decades, e.g., the ones based on trees [14], hashing [92], quantization [12], and graphs [138, 200, 69]. Recently, many experimental results [138, 69] show that graph-based approaches usually outperform others, resulting in the best execution performance and recall. That is because graph-based approaches can better express the neighbor relationship, allowing to check much fewer points in neighbor-subspaces.

Graph-based ANNS relies on a graph structure as its index, in which a vertex represents a data point in the data set and an edge links two points. A vertex p_2 is called a *neighbor*

of a vertex p_1 if and only if there is an edge from p_1 to p_2. Many prior efforts focus on constructing optimal graphs for efficient ANNS [69, 139]—which is not the focus of this work. In contrast, this work is based on the state-of-the-art graph construction approach [69], and aims to parallelize ANNS itself with a thorough study of its bottleneck and a set of advanced techniques addressing these bottlenecks.

Given the graph-based index built ready, Best-First Search algorithm is widely used by many graph-based methods for searching nearest neighbors [56, 15, 84, 105, 138, 139, 87]. Given a query point Q and a starting point P, the algorithm is to search for K nearest neighbors to Q. In the first search step, it visits P's neighbors and computes their distance to Q respectively to choose the closest vertex or candidate, and the next search step starts from the chosen candidate from the last step. All visited vertices are recorded and kept in order according to their distance to Q. The search step stops when the first K visited vertices do not change anymore, which are the final K nearest neighbors. The time spent to find the K nearest neighbors is the query's *latency*.

4.3 Complexities in Graph-based ANN Search for Optimizations

4.3.1 Overview of Graph-based ANN Search

The search procedure in existing similarity graph algorithms, such as NSG [69] and HNSW [139], is a *best-first traversal* that starts at a chosen (e.g., medoid or random) point and walks along the edges of the graph while getting closer to the nearest neighbors at each step until it converges to a local minimum. Algorithm 7 shows the basic idea. In a similarity graph, nodes represent entities in a problem domain (e.g., a video or image in a recommendation system), with each carrying a ***feature vector***. Edges between nodes capture their closeness relationship, which can be measured through a metric distance (e.g., Euclidean). There are a few main differences between the best-first traversal and

classic BFS (breadth-first search) and DFS (depth-first search) algorithms. The first is an ***ordering-based expansion***. During graph traversal, the algorithm selects the closest unchecked node v_i, called an ***active node***, and computes the distance of all neighbors of v_i to the query with their feature vectors (Line 9-13), and *only* inserts promising neighbors into a *priority queue* as new unchecked candidates for future expansion. In this way, the search can limit the number of distance computations needed to converge to near neighbors. Second, different from the BFS and DFS, which traverse all the connected nodes, the best-first search ***converges*** when no new (unchecked) vertex can be found to update the priority queue, leading to a different number of convergence iterations (i.e., the number of while loop iterations in Algorithm 7) for different datasets and queries.

Algorithm 7: Best-First Search (*BFiS*)

Input: graph G, starting point P, query Q, queue capacity L
Output: K nearest neighbors of Q

```
1  priority queue S ← ∅
2  set S's capacity as L
3  index i ← 0
4  compute dist(P, Q)
5  add P into S
6  while has unchecked vertices in S do
7      i ← the index of the 1st unchecked vertex in S
8      mark v_i as checked
       /* Expand v_i                                          */
9      foreach neighbor u of v_i in G do
10         if u is not visited then
11             mark u as visited
12             compute dist(u, Q)
13             add u into S
14 return the first K vertices in S
```

4.3.2 Complexities for Optimizations

The graph traversal process in similarity graphs shares some common complexities with traditional graph processing for performance optimizations, but it also owns some dis-

tinctive features. However, no previous work has given a systematic examination of these complexities. Such knowledge is essential for optimizing similarity graph search, especially at a large scale.

Challenge I: Best-First Search (*BFiS*) takes long iterations to converge, resulting in a prolonged critical path with heavy control dependency. As Algorithm 7 shows, this search consists of a sequence of search steps (Line 6-13) in which the candidates in the current step are determined by the last step. Consider that ANNS usually queries for the top K nearest neighbors, requiring the first K elements in the priority queue to become stable. This state update usually converges slowly (e.g., $>$ 400 search steps or convergence steps to find the 100-nn with 0.999 recall for a million-scale dataset SIFT1M), resulting in a long critical path of execution.

Challenge II: Limited edge-wise parallelism in traversal and memory bandwidth under-utilization. Beyond the aforementioned long convergence steps, it is possible to parallelize the neighbor expansion step (Line 9-13 in Algorithm 7) to reduce the execution time by dividing the neighbors into disjoint subsets and having multiple threads each compute the distance for a subset in parallel, which is called *edge-wise parallelism*. However, this parallelism strategy often achieves sub-optimal performance, because many similarity graphs have a small truncated out-degree on all nodes to avoid the *out-degree explosion problem* [69]. As a result, dividing the work across more worker threads would result in each thread processing only a very small number of vertices. Furthermore, edge-wise parallelism also adds synchronization overhead (e.g., at Line 14) to maintain an ordered expansion. Our preliminary experiment results in Table 4.1 show that the edge-wise parallelism strategy (e.g., running with 64 threads on five datasets) leads to less than 5% of the peak hardware memory bandwidth (∼80 GB/s), indicating a large performance potential remains yet to tap into.

Table 4.1: Memory bandwidth (bdw.) measurement for edge-wise parallelism strategy.

Datasets	SIFT1M	GIST1M	DEEP10M	SIFT100M	DEEP100M
bdw. (GB/s)	1.9	3.3	1.6	1.0	1.1

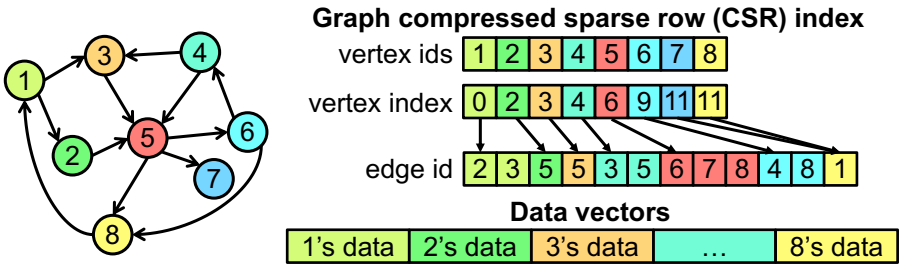


Figure 4.2: The storage structure of the graph-based index. The graph topology is stored in compressed sparse row (CSR) format, and the data vectors are stored in consecutive arrays.

Challenge III: Strict expansion order leads to high synchronization cost. Existing similarity graph search algorithms use a priority queue to maintain the strict priority order of all candidates according to their distances to the queue point. Although it is possible in principle to use a concurrent priority queue that uses locks or lock-free algorithms to synchronize the candidate insertions (Line 14), we observe that the parallel scalability is severely limited by maintaining this strict order because each worker thread only performs distance computations for a few vertices.

Challenge IV: Poor locality brought by irregular memory accesses. Existing similarity graphs often store the graph index (e.g., in the compressed sparse row (CSR) format that contains a vertex array and an edge array) and feature vectors (e.g., in one embedding matrix) separately in memory as different objects, as shown in Figure 4.2. There are two points in this design that lead to inefficiencies. First, the accessed nodes often reside discontinuously in memory, which leads to unpredictable memory accesses. Second, it requires one-level of indirection to access feature vectors, leading to difficulties for memory locality optimizations.

4.4 Design of *Speed-ANN*

Based on the observations from Section 4.4.1, we introduce *Speed-ANN*, a parallel search algorithm that exploits lightweight intra-query parallelism (i.e., path-wise parallelism and

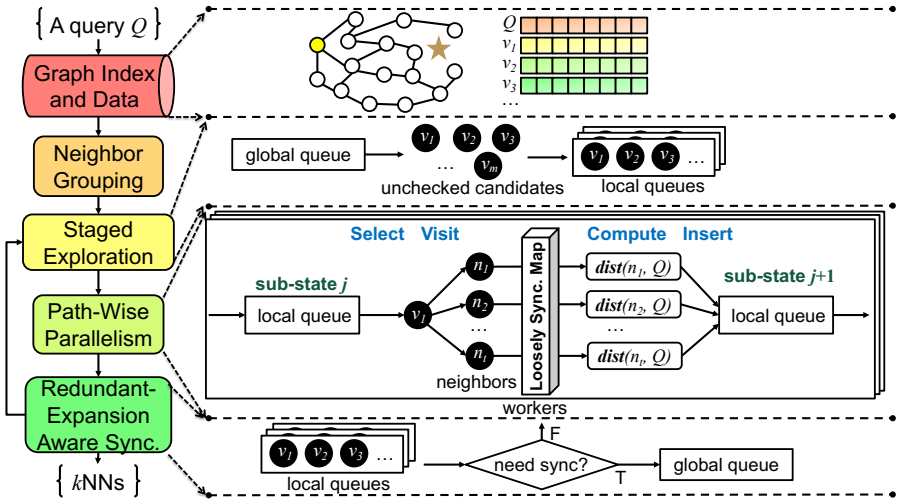
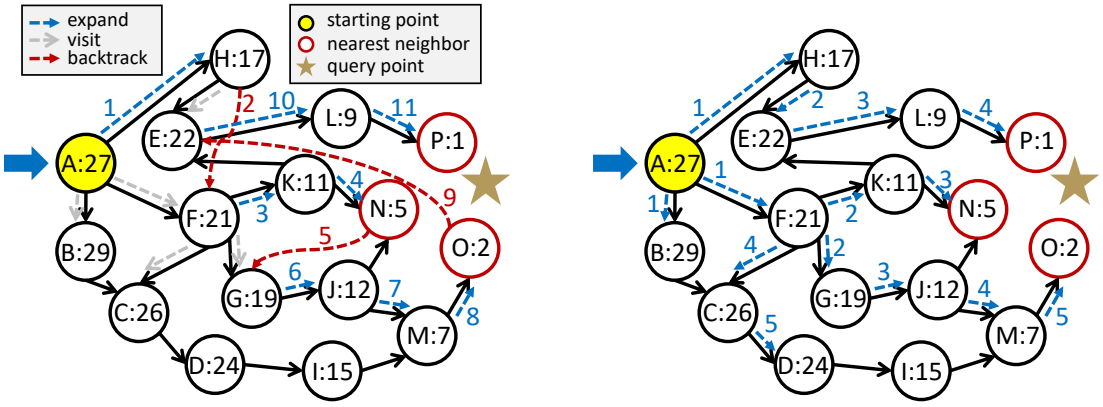


Figure 4.3: Overview of *Speed-ANN*.

edge-wise parallelism) to accelerate the search efficiency of similarity graphs on multi-core CPU architectures. We first provide an overview of our architecture-aware design, and then we discuss technical details.

Figure 4.3 depicts *Speed-ANN*'s overall design that addresses the challenges mentioned in Section 4.3 to perform an efficient similarity graph search. To reduce the long critical path dependency (Challenge I) and increase the amount of parallelism, *Speed-ANN* uses *path-wise parallelism* to deliver coarse-grained parallelism. *Speed-ANN* further introduces a staged search strategy to reduce redundant computations caused by over-expansion during a parallel search. To limit global synchronization overhead (Challenge III), *Speed-ANN* adopts *redundant-expansion aware synchronization* to adaptively adjust synchronization frequency. As such, *Speed-ANN* reduces the number of global synchronizations while still achieving high search accuracy. Besides, *Speed-ANN* uses loosely synchronized visit maps for lightweight communication and also performs neighbor grouping to improve memory locality (Challenge IV).

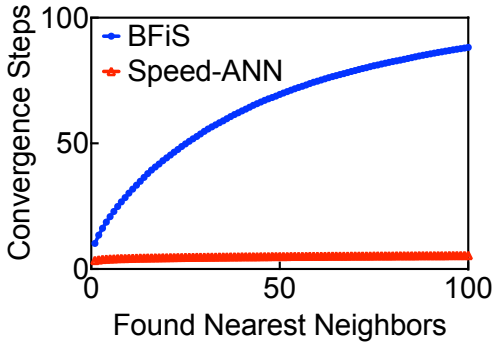


(a) Best-First Search w/ backtrack. (b) *Speed-ANN*: expand top-3 candidates.

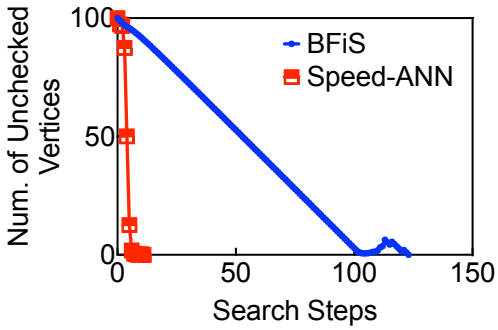
Figure 4.4: Comparison of BFiS and *Speed-ANN*. BFiS needs a long search path with backtrack to find nearest neighbors (11 steps). *Speed-ANN* reduces backtrack and completes with a shorter path (5 steps).

4.4.1 Path-Wise Parallelism

Although it is challenging to parallelize the Best-First Search (*BFiS*) process due to its long critical path and limited edge-wise parallelism, the semantics of the algorithm does not seem to always require a strict order as long as the goal is to minimize the total search time of near neighbors. In this section, we exploit whether the search is robust to deviation from a strict order by allowing concurrent expansion of multiple active nodes. For practical similarity search, e.g., NSG and HNSW, there is no guarantee that a *monotonic search path* always exists for any given query [69]. As a result, the search can easily get trapped into the local optimum. To address this issue, *BFiS* may *backtrack* to visited nodes and find another out-going edge that has not been expanded to continue the search. Figure 4.4(a) illustrates a search path with backtracking. The search starts from vertex A and calculates the distance (indicated by the number following the letter on each vertex) between the three neighbors of A (B, F, and H) and the query point. Because H's distance is locally the smallest, *BFiS* would select H as the active node in the next step. However, given that further expanding H no longer leads to a closer candidate, the search reaches a local minimum and performs a *backtracking* to the next promising candidate F. The search



(a) Convergence steps to find the K-th nearest neighbor in the queue. K is specified by the x-axis. Although *BFiS* can find the first neighbor quickly, it still needs many steps to find all others.



(b) Numbers of unchecked candidates (vertices) in the queue after every search step. While *BFiS* needs 100+ steps to converge, *Speed-ANN* only needs 10+ steps. Values are the average of 10K queries.

Figure 4.5: *Speed-ANN* results in much less search steps than *BFiS*. Dataset is SIFT1M. They have the same $L = 100$. *Speed-ANN* has $M = 64$, where M means the top M unchecked candidates.

process then may backtrack multiple times until it either finds the near neighbor (e.g., O) or exhausts the search budget.

Backtracking creates additional dependencies in *BFiS* process and increases the convergence steps to find near neighbors. However, many of these backtracking dependencies can be **"fake" dependencies** if we perform *path-wise parallelism*, e.g., *by expanding multiple active nodes concurrently, it is possible to shorten the convergence steps by starting early at one of those backtracking points.* As an example, while it takes 11 steps to find the near neighbor in Figure 4.4(a), it only takes 5 steps in Figure 4.4(b) if we expand nodes F, G, J, M right after expanding their parent nodes.

Based on this insight, we introduce *Speed-ANN*. In this scheme, the priority order is relaxed such that in each step, **top M unchecked candidates** are selected as active nodes for expansion instead of just the best candidate.

***Speed-ANN* exposes hidden parallelism.** The relaxation of the order enables two levels of parallelism: the *path-wise parallelism* where multiple threads can concurrently expand the search frontier, and the *edge-wise parallelism* when expanding an individual

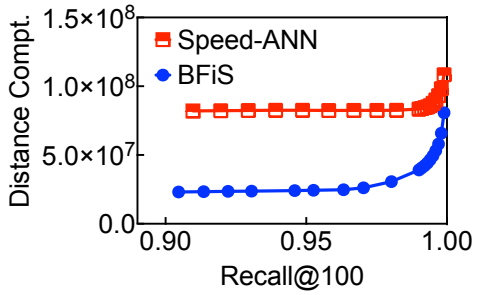


Figure 4.6: Distance computations of *BFiS* and *Speed-ANN*, where $M = 64$.

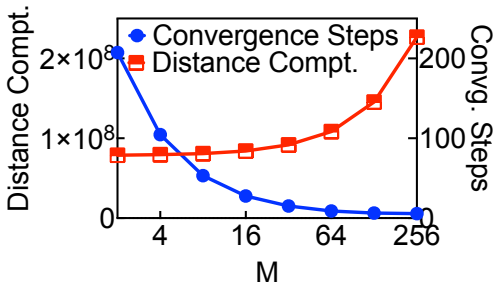


Figure 4.7: Distance computations and search steps of *Speed-ANN* when M changes.

active node. Moreover, instead of having a global queue to maintain strict expansion orders among all workers, each worker has a local priority queue, which allows a thread to exploit a small number of *order inversions* (i.e., allowing a worker thread to locally select and expand active nodes ahead of the global order), which can dramatically reduce communication, synchronization, and coordination between threads.

***Speed-ANN* converges faster to near neighbors.** One key benefit of *Speed-ANN* is that it significantly shortens the *convergence steps* compared to *BFiS*. Figure 4.5 shows the comparison results of convergence steps between *BFiS* and *Speed-ANN*. The results are measured on dataset SIFT1M using 10K queries with 0.90 recall target, and M is set to 64. *Speed-ANN* takes on average 3.4, 5.0, and 5.4 steps to find the 1st, 50th, and 100th nearest neighbor, respectively, whereas *BFiS* takes 10.1, 69.4, and 88.1 steps, respectively. From another aspect, *Speed-ANN* takes much fewer steps to finish examining all the unchecked vertices in S than *BFiS*, as shown in Figure 4.5b. Both results indicate that *Speed-ANN* has a much faster convergence speed than *BFiS*.

Tree-based Expansion View. Similar to the classical DFS/BFS, *BFiS* naturally introduces an expansion tree: the root node T_r of the tree is the starting vertex P in graph G; the children of a tree node T_i (corresponding to a graph vertex v_i) are the unvisited neighbors of v_i. The expansion of *BFiS* bears many similarities to DFS, as each time, it will expand only one leaf node. However, different from DFS, which expands the one with

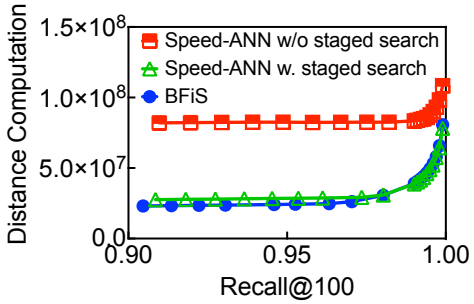
the most depth, *BFiS* expands the one which is closest to query Q. Thus, we have the same concepts of *backtracking* and *Steps* in *BFiS*.

The power of *Speed-ANN* is that it expands the M leaves simultaneously of the tree, which are M nearest neighbors of query Q among all the leaves of the *current* expansion tree. This effectively searches/extends M paths in parallel instead of a single path (in *BFiS*). Thus, *Speed-ANN* can potentially reduce the total number of *steps* of *BFiS* by a factor of M times, as for k *Steps*, *Speed-ANN* can expand kM tree nodes/leaves. Further, due to the hardware capability, at the same *time*, *Speed-ANN* can process M leaves/paths expansion as only what is in *BFiS* (one single leave or path expansion), leading to the low latency of query processing. We also note that the *BFiS* becomes a special case of *Speed-ANN* where $M = 1$, and both parallelization are under Bulk Synchronous Parallel (BSP) model [190] though *BFiS* has rather limited parallelism to explore.

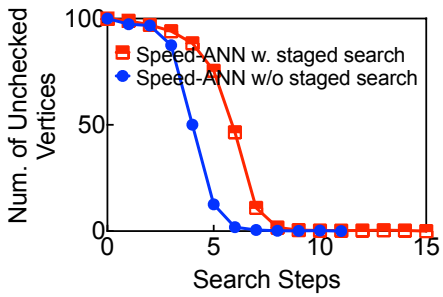
4.4.2 Staged *Speed-ANN* to Avoid Over-Expansion

Despite the faster convergence speed, intra-query parallel search incurs additional challenges in increased distance computations. Figure 4.6 shows that to reach the same recall, *Speed-ANN* often leads to more distance computations than *BFiS*. *Speed-ANN* has more computations because path-wise parallelism allows a query to take fewer steps to reach the near neighbors by avoiding fake dependencies from backtracking but it also introduces more computations to explore additional paths. Furthermore, we observe that although the convergence steps continue to decrease with larger M, the number of distance computations also increases dramatically, as shown in Figure 4.7.

When the number of parallel workers is large, the search speed of *Speed-ANN* might be sluggish because the over-expansion of neighbors can result in many redundant computations during the entire search process. To avoid unnecessary distance computations caused by over-expansion, we take a *staged* search process by gradually increasing the expansion width (i.e., M) and the number of worker threads every t steps during the search procedure. The intuition is that the search is less likely to get stuck at a local minimum at



(a) Distance computation of *BFiS*, *Speed-ANN* w/o staged search, and *Speed-ANN* w. staged search. Staged search avoids additional distance computation from over-expansion.



(b) Number of unchecked candidates after each search step. *Speed-ANN* carries fast convergence properties.

Figure 4.8: Comparison between *Speed-ANN* without staged search and with staged search: distance computation & search steps. $M = 64$.

the beginning of the search, so the best-first search with a single thread can already help the query to get close to near neighbors. As the search moves forward, it becomes more likely that a query will get stuck at a local minimum and requires backtracking to escape from the local minimum. Therefore, path-wise parallelism with a larger expansion width in later phases can better help reduce the convergence steps. We find that a simple staging function works well in practice: when the search begins, we first set a starting value and a maximum value for M. The starting value is usually one, and the maximum value can be as large as the number of available hardware threads. Subsequently, for every t steps (e.g., $t = 1$) we double the value of M until M reaches its maximum. Figure 4.8a shows that staged *Speed-ANN* reduces the amount of redundant significantly in comparison to *Speed-ANN* without a staged search and leads to distance computations close to *BFiS*. On the other hand, staged *Speed-ANN* is able to converge as almost fast as *Speed-ANN* without staged search, as shown in Figure 4.8b. These results indicate that our staged search method still achieves fast convergence speeds without incurring too many distance computations caused by over-expansion through the parallel search on a large number of workers.

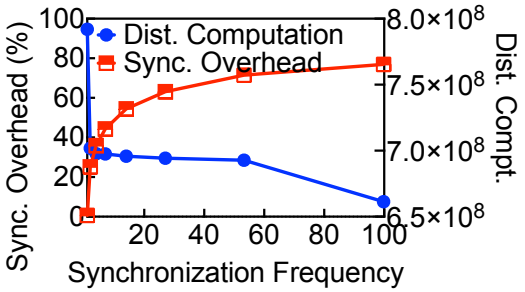


Figure 4.9: *Speed-ANN*'s sync. overhead and distance computation vs. sync. frequency.

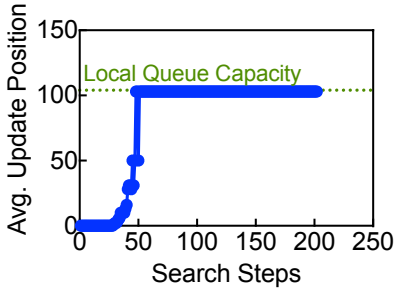


Figure 4.10: A query's average update positions during searching.

4.4.3 Redundant-Expansion Aware Synchronization

As mentioned in Section 4.3, yet another big performance bottleneck in intra-query parallelism resides in the synchronization overhead. Figure 4.9 shows how the global synchronization frequency influences the synchronization overhead (calculated by synchronization time divided by overall execution time) and the overall distance computations. All results in this figure return the same recall value. It shows that the synchronization overhead increases significantly when the synchronization frequency grows. We also find that order inversion (with insufficient synchronization) slows down the search convergence and results in growing distance computations (as shown in Figure 4.9). This is because, with insufficient synchronization, worker threads keep searching their own (unpromising) areas without benefiting from other threads' latest search results that may lead to faster convergence. This study demonstrates that a proper synchronization frequency is desired to achieve high system performance.

Measuring redundant expansion via update positions. To unleash the full power of multi-core systems, *Speed-ANN* performs a unique form of lazy synchronization so that worker threads do not need to synchronize at every search step in most cases. Especially, our synchronization scheme is *redundant-expansion aware*, which means instead of having a strict order through the entire convergence steps, we allow some relaxation of the order as long as each worker thread is still performing some effective search and the global order

becomes consistent again after a large amount of redundant expansion has been detected. In this work, we propose a new way to measure the effectiveness of intra-query parallel search based on the *update positions* of workers. When a worker expands an unchecked candidate, its neighbors are then inserted into the worker's local queue, and the update position is defined as the *lowest (best)* position of all newly inserted candidates. Thus, *the average update position* is the mean of all update positions of workers. Figure 4.10 demonstrates how an example query's average update position changes during the search steps without global synchronization. It shows that the average update position increases gradually to the local queue capacity and resides there to the end. When the average update position is close to the queue capacity, it indicates that most workers are searching among unpromising areas and cannot find good enough candidates to update their local results. Therefore, the average update position can be used as a metric to determine if all workers need to synchronize their local results to adjust the search order. We would like to note that there could be more than one metric to decide when to perform the lazy synchronization. We leave it as an open research question and more advanced methods might lead to better performance improvements.

Algorithm 8 describes how to use the average update position as the metric to decide when to perform a lazy synchronization. Given the queue capacity L and a position ratio R, the threshold of the average update position to do synchronization is set as $L \cdot R$. If the *checker* finds the average update position is greater than or equal to the threshold (Line 2), it returns `true` indicating a global synchronization in Algorithm 9. Empirically, the ratio R is close to 1.0, such as 0.9 or 0.8. The input vector of all update positions is updated by workers regularly without locks. The return flag is only written by the *checker* who is assigned among workers in a round-robin manner.

Table 4.2 shows preliminary results about the performance comparison between adaptive synchronization and no-synchronization. No-synchronization means each thread performs its local search and only combines the results in the end. The results show that adaptive synchronization is able to improve search efficiency with fewer distance computa-

Algorithm 8: CheckMetrics() (Update Position Version)

Input: vector of update positions U, queue capacity L, position ratio R, number of workers T
Output: true or false

```
1 ū ← average positions of elements in U
2 if ū ≥ L · R then
3     return true
4 else
5     return false
```

Table 4.2: Comparison between no-sync. and adaptive sync. 8 threads on SIFT1M for Recall@100 0.9. Adaptive sync. check workers' dynamic status and merge queues adaptively. `Lt.` denotes latency. `Compt.` denotes distance computation.

Dataset	no-sync.		adaptive sync.	
	Lt. (ms.)	Compt.	Lt. (ms.)	Compt.
SIFT1M	1.16	125.3 M	0.70	33.1 M

tions. Overall, the reduced synchronization and distance computation from our redundant-expansion-aware synchronization is especially helpful for path-wise parallelism on a large number of workers, because global synchronization across multiple threads is still expensive and not very scalable as the number of cores increases.

Putting It Together. Algorithm 24 describes the overall algorithm of *Speed-ANN*. At the beginning of each global step, the global queue evenly divides its unchecked candidates among all local threads. After that, each worker performs a local best-first search based on its own local queue of sub-states (Line 11 to Line 21). Different from the global state that involves updating the global queue, a worker's local *sub-state* is the state of its private queue. In a local search step, a worker expands its own best unchecked candidate and updates its private queue accordingly. Before the global queue's state is updated, a worker can have multiple sub-states of its own private queue. A worker continues expansion until `CheckMetrics()` raises a flag for merging or it has no unchecked candidates left locally. In a round-robin way, a worker is assigned as the *checker*. The checker's duty is to check (as what `CheckMetrics()` does) if all workers need to synchronize their sub-states by merging

all private queues into the global queue. If so (Line 19), all workers will stop their local search and merge their queues.

Algorithm 9: *Speed-ANN* Intra-Query Parallel ANN Search

Input: graph G, starting point P, query Q, queue capacity L, number of workers T

Output: K nearest neighbors of Q

```
1  expansion width M ← 1
2  global priority queue S ← an empty queue with capacity L
3  local priority queues LS ← T empty queues with capacity L
4  compute dist(P, Q)
5  add P into S
6  while true do
7      divide all unchecked vertices from S into LS
8      if all LS are empty then
9          break
10     foreach worker t out of M in parallel do
11         while LS[t] contains unchecked vertices and doMerge is false do
12             vertex v ← the first unchecked vertex in LS[t]
13             mark v as checked
14             foreach neighbor u of v in G do
15                 if u is not visited then
16                     mark u as visited
17                     compute dist(u, Q)
18                     add u into LS[t]
19             if t is the checker and CheckMetrics() returns true then
20                 doMerge ← true
21                 assign the next checker in round-robin way
22     merge LS into S
23     if M < T then
24         M ← 2M
25 return the first K vertices in S
```

4.4.4 Additional Optimizations

Loosely Synchronized Visiting Map. There is one potential bottleneck to multi-threaded parallel scaling in Algorithm 9 on our target architectural platforms (multi-core systems). Consider visiting a neighbor of a candidate. This is typically after a check and

then an update to a visiting map to ensure that a vertex is calculated once (Line 15-16). During path-wise parallelism, the visiting map is shared by all workers to indicate if a vertex has been visited. Since multiple threads may access the shared visiting map concurrently, locking or lock-free algorithms are required if we still want to ensure a vertex is visited only once. However, this approach involves a significant scalability bottleneck, because it leads to lock contention and sequentialization of updating the visiting map.

We observe that the ANN search algorithm is still correct even if a vertex is calculated multiple times because the local candidates are guaranteed to be merged back to the global priority queue and the visiting map is also guaranteed to have *eventual consistency* the next time of global synchronization. Furthermore, by inserting memory fences, cache coherence further ensures that the updated visiting map is visible to other cores. Due to the potential out-of-order execution in processors, modern multi-core processors provide *fence* instructions as a mechanism to override their default memory access orders. In particular, we issue a fence after a thread updates the visiting map to guarantee a processor has completed the distance computation of the corresponding vertex and has updated the visiting map (otherwise, there is no guarantee the updated visiting map is visible to other cores before next step of global synchronization).

By doing the loosely synchronized local search, we observe that the search algorithm only performs a very small percentage of additional distance computations (less than 5%) for SIFT1M (and similar for other datasets) with 8-way parallelism. This reduces the overhead from synchronization by 10% and allows us to avert the issue of non-scaling locking across the multi-threading search. This optimization was also considered by Leiserson and Schardl [119] (termed as "benign races") for their parallel breadth-first search algorithm. Furthermore, we use a bitvector to implement the visiting map instead of a byte-array. This optimization allows the cache to hold the largest possible portion of the visiting map and therefore improves the data locality for memory accesses.

Cache Friendly Neighbor Grouping. When a feature vector is loaded into memory for distance computation, modern CPU architectures actually automatically load vectors

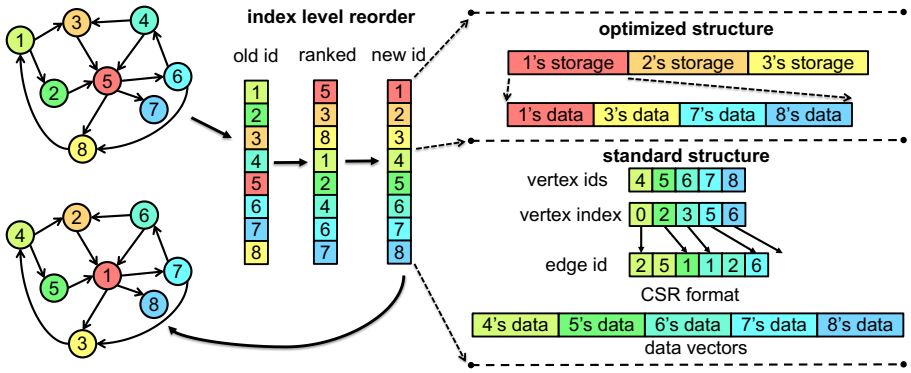


Figure 4.11: Example of neighbor grouping and hierarchical data storage. Vertices are ranked according to their in-degree. Vertices are first reordered into new ids according to their ranks. High ranked vertices are stored in an optimized index where every vertex's neighbors' data are stored in contiguous locations right after its own data to make expanding cache-friendly. Other low ranked vertices are stored in a standard index where the graph index and data vectors are stored separately.

from nearby memory locations as well. Our neighbor grouping technique taps into this feature to mitigate the two levels of irregularity mentioned in Section 4.3.

First, *Speed-ANN flattens* the graph indices by placing the embeddings of neighbor vertices in contiguous memory, which would avoid one-level of implicit memory addressing and enables a thread to pre-fetch neighbor feature vectors once an active node is selected. Second, *Speed-ANN* also regroups nodes, such that vertices that are likely to be visited during the graph traversal are already pre-load into the CPU memory and cache. Together, these two optimizations increase the cache hit rate and help speed up the search process.

One caveat of this approach is that it introduces additional memory consumption, because two neighbor lists may share the same vertex as a common neighbor. It is therefore may require more memory consumption than the original approach. To avoid increasing the memory consumption, *Speed-ANN* takes a hierarchical approach by regrouping only a subset of vertices. In particular, *Speed-ANN* divides a graph to a two-level index as shown in Figure 4.11, where only the top-level vertices have their neighbors flattened and stored in contiguous memory, and the bottom-level index stores other vertices using the standard structure. In this work, we explore two strategies to graph division: **Degree-**

centric, which puts high in-degree nodes to the top-level of the indices. The intuition is that high in-degree nodes are more frequently accessed, and therefore improving their locality would benefit the most for the overall search efficiency. **Frequency-centric**, which exploits query distribution to figure out which nodes are more frequently accessed and puts those frequently accessed nodes into an optimized index. Section 4.5 evaluates both strategies and shows that *Speed-ANN*'s neighbor grouping strategy brings 10% performance improvements with selecting only 0.1% vertices as the top level for a dataset with 100M vertices.

4.5 Evaluation

This evaluation proves that *Speed-ANN* can significantly reduce the ANN search latency with the proposed effective parallel optimizations.

Evaluation Objectives. This evaluation targets five specific evaluation objectives: (1) **latency**—demonstrating that *Speed-ANN* outperforms existing ANN search algorithms (NSG [69], HNSW [139], and a parallel version of NSG) by up to 76.6× speedup in terms of the latency without any precision compromise; (2) **scalability**—confirming that *Speed-ANN* scales well on modern multi-core CPU architectures with up to 64-cores; (3) **optimization effects**—studying the performance effect of our key optimizations (path-wise parallelism, staged search, redundant-expansion aware synchronization, and cache friendly neighbor grouping) on overall latency, distance computations, synchronization overhead, etc; (4) **portability**—proving *Speed-ANN* has good portability by evaluating it on other multi-core CPU architectures; (5) **practicability**—showing that *Speed-ANN* is practical, applicable to extremely large datasets (e.g., `bigann`) with billions of points and outperforming an existing GPU implementation (i.e. Faiss) by up to 6.0× speedup with 32 CPU cores.

Implementation. A natural question is if our implementations can leverage any existing graph libraries (e.g., Ligra [175]); however, it turns out this is very difficult due to multiple

Table 4.3: Characterization of datasets. `Dim.` denotes the dimension of the feature vector of each point, `#base` denotes the number of points, and `#queries` denotes the number of queries.

Dataset	Dim.	#base	#queries
SIFT1M	128	1M	10K
GIST1M	960	1M	1K
DEEP10M	96	10M	10K
SIFT100M	128	100M	10K
DEEP100M	96	100M	10K

reasons: First, ANN algorithms do not pass messages between vertices. The computation only happens between a vertex and the query point. Second, ANN algorithms need to do computation with vector values. Third, ANN algorithms need to keep output results sorted. This requires extra efforts to maintain the results especially after synchronization between workers. Fourth, existing libraries' optimization techniques for general graph processing are usually not suitable for ANN algorithms. For example, Ligra [175] can switch between push and pull modes according to the number of active vertices. However, in ANN algorithms, the number of active vertices is capped by the expected output number of nearest neighbors, making the switching never happen. Besides, *Speed-ANN* runs in a semi-synchronous pattern with delayed synchronization among workers, which is different from the BSP model [190] with strict synchronization after every parallel step. Therefore, we have our high-performance implementation of those algorithms without using existing graph processing libraries. Our proposed ANN algorithms are written in C++ compiled by Intel C++ Compiler 2021.4.0 with "`-O3`" option. We use OpenMP 5.0 to handle the intra-query parallelism.

Platform and Settings. Unless otherwise specified, all major experiments are conducted on Intel Xeon Phi 7210 (1.30 GHz) with 64 cores and 109 GB DRAM (*KNL* for short). *Speed-ANN* sets the average update position ratio as 0.8 for SIFT1M, GIST1M, and SIFT100M, and 0.9 for DEEP10M and DEEP100M.

Datasets. This evaluation uses five datasets that are characterized in Table 4.3. SIFT1M

and GIST1M are from the datasets[1] introduced by Jégou et al. [97]; SIFT100M is sampled from the SIFT1B (`bigann`) introduced by Jégou et al. [98]; DEEP10M and DEEP100M are sampled from DEEP1B[2] which is released by Babenko and Lempitsky [18]. These are common datasets for ANN algorithms evaluation [69].

Baselines. *Speed-ANN* is compared with two state-of-the-art sequential ANN search implementations, NSG[3] [69] and HNSW[4] [139]. NSG employs a search algorithm called Best-First Search, and HNSW uses its own best-first search algorithm corresponding to its hierarchical index. The hyperparameters used for building their indices are set as default values as long as the authors provided them. Otherwise, several values are tested and the best performance is reported.

For NSG, we use its optimized version of searching for SIFT1M, GIST1M, and DEEP10M, and its normal version for SIFT100M and DEEP100M because of memory limit. We also implement a Naïve Parallel NSG that parallelizes neighbor visiting during expansion.

4.5.1 Search Latency Results

Figure 4.12 compares the latency of *Speed-ANN*, NSG, and HNSW. *Speed-ANN* uses 32 threads while NSG and HNSW are sequential approaches. The *query latency* is the average latency of all queries, i.e., it equals the total searching time divided by the number of queries. All methods search the 100 nearest neighbors for every query (i.e. $K = 100$). The measure *Recall@100* is calculated according to Formula 4.1 with $K = 100$, which means the ratio of ground-truth nearest neighbors in searching results for each query. The value of Recall@100 is the average of all queries. All recalls mentioned in this section are Recall@100 if not specified.

Figure 4.12 shows that *Speed-ANN* outperforms NSG and HNSW on all five datasets. *Speed-ANN*'s latency advantage increases with the growth of recall requirement, and it

[1] `http://corpus-texmex.irisa.fr/`
[2] `https://sites.skoltech.ru/compvision/noimi/`
[3] `https://github.com/ZJULearning/nsg`
[4] `https://github.com/nmslib/hnswlib`

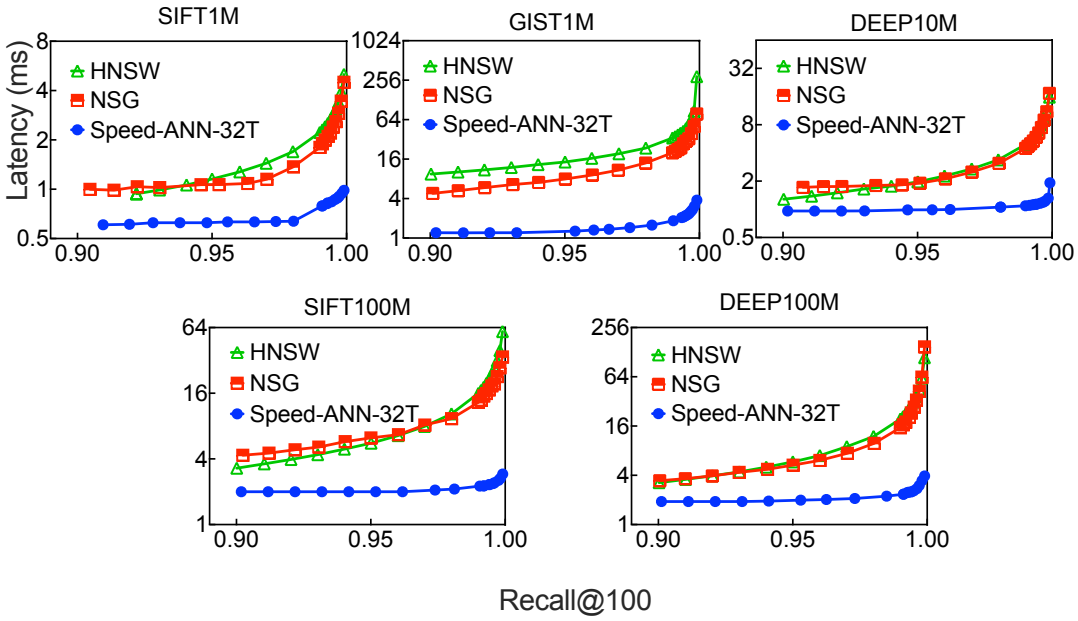


Figure 4.12: Latency (ms) comparison among *Speed-ANN*, NSG, and HNSW on five datasets. *Speed-ANN* use 32 threads.

performs significantly better for high recall cases (e.g., from 0.995 to 0.999). For the cases of Recall@100 (R@100) being 0.9, 0.99, and 0.999, on all five datasets, *Speed-ANN* achieves 2.1×, 5.2×, and 13.0× geometric mean speedup over NSG, and 2.1×, 6.7×, and 17.8× over HNSW, respectively. As the recall becomes 0.999, *Speed-ANN* achieves up to 37.7× speedup over NSG on DEEP100M, and up to 76.6× speedup over HNSW on GIST1M. *Speed-ANN* achieves significantly better performance for high recall situations mainly because of two reasons. First, *Speed-ANN*'s path-wise parallelism effectively reduces convergence steps (comparing with NSG) because it is not easily trapped at a local optimum and can explore a local region more quickly than a sequential search. This is particularly critical for a large graph (e.g., DEEP100M) to achieve high recall, where a query can more easily get stuck at a local optimum. Second, *Speed-ANN* has better data locality from using aggregated L1/L2 cache provided by multiple threads, in contrast to a sequential search where only private cache can be used. Further profiling results are provided in Section 4.5.3.

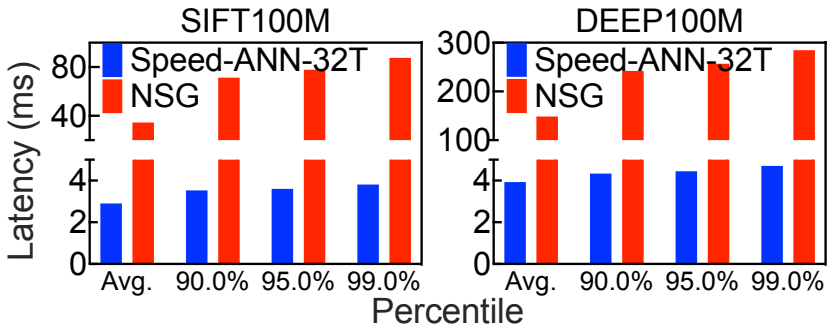


Figure 4.13: Percentile latency of *Speed-ANN* & NSG. Recall: 0.999.

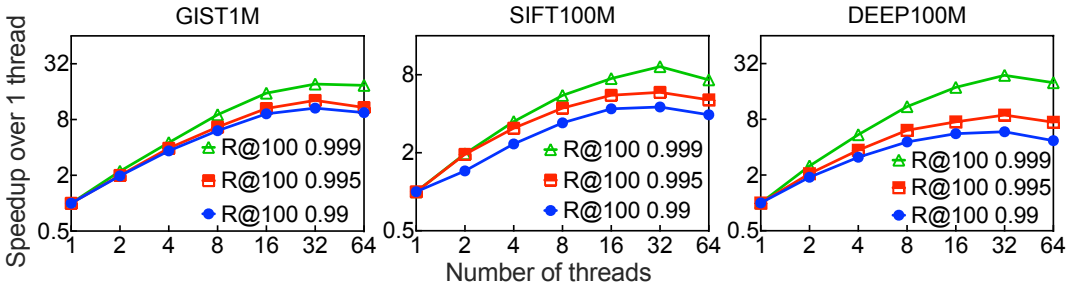


Figure 4.14: Speedup of *Speed-ANN* over 1 thread on three datasets.

Impact on Tail Latency. For online inference, tail latency is as important, if not more, as the mean latency. To see if *Speed-ANN* provides steady speed-ups, we collect the 90th percentile (90%tile), 95th percentile (95%tile), and 99th percentile (99%tile) latency from running NSG and *Speed-ANN* on SIFT100M and DEEP100M in Figure 4.13. The results show that while NSG's 99%tile increases significantly by 154% and 91% for SIFT100M and DEEP100M, respectively, the *Speed-ANN*'s 99%tile increases only by 31% and 19% over its average for SIFT100M and DEEP100M, respectively. *Speed-ANN* leads to a relatively smaller increase in tail latency presumably because intra-query parallel search is particularly effective in reducing latency on long queries.

4.5.2 Scalability Results

Scaling with An Increasing Number of Threads. Figure 4.14 reports the speedup of 1- to 64-thread *Speed-ANN* over 1-thread on three datasets for three selected recall

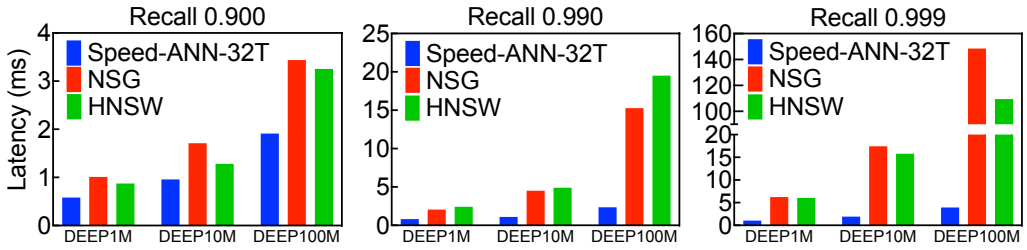


Figure 4.15: Scalability with varied graph sizes for *Speed-ANN*, NSG, and HNSW on DEEP1M, DEEP10M, and DEEP100M. *Speed-ANN* uses 32 threads.

(0.99, 0.995, and 0.999), respectively. It shows that this scalability increases as the target recall grows because of the increased distance computations that offers more parallelism opportunities. The geometric mean speedup of all datasets for the highest recall (0.999) is 9.6×, 11.1×, and 9.2× for 16-, 32-, and 64-thread, respectively. *Speed-ANN* only scales to 16 threads for SIFT1M because SIFT1M is too small without enough workload for more threads. *Speed-ANN* demonstrates super-linear speedup (up to 16 threads) for 0.999 recall on GIST1M and DEEP100M. This phenomenon will be further analyzed in Section 4.5.3. *Speed-ANN* does not scale well for 64 threads due to various reasons. For datasets with high dimensional vectors (e.g. GIST1M), 32-thread *Speed-ANN* has saturated memory bandwidth already. For others (e.g., SIFT1M, DEEP10M, and DEEP100M), extra distance computations of too many unnecessary expansions gradually dominate overall execution.

Scaling with An Increase of the Graph Sizes. Our experiments also evaluate the scalability with varied dataset sizes (DEEP1M, DEEP10M, and DEEP100M) for *Speed-ANN*, NSG, and HNSW, respectively. Figure 4.15 reports the latency results of *Speed-ANN*, NSG, and HNSW for the recall of 0.9, 0.99, and 0.999, in which *Speed-ANN* uses 32 threads. *Speed-ANN* constantly outperforms NSG and HNSW, and the heavier workload, the better performance *Speed-ANN* shows. More specifically, with the growth of dataset size, the speedup of *Speed-ANN* over NSG and HNSW increases. For example, when the recall is 0.999, the speedup of *Speed-ANN* over NSG grows from 5.9x to 27.8x when the dataset size changes from 1M to 100M. This trend becomes increasingly obvious with the

growth of the recall. The results reflect that *Speed-ANN* is particularly effective and offers more speedups than existing search methods for larger graphs.

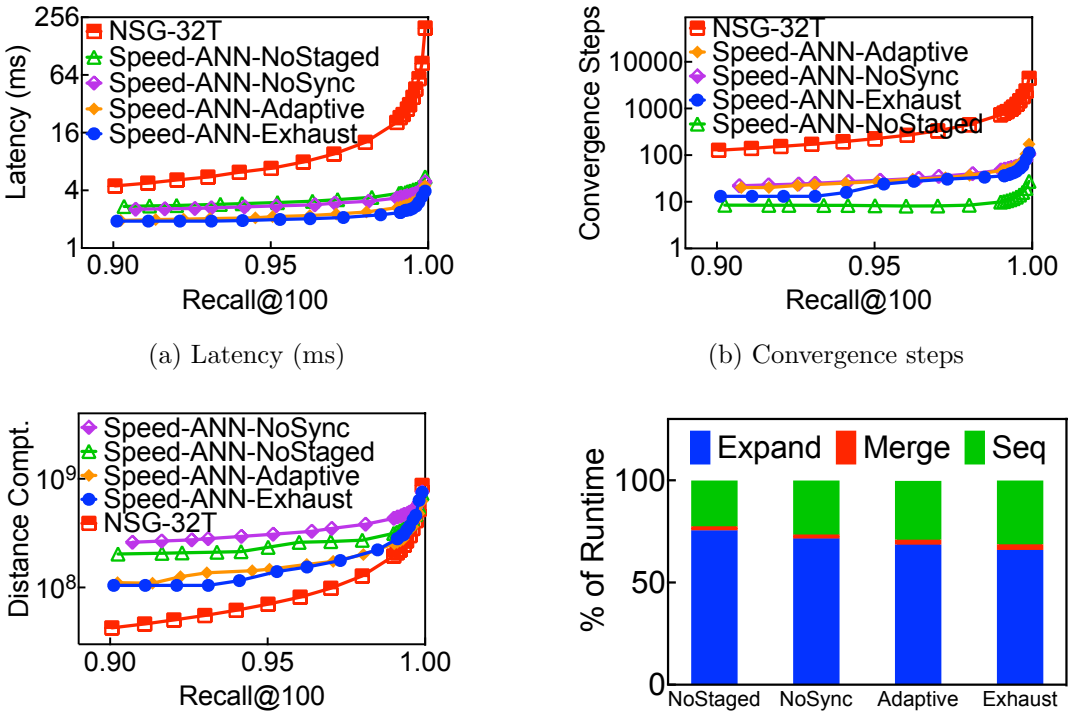
4.5.3 Analysis Results

This section performs a series of experiments to show where *Speed-ANN*'s improvements come from. It first compares *Speed-ANN*'s performance with several alternative parallel search schemes. (i) NSG-32T: This configuration extends NSG with path-wise parallelism only (e.g., M=1). (ii) *Speed-ANN-NoStaged*: This configureation is *Speed-ANN* but without using the staged search process. (iii) *Speed-ANN-NoSync*: This configureation performs path-wise parallelism but never synchronizes among workers until the very end. (iv) *Speed-ANN-Exhaust*: This configureation uses an exhaustive search to preprocess the dataset and obtain the proper synchronization settings. It should have the best latency performance, although requiring more than ten hours of tuning for the given dataset. (v) *Speed-ANN-Adaptive*: This is the configuration described in Section 4.4, which adopts redundant-expansion aware synchronization.

For this comparison, we report results on DEEP100M dataset with 32 threads in Figure 4.16. Other datasets and threads show the same trend, thus we omit them due to the space constraint.

Effects on Latency. Figure 4.16a first reports the latency results of all five versions when we change recall from 0.90 to 1.00. Compared with NSG-32T, *Speed-ANN-NoStaged* has 4.9× speedup on average for all recall cases, because of the convergence iterations reduction from path-wise parallelism. *Speed-ANN-Exhaust* has an extra 1.5× speedup over *Speed-ANN-NoStaged* mainly due to its reduction in synchronization optimization. *Speed-ANN-Exhaust* achieves slightly better performance than *Speed-ANN-Adaptive* (e.g., 1.1× speedup). However, *Speed-ANN-Adaptive* does not require the expensive offline tuning process as *Speed-ANN-Exhaust*.

Effects on Convergence Iterations. Figure 4.16b profiles the convergence steps of the five parallel methods. Each point is averaged from all queries. NSG-32T results in the most



(a) Latency (ms)

(b) Convergence steps

(c) Distance computation

(d) Runtime breakdown

Figure 4.16: Synchronization study w/ 32 threads on DEEP100M.

steps of convergence; while *Speed-ANN-NoStaged* results in the fewest. All three versions of *Speed-ANN* result in comparable convergence steps to *Speed-ANN-NoStaged* that are much less than NSG-32T. This is because *Speed-ANN-NoStaged* employs a fixed and relatively large number of multiple paths throughout the searching, resulting in the most aggressive exploring. *Speed-ANN-Adaptive* and *Speed-ANN-Exhaust* adopt a staged search, which slightly increases the convergence steps but significantly reduces distance computations. Meanwhile, *Speed-ANN-NoSync* suffers more divergence compared to *Speed-ANN-Adaptive* and *Speed-ANN-Exhaust*.

Effects on Distance Computation. Figure 4.16c profiles the number of distance computations for those five methods. *Speed-ANN-NoStaged* with a fixed value of $M = 32$ leads to more distance computations than NSG-32T, *Speed-ANN-Exhaust*, and *Speed-ANN-Adaptive* to achieve the same recall (especially for low recall cases). While completely

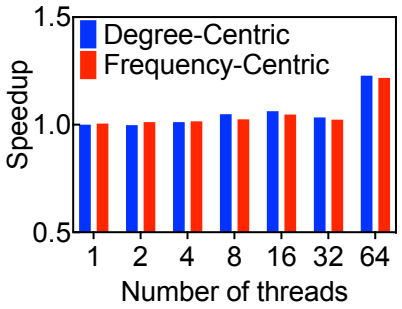


Figure 4.17: Speedup of *Speed-ANN*'s neighbor grouping on DEEP100M for recall 0.999.

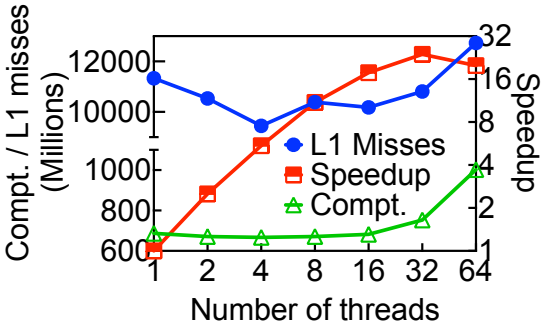


Figure 4.18: *Speed-ANN*'s L1 misses, speedup over 1-thread, and distance computation w/ recall 0.999 on DEEP100M.

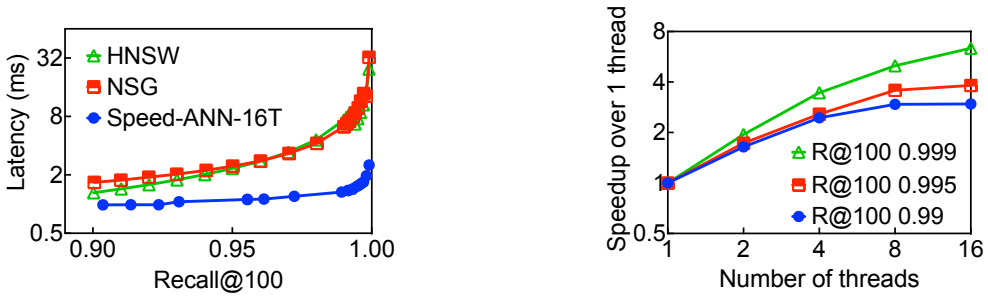
removing synchronization, *Speed-ANN-NoSync* has the most distance computations than others. However, as shown in Figure 4.16a, it still achieves lower latency than *Speed-ANN-NoStaged* because synchronization overhead can dominate the total search time when the number of parallel workers is large.

Effects on Synchronization Overhead. Figure 4.16d reports the execution time breakdown of our four approaches. It splits the whole execution time into three parts: Expanding part (*Expand*), Merging part (*Merge*), and Sequential part (*Seq*). *Expand* denotes the parallel phase of a query that workers expand their unchecked candidates. It consists of computing distances and inserting visited neighbors into their queues. *Merge* denotes the phase that workers merge their local queues into a global queue after they complete expanding. It reflects the major synchronization overhead. Other sequential execution of a search is included in *Seq*. All results are for recall 0.999. Figure 4.16d shows that redundant-expansion aware synchronization strategy effectively mitigates the synchronization overhead, allowing *Speed-ANN-Adaptive* to achieve a similar portion of synchronization overhead as *Speed-ANN-Exhaust*.

Effects of Neighbor Grouping. Our fully optimized *Speed-ANN*-32T also includes another optimization, i.e. neighbor grouping. Figure 4.17 shows that our two proposed strategies (*degree-centric* and *frequency-centric*) outperform no-grouping by up to 1.22× and 1.21× speedup, respectively, when we change the thread numbers from 1 to 64. This

speedup mainly comes from the reduction of the last-level cache miss and TLB (translation lookaside buffer) cache miss. This profiling result is omitted due to the space constraint.

Super-linear Speedup Observation. Section 4.5.2 shows that *Speed-ANN* results in an interesting super-linear speedup (up to 16 threads) for 0.999 recall on GIST1M and DEEP100M. Figure 4.18 reports three profiling results, distance computations, L1 cache misses, and performance speedup for DEEP100M when changing the thread numbers from 1 to 64. The left x-axis shows the first two profiling results while the right x-axis shows the last one. It shows that as we increase the number of threads, the L1 cache misses and distance computations first decrease and then increase. This causes the super-linear speedup for the cases whose thread numbers are less than 16. Distance computation shows this trend because: on the one hand, path-wise parallelism helps avoid the search from being trapped by local minimal candidates and quickly pick up promising searching paths for more nearest neighbors; on the other hand, too many exploring threads cause unnecessary expansion of non-promising candidates, increasing distance computations. L1 cache miss shows this trend because multi-threads increase the total size of L1 cache.



(a) Latency (ms). (b) Speedup.

Figure 4.19: Portability study: DEEP100M on Skylake.

4.5.4 Portability Evaluation

To evaluate the portability, *Speed-ANN* is also tested on Intel Skylake architecture, Xeon Gold 6138 (2.0 GHz) with 20 cores and 187 GB DRAM (`Skylake` for short). For the sake of space saving, only results on DEEP100M are presented as other datasets show

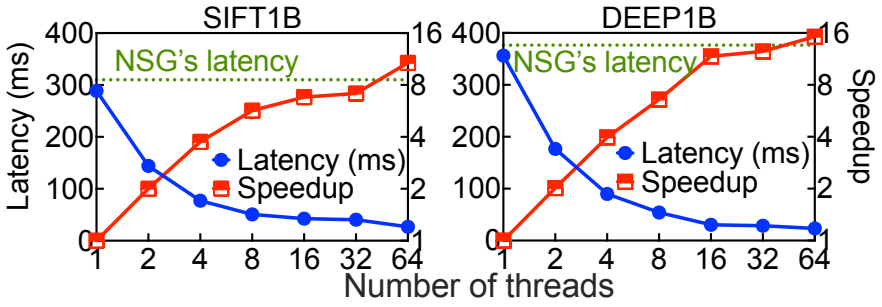


Figure 4.20: Performance comparison of *Speed-ANN* and NSG on SIFT1B (`bigann`) and DEEP1B. *Speed-ANN*'s speedup is over its 1-thread. Recall is 0.9.

the same trend. Figure 4.19a compares the latency of *Speed-ANN*, NSG, and HNSW, in which, *Speed-ANN* uses 16 threads. It shows a similar trend as previous, i.e., *Speed-ANN* outperforms NSG and HNSW for all recall. For 0.9, 0.99, and 0.999 cases, *Speed-ANN* achieves 1.7×, 4.5×, and 12.9× speedup over NSG, and 1.3×, 5.3×, and 9.7× over HNSW, respectively. Figure 4.19b evaluates *Speed-ANN*'s scalability. Similarly, target recall 0.999 can achieve the best speedup over 1 thread, and speedup for 0.999 is 4.9× and 6.3× for 8 threads and 16 threads, respectively.

4.5.5 Practicality Evaluation

This section evaluates *Speed-ANN*'s practicality with two case studies: 1) evaluating it on very large datasets, SIFT1B (`bigann`) and DEEP1B that contain over 1 billion data vectors; 2) comparing it with a state-of-the-art GPU implementation.

Billion-Scale Datasets. This experiment is conducted on a particular machine with Xeon Gold 6254 (3.10 GHz) 72 cores and 1.5 TB memory because of the large memory requirement. Figure 4.20 compares the latency of *Speed-ANN* and NSG. *Speed-ANN* uses up to 64 threads, and the recall target is 0.9. When using 64 threads, *Speed-ANN* outperforms NSG with 11.5× and 16.0× speedup for SIFT1B and DEEP1B, respectively. As we increase the number of threads, *Speed-ANN* shows sub-linear speedup because of the well-known NUMA effect (this machine has 4 NUMA domains). These results indicate

Table 4.4: Latency comparison of *Speed-ANN* and Faiss-GPU on five datasets. `Lt.` means *Latency*. `OOM` means *out of memory*. Faiss-GPU's index format is IVFFLat. *Speed-ANN* uses 32 threads.

Datasets	Faiss-GPU w/ IVFFlat		*Speed-ANN*-32T on KNL	
	R@100	Lt. (ms.)	R@100	Lt. (ms.)
SIFT1M	0.52	0.87	**0.91**	**0.61**
GIST1M	0.36	7.25	**0.90**	**1.21**
DEEP10M	0.62	5.79	**0.90**	**0.96**
SIFT100M	OOM	OOM	**0.90**	**2.00**
DEEP100M	OOM	OOM	**0.90**	**1.91**

the effectiveness of our method in speeding up the search process on billion-scale datasets.

Compare with a GPU Implementation. We also compare *Speed-ANN* with a GPU-based large-scale ANN search algorithm [106] in Faiss library [6]. The GPU experiments are conducted on an NVIDIA Tesla P100 with CUDA 10.2. Faiss is set to have one query in every batch, because we focus on reducing the online query latency to meet stringent latency requirement. Table 4.4 shows the latency comparison results on five datasets. *Speed-ANN* uses 32 threads on KNL. For the SIFT100M and DEEP100M, Faiss-GPU complains of out-of-memory errors. For other datasets, *Speed-ANN* outperforms Faiss-GPU with 1.4× to 6.0× speedup and much better recall, which indicates that *Speed-ANN* can effectively achieve faster search speed than GPU-based search algorithms on CPUs, which are often much cheaper than GPUs.

4.6 Related Work

This section describes prior efforts closely related to our work.

Graph-based ANN. Navigating Spreading-out Graph (NSG) [69] is one of the state-of-the-art graph-based indexing methods. It is a close approximation of Monotonic Relative Neighborhood Graph (MRNG) that ensures a close-logarithmic search complexity with limited construction time. NSG (and many other graph-based methods [56, 15, 84, 105, 138, 139, 87], e.g., FANNG [87], NSW [138], and HNSW [139]) rely on best-first search to

process queries. Other graph-based methods include [121, 208, 94, 120, 21, 127, 159, 58, 22]. In contrast to these efforts that mostly focus on indexing building, our work *for the first time* unveils the real bottleneck of intra-query graph search, and significantly reduces search latency (particularly for billion-scale graphs) with multiple advanced architecture-aware parallel techniques.

Non-Graph based ANN Methods. Hashing-based methods [93, 52, 10, 11] map data points into multiple buckets with a certain hash function such that the collision probability of nearby points is higher than the probability of others. Quantization-based methods [96, 73, 199, 195, 192] (e.g., IVF [97], and IMI [17]) compress vectors into short codes to reduce the number of bits needed to store and compute vectors. Faiss [106] is implemented by Facebook with produce quantization (PQ) methods. Tree-based methods (e.g., KD-tree [176] and R* tree [23]) hierarchically split the data space into lots of regions that correspond to the leaves of a tree structure, and only search a limited number of promising regions. Flann [146] is a library based on KD-tree. Graph-based methods have been proved to outperform these non-graph-based methods by checking fewer data points to achieve the same recall [69, 16, 121, 62]. Another line of work that is closely related to *Speed-ANN* is to accelerate ANN search by varied accelerators, e.g., FPGA [206] and GPU [106].

Parallel Graph Systems. Many graph engines and frameworks have been developed in the past decade. Some of them are shared-memory, focusing on processing in-memory datasets within a computation node, e.g., Galois [148], Ligra [175], Polymer [207], Graph-Grind [188], GraphIt [211], and Graptor [191]. Some are distributed systems, e.g., Pregel [136], GraphLab [133], and PowerGraph [79]. Some efforts focus on out-of-core designs (e.g., GraphChi [114] and X-Stream [166]) and process large graphs with disk support. Many graph frameworks are also on GPUs, such as CuSha [109], Gunrock [194], GraphReduce [172], and Graphie [86]. These graph systems are either based on a vertex-centric model [136] or its variants (e.g., edge-centric [166]). These models are in the strict BSP model [190]. Different from them (and other asynchronous graph traversal efforts [86, 85]), *Speed-ANN* uses delayed synchronization that is in the spirit of stale synchronization [88]

where workers are running in an asynchronous fashion before synchronization, which makes it possible to retain high parallelism and also a low amount of distance computations. Moreover, as aforementioned in the implementation, due to the uniqueness of ANN, it is challenging to migrate many of these system designs to *Speed-ANN* directly.

Generic Search Schemes. Many efforts aim to parallelize various search schemes (e.g., BFS [175], DFS [147], and Beam search [141]). Although *Speed-ANN*'s path-wise parallelism design is inspired by prior parallel search algorithms on graphs, our work has a very different focus and aims to: 1) identify that ANN's convergence bottleneck comes from the fact that ANN requires to find many targets that may be (or not be) present in the graph—a search scenario that is very different from many previous graph search problems; 2) several optimizations specifically tailored for reducing the number of distance computations and synchronization overhead from path-wise parallelism, such as staged search and redundant-expansion aware synchronization.

4.7 Chapter Summary

This work looks into the problem of accelerating graph-based ANN search algorithms on multi-core systems, performing comprehensive studies to reveal multiple challenges and opportunities to exploit intra-query parallelism for speeding up ANN searching. Based on the detailed performance characterization, we propose *Speed-ANN*, a similarity search algorithm that takes advantage of multi-core CPUs to significantly accelerate search speed without comprising search accuracy. *Speed-ANN* consists of a set of advanced parallel designs, including path-wise parallelism, staged search, redundant-expansion aware synchronization, loosely synchronized visit map, and cache friendly neighbor grouping, systematically addressing all the identified challenges. Evaluation results show that *Speed-ANN* outperforms two state-of-the-art methods NSG and HNSW by up to 37.7× and 76.6× on a wide range of real-world datasets ranging from million to billion data points.

Chapter 5

Optimizing Computational Graph-based Deep Neural Network Inference on Microcontrollers

5.1 Introduction

In this chapter, we explore the range of applications to the machine learning applications running on embedded systems such as microcontrollers units (MCUs). Tiny machine learning (TinyML) is a fast-growing field that is the intersection of machine learning and embedded systems [125]. On the one hand, machine learning (ML) applications have been employed in various hardware, from large-scale clusters to personal mobile phones, while the demand to deploy ML applications to new platforms has been growing continuously. On the other hand, the number of embedded devices has reached 250 billion and still has a strong projected growth over coming years [216]. The combination of machine learning and embedded systems gives rise to new opportunities for TinyML which might bring the application scope of ML to a new level.

Compared to machine learning applications that run on large machines or even mobile phones, applications running locally on tiny devices have three major advantages. The first advantage is low cost. Tiny devices such as MCUs are very cheap. For example, the average selling price of 23-bit MCUs is $0.20 in 2020 [182]. Enabling ML applications on those tiny devices can allow more people to enjoy the convenience that ML techniques bring. The second advantage is low network latency. Local computation on the device does not depend on a connection to the cloud or servers, which is suitable for applications that require stringent response latency or that run in extreme environments without an internet connection. The third advantage is high privacy. Some small applications, such as wake word detection, can be highly integrated with daily life. As the tiny device does not have an internet connection, all input data monitored and collected are processed locally without privacy concerns.

The major challenge of deploying ML applications to embedded devices is the limited computational resources, especially the computational power of processors and the size of memory [144]. First, the limited power of the processor might cause high response latency that is longer than expected. Second, the scarcity of memory limits the range of applications that can be deployed. For example, a state-of-the-art ARM Cortex-M4 MCU has only 324KB SRAM and 2MB Flash storage, which is impossible to run some large off-the-shelf deep learning models (DNNs).

Taking into account the hardware constraints, the objective of this work is to improve the inference performance of deep neural networks (DNNs) on MCUs and also mitigate the memory constraints by taking full advantage of hardware resources. First, for the neural networks that can directly fit in the memory, we optimize the operations to provide better latency performance. Second, for the large models that cannot fit in on-chip memory, we exploit the off-chip memory and study its performance effects.

In these preliminary results, we use fine-tuned loop unrolling to improve the performance of the convolution operation and use lightweight quantization to reduce the overhead of re-quantization. Moreover, we test the performance effects of off-chip memory.

The ultimate goal in the future is to design an end-to-end inference framework that can provide optimal latency performance for various neural networks.

5.2 Background and Challenges

5.2.1 Neural Network Execution

A given neural network can be regarded as a computational graph where the vertices correspond to operations (also called layers or operators). An operation takes one or multiple input tensors and produces an output. The edges in the computational graph correspond to the dependencies between individual operations, as one predecessor operation's output will become its successor operations' input.

The inference proceeds by evaluating operations one by one in their topological order in the graph. Some operations, such as convolution, have weights or parameters that will load into memory during evaluating. Meanwhile, an operation requires buffers for its inputs and output to be present in memory. Otherwise, the operation cannot commence. The total size of its weights, inputs, and output make up the memory footprint of the operation, and the total size of weights or parameters determines the size of the neural network or model.

5.2.2 Resource Scarcity of Microcontrollers

As a device with limited resources, a microcontroller typically has a low-frequency processing unit from tens to hundreds of megahertz, such as 180 MHz for STM32F469NI [184], equipped with 128–2048 KB of on-chip memory. The on-chip memory consists of two parts, i.e., the SRAM that supports read and write by programs, and the Flash memory that is read-only during execution.

While the Flash acts as the storage to hold the executable code and static data, SRAM acts as the memory buffer where the code can allocate, read, and write temporary variables. Unlike general computers, there is no intermediate memory level between the

SRAM and Flash. However, an MCU board can equip peripherals, including off-chip memory, such as QSPI, SDRAM, and SD-card. Those off-chip memories usually have larger capacity but lower speed than on-chip memory.

Most MCUs are bare-metal devices and do not have operating systems. After a compiled program is loaded into its Flash, the MCU keeps running the program once it has been powered on. The memory limit for the program comes from two parts. First, the total size of the binary program itself cannot exceed the Flash size (usually 1–2 MB). Second, the program cannot have a memory footprint larger than the SRAM size (typically 16–1024KB).

Likewise, neural network inference on MCUs has the same memory requirements coming from the two types of on-chip memory. Weights or parameters of a model are immutable and compiled into the executable code as static data stored in the Flash. All intermediate tensors that are allocated when evaluating an operation at runtime would have to be stored in SRAM. Therefore, the model size and peak memory footprint are constrained by the capacities of the Flash and the SRAM, respectively.

5.2.3 Challenges

Considering the constraints from hardware resources, the challenges of neural network inference on MCUs can be divided into two parts:

1. For small models that are able to fit in the Flash and SRAM, how to improve their inference latency?

2. For large models that cannot even fit in the Flash, how to deploy them on MCUs for inference?

5.3 Optimizations

We studied some optimization techniques to improve deep learning inference performance. For small networks that can fit in memory, we studied lightweight quantization and loop

unrolling to improve their inference latency. For large networks that can not fit in the on-chip memory, we tried exploiting off-chip memory to execute the inference.

5.3.1 Lightweight Quantization

Quantization is a technique that can provide a significant decrease in not only a neural network's memory footprint but also its whole size. In general, quantization reduces the number of bits used to represent each weight of the model so that the total size is reduced by the same factor [150]. In the TinyML scenario, this is very important as memory scarcity is a major constraint for MCU hardware. Besides, quantization enables the use of fixed-point instead of floating-point encoding. In other words, weights are represented as integers (e.g., `int8`) rather than floating-point types (e.g., `float32`), which allows operations to be performed using integer instructions. This is of benefit because some MCUs do not have floating-point units. Thus their floating-point instructions must be emulated in software, introducing a large overhead [158]. Besides, presenting values in 8-bit integers also enables the usage of SIMD instructions supported by some MCUs.

We inspected the inference workflow of quantized neural networks in CMSIS-NN. In the beginning, it takes quantized neural networks whose weights are encoded in `int8` in advance. When executing a convolution operation, the intermediate results are stored in `int32` formats at first and then are re-quantized into `int8` as the final results. The re-quantization scheme is shown in Formula 5.1.

$$Q_c = \frac{Scale \times Q_a \times Q_b + 2^{30}}{2^{31}} \gg Shift \tag{5.1}$$

Here the Q_a and Q_b are in `int8` and their product is in `int32`. The final result Q_c is in `int8` after quantization. The values $Scale$ and $Shift$ are in `int32` and are calculated according to current filters during the inference, and this scheme contains at least two multiplications and one division. According to profiling results, the overhead of re-quantization computa-

tion takes 23.0% of the total inference latency for a given MobileNetV2 model [167], not including the calculation overhead of $Scale$ and $Shift$.

In order to reduce the re-quantization overhead, we employed a new lightweight quantization scheme with the help of quantization-aware training. The scheme is shown in Formula 5.2.

$$Q_c = (Q_a \times Q_b) \gg Shift \tag{5.2}$$

Here the value of $Shift$ is given after training, so it does not cost calculation overhead, and this scheme only has one multiplication and a shift operation. The preliminary evaluation results of latency are shown in the evaluation section.

5.3.2 Loop Unrolling

Convolution is an important operation in neural networks, especially in Convolution Neural Networks (CNNs). It convolutes the input tensors with its filters by computing their inner product. A typical implementation of convolution is using *im2col* (image-to-column) approach [61], which has been highly successful in neural network frameworks such as TensorFlow Lite for Microcontrollers (TFLM) [54] and CMSIS-NN [116]. For example, CMSIS-NN uses *im2col* to transform two columns of input matrices and compute their inner produce with the kernels in every iteration. Therefore, the convolution turns out to be implemented as matrix multiplication that consists of nested for-loops.

Loop unrolling is a traditional optimization technique for improving loop performance. It occurs by extending the necessary code manually for the loop to occur multiple times in the loop body and then updating the conditions and the iteration index accordingly [112]. When unrolling a loop by an *unrolling factor* of K, the loop body repeats K times, and the iteration space is reduced by K.

The benefits of loop unrolling come from three parts. First, it reduces loop overhead

since the total number of iterations is reduced to produce the same output. Second, it allows some variables to be kept in registers for fast access. Third, increase the opportunity of instruction level parallelism [36]. However, a too-large unrolling factor might hurt the performance because of the increased code size and register spilling [39]. Therefore, how to choose a proper unrolling factor is a challenge to achieving optimal performance for operations that depend on loop-structure computation.

We examined TFLM and found that it used CMSIS-NN's operation implementation. Besides, it implemented the convolution operation by using matrix multiplication which applied loop unrolling with the factor of 2. We tested different unrolling factors to see their effects on inference latency. The preliminary results are shown in the evaluation section.

5.3.3 Off-Chip Memory

Besides on-chip memory, off-chip memory is the memory outside the MCU core. Common off-chip memory available for MCUs includes Quad-SPI NOR Flash (QSPI) and SDRAM. QSPI corresponds to the on-chip Flash and can hold the executable code. Meanwhile, SDRAM corresponds to the on-chip SRAM and can be used for intermediate variables during the execution. The capacity of QSPI and SDRAM is usually tens of megabytes, which is larger than the on-chip memory. However, the access speed for them is slower.

Some neural networks cannot be executed on MCUs because their whole sizes are too large. As mentioned in Section 5.2.2, a model is compiled into the executable code before being loaded into the Flash of an MCU device. If the model size is too large, it cannot be held by the Flash directly.

The intuitive idea is to exploit the available off-chip memory to hold large models when running on MCUs. The challenge is how to determine which part of the network should be put on slow memory. In the preliminary testing, we put the whole model into the off-chip memory, allocated intermediate variables in off-chip memory, and tested its inference latency. The preliminary results are shown in the evaluation section.

5.4 Preliminary Evaluation

This evaluation section shows some preliminary results of techniques that were introduced in this work, including loop unrolling, lightweight quantization, and off-chip memory.

Evaluation Objective. The major evaluation objective is the latency – how long it takes to finish the inference compared with the baseline. Specifically, the objective is to show how the loop unrolling, lightweight quantization, and off-chip memory influence inference latency.

Platform. The MCU device used for testing is STM32F469NI-DISCO [183]. It has an ARM Cortex-M4 processor with a frequency of up to 180MHz, 324KB SRAM for temporary variables, and 2 MB Flash for binary code. It is also equipped with other peripherals including 16 MB Quad-SPI NOR Flash (QSPI) and 16 MB SDRAM.

Baseline. The inference framework used as the baseline is TensorFlow Lite Micro (TFLM) [54]. It is an interpreter-based framework designed for deploying deep learning models to embedded hardware. It also allows hardware vendors to provide platform-specific optimization for some particular operations. In the evaluation, TFLM employs high-performance ARM CMSIS-NN libraries [116] for common deep neural network operations, such as convolution, to provide optimized performance.

Models. For testing loop unrolling and lightweight quantization, we use a modified MobileNetV2 model that has 3.3M MACs (multiply-accumulate operations). However, this model is filled with random `int8` values without training, so we did not test the accuracy of our re-quantization method. For testing off-chip memory, we use another larger modified MobileNetV2 model called *mbv2-w0.3-r80_imagenet.tflite* that is provided by MCUNet [125]. It has 7.3M MACs and can barely be fitted in the Flash.

Implementation. The implementation of the optimization techniques is written in C++. For loop unrolling and lightweight quantization, we implement the corresponding modified convolution operation and re-quantization operation and then integrate them into the original TFLM framework. For off-chip memory, we add methods to change the work-

flow and put the whole binary code into QSPI and allocate new variables in SDRAM, respectively, to test its effects on latency performance.

5.4.1 Effects of Lightweight Quantization

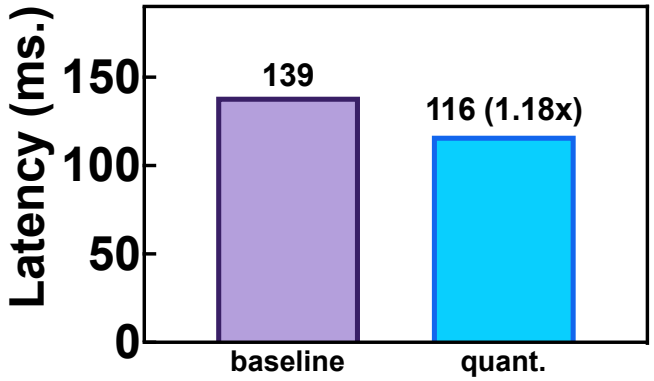


Figure 5.1: Lightweight quantization latency performance and combined with loop unrolling.

Figure 5.1 shows the inference latency performance of lightweight quantization (*quant.*) and also the combination of quantization and loop unrolling (*quant.+unroll.*). With only the lightweight quantization, it can have 1.18× speedup over baseline. As the model has not been trained yet, the effect of quantization on accuracy is unknown at this moment. Providing a suitable quantization method while guaranteeing the accuracy is part of future work.

5.4.2 Effects of Loop Unrolling

Figure 5.2 shows the inference latency comparison between the baseline and different loop unrolling settings for the convolution operation. The computation kernel contains a 2-level for-loop. For the convenience of discussion, here *out*X-*in*Y denotes the outer loop uses unrolling factor of X, and the inner loop uses Y. For example, the baseline uses the unrolling setting of *out*2-*in*2, meaning that the unrolling factors for its outer and inner

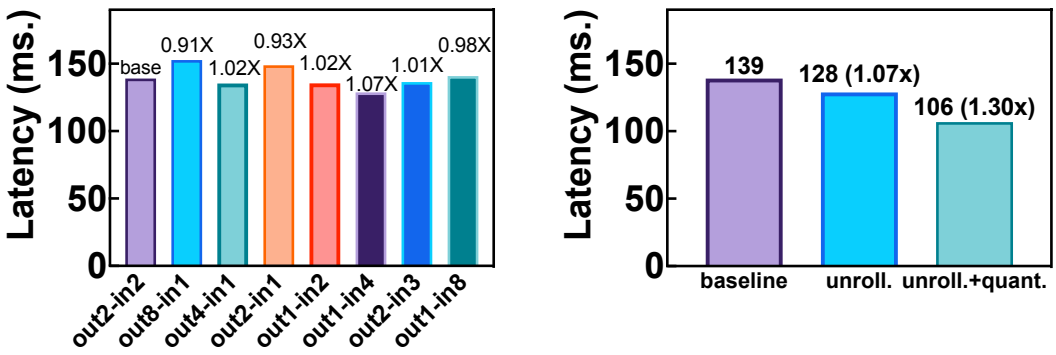


Figure 5.2: Loop rolling factor settings and latency performance.

loops are both 2.

We tested several different unrolling settings. The best performance comes with *out*1-*in*4 that has 1.07× speedup over the baseline. The speedup does not show a significant improvement in latency for two reasons. First, only the convolution operations use fine-tuned loop unrolling settings, and there are other operations in the neural network, although convolution accounts for about 70% total latency. Second, the baseline already exploits loop unrolling to improve performance, although the unrolling factor might not be universally optimal. How to automatically choose a proper setting for a given neural network and a device is a part of future work. Together with lightweight quantization, it can have 1.30× speedup over baseline.

5.4.3 Effects of Off-Chip Memory

Figure 5.3 shows the effects of off-chip memory upon inference latency. The neural network for testing can barely be fitted in the on-chip Flash. In total, the profiling contains four memory usage patterns: 1) *Flash+SRAM* uses only on-chip memory. The binary code compiled with the model is loaded in the Flash, and the program creates temporary variables in the SRAM; 2) *Flash+SDRAM* keeps the binary code in the Flash while creating temporary variables in the off-chip SDRAM. 3) *QSPI+SRAM* uses the off-chip QSPI to house the binary code and uses on-chip SRAM to hold intermediate variables.

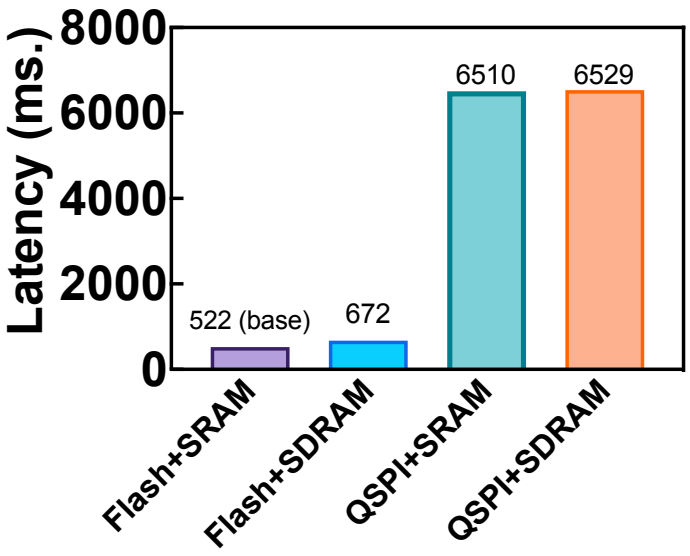


Figure 5.3: Off-chip memory latency performance.

4) *QSPI+SDRAM* uses both the off-chip QSPI and SDRAM for holding the binary code and temporary variables, respectively.

The results show that using SDRAM instead of SRAM increases the latency to 1.28× over the baseline. In contrast, using QSPI instead of Flash increases the latency to 12.47× over the baseline, which is a large performance downgrade. Using both SDRAM and QSPI causes the latency to increase to 12.50×. Here neither the binary code nor the temporary variables use a mix of both on-chip and off-chip memory. How to properly choose the part of data to be put on off-chip memory in order to mitigate the performance decline is part of future work.

5.5 Related Work

Hardware. Most published work about neural network applications on MCUs ([125, 124, 167, 144, 20, 54]) uses the devices based on ARM processors for testing purposes, such as Cortex-M4 and Cortex-M7. Some commercial products based on the ARM platform include STM32 series [35], Arduino Nano [13], SparkFun Edge [180], etc. Meanwhile, some

existing work targets the open PULP platform [160]. GAP8 processor is a commercial implementation of the PULP platform, which is designed based on RISC-V architecture and distributed by GreenWaves [80].

Software. There are several deep learning frameworks for MCUs. ARM provides CMSIS-NN that consists of efficient implementation of many operations [116]. TensorFlow Lite for Microcontrollers (TFLM) is a general framework developed by Google [54]. It is an interpreter-based framework that contains its own implementation of operations but also is able to integrate with other libraries such as CMSIS-NN. Other than that, there are several frameworks including TinyEngine [125], MicroTVM [44], CMix-NN [34], etc. STMicroelectronics also provides an extension tool named X-CUBE-AI to deploy deep learning models on its STM32 series MCUs [185].

There are also frameworks particularly for PULP-based platforms. GAP8 has its own operation library named PULP-NN [72]. Other frameworks aiming to deploy deep learning models on GAP8 processors include FANN-on-MCU [193] and Dory [31].

Deep Learning Applications. Some existing work provides efficient design for general deep learning applications on MCUs. For example, MCUNet [125] uses the neural architecture search (NAS) technique to search for suitable model structures for running on MCUs. Its successor work uses a patch-based method to reduce the peak memory footprint when doing inference [124]. SwapNN [144] dynamically swaps neural network data chunks between SRAM and external SD card, although it does not consider the QSPI or SDRAM. Some existing work aims at some specific applications running on MCUs. Those applications includes short commands recognition [82], face recognition [204], environmental sound classification [64], etc. Besides inference, there is also existing work conducting neural network training on MCUs [126, 154].

5.6 Chapter Summary

Deep Neural Network (DNN) applications are widely used in daily life. The platforms running those applications are diverse, from mobile phones to cloud clusters. Meanwhile, large numbers of tiny edge devices such as microcontrollers (MCUs) are integrated into daily devices and have a promising growing trend. However, deploying and employing DNN applications on tiny devices is still challenging because of the limited computing resources those devices have. We study some possible optimization techniques to bridge the gap between the applications and the devices. For small enough DNN models, we aim to improve the inference latency on MCUs by lightweight quantization and loop unrolling. For large models that cannot fit in the device directly, we test the off-chip memory performance to take full advantage of the resources. Preliminary results show latency improvement to some extent. The future goal is to provide optimal latency performance for various neural networks.

Chapter 6

Conclusion and Future Research Directions

6.1 Summary of Dissertation Contributions

Graph processing is an important building block of many modern applications. On the one hand, those graph-based applications have difficulty achieving high performance from parallelism techniques due to their irregular computation pattern and unpredictable control flow, and some of the applications in the database and machine learning fields even use sophisticated algorithms that are inherently sequential and thus hard to be parallelized. On the other hand, modern multi-core architectures provide massive computing resources through their capability of parallelism, including thread-level parallelism, data-level parallelism, and advanced memory hierarchy. This dissertation focuses on how to improve the performance of graph-based applications via algorithm and system co-design. It studies the characteristics of graph-based applications and provides deliberate optimizations to improve the data locality, workload balance, and synchronization overhead. Furthermore, it proposes parallel solutions to break the inherent dependencies of particular applications.

First, for typical graph algorithms with available parallel solutions, this dissertation presents the insight that the whole graph processing system stack, including data rep-

resentation, the execution model, and job scheduling, should match the features of the hardware. It presents GraphPhi as a new approach to graph processing on emerging Intel Xeon Phi-like architectures. Specifically, GraphPhi consists of 1) an optimized hierarchically blocked graph representation to enhance the data locality for both edges and vertices within and among threads, 2) a hybrid vertex-centric and edge-centric execution to efficiently find and process active edges, and 3) a uniform MIMD-SIMD scheduler integrated with a lock-free update support to achieve both good thread-level load balance and SIMD-level utilization. Besides, our efficient MIMD-SIMD execution is capable of hiding memory latency by increasing the number of concurrent memory access requests, thus benefiting more from the latest High-Bandwidth Memory technique. We evaluate our GraphPhi on six graph processing applications. Compared to two state-of-the-art shared-memory graph processing frameworks, GraphPhi results in speedups up to 4× and 35×, respectively.

Second, for particular graph applications without nontrivial parallel solutions, this dissertation studies a state-of-the-art 2-hop labeling approach named Pruned Landmark Labeling (PLL), which is used to solve the shortest path distance problem for large graphs. PLL imposes a control-flow dependency among each graph traversal iteration, which reduces its algorithmic complexity but becomes its major obstacle to enabling parallelism. This dissertation re-designs PLL from a parallel processing perspective and proposes the algorithm named BVC-PLL, which breaks PLL's inherent dependencies and parallelizes it in a scalable way. This dissertation also demonstrates how the BVC-PLL algorithm can be extended to handle directed graphs and weighted graphs and how the version for weighted graphs can benefit from SIMD parallelization. In the results, the sequential BVC-PLL can run above 2× faster than the original PLL (both using one single thread). And the parallel BVC-PLL shows an average speedup of 6.6× over sequential BVC-PLL on a 20-core shared memory machine, and up to 11.8× on a 16-node cluster.

Third, for particular applications using graph-based solutions, this dissertation researches the sequential search algorithm for the graph-based indexing methods used for the approximate nearest neighbor search (ANNS) problem. The sequential search algo-

rithm employs best-first traversal along the underlying graph indices to search nearest neighbors for given queries. This dissertation proposes *Speed-ANN*, a parallel similarity search algorithm that adopts hidden intra-query parallelism that uses multi-cores to accelerate the search speed while achieving high accuracy. It allows multiple walkers to simultaneously advance the search frontier and relax the strict global order. The results show that *Speed-ANN* reduces query latency by 13× and 17.8× on average of million-scale datasets than two state-of-the-art graph-based solutions at 0.999 recall target, respectively. It also offers up to 16.0× speedup on two billion-scale datasets.

Fourth, this dissertation explores the optimization opportunities for deep neural network applications on MCUs. This dissertation exploits the lightweight quantization and fine-tuned loop unrolling techniques to improve the inference latency, which provides totally 1.30× speedup over baseline. This dissertation also tested the impact of off-chip memory by putting the executable code and intermediate variables in the QSPI and SDRAM, respectively. Compared with SDRAM, QSPI results in a much larger decline in inference latency.

6.2 General Strategies of Optimization

Based on the observation from this dissertation, there are some general strategies for optimizing the performance of graph-based applications.

6.2.1 Algorithm Side Strategies.

First, it is important to design a parallel algorithm to expose potential parallelism opportunities. As mentioned before, some applications only have sequential algorithms. Those algorithms are elaborated to reduce theoretical time complexity but also have an inherent dependency, which is difficult to be parallelized. Thus, designing a parallel algorithm for the applications can bring new opportunities for parallelism, allowing them to take advantage of underlying parallel architectures.

Second, the designed parallel algorithm should consider providing coarse-grained parallelism rather than fine-grained parallelism. Compared with fine-grained parallelism, coarse-grained parallelism is able to provide enough workloads for multiple processors and thus reduce the number of synchronization and corresponding overhead.

6.2.2 System Side Strategies.

First, data grouping is a general strategy not only for CPU but also GPU [109]. Data grouping stores relevant data in contiguous memory, which is able to improve temporal data locality and also spacial data locality. Because graph-based applications usually have irregular memory access patterns, the improvement in data locality can make a big difference in the final performance. Moreover, data grouping brings new dimensions for organizing the data, which also provides opportunities for improving the load balance among processors. In order to group relevant data together, some new storage formats might be necessary. The new format should take into account the access and computation pattern of the given application.

Second, when grouping data together, it is also important to consider the influence of hierarchies. The design of the data hierarchy should match the features of the hardware. For example, for CPU architectures, a hierarchy can achieve good performance when its sizes match the CPU's memory hierarchy; For GPU architectures, it is important to take into account the size of warps and thread blocks to improve GPU utilization [173].

Third, some applications need to take the synchronization overhead into account, especially for ordered graph algorithms [212]. Designing a good synchronization mechanism can reduce synchronization overhead and hence improve the final performance.

6.2.3 Comparison with GPUs

Although this dissertation focuses on optimization performance on CPUs, some challenges and optimization ideas are also shared with GPUs.

First, data layout and format for memory access regularity. GPU has a global memory shared by all GPU processors. When accessing the global memory, the threads of a warp use one single coalesced memory access if they are accessing contiguous memory addresses [109]. However, graph-based applications usually have irregular memory accesses, which require multiple memory accesses by a GPU to fetch all data scattered across different memory locations. This irregularity impedes a GPU from achieving its peak memory bandwidth. Therefore, designing a proper graph data layout and representation is also important for GPUs. To overcome the irregular memory access issue, some work, such as CuSha [109] and GTS [110], propose particular data formats to reduce non-coalesced memory accesses.

Second, workload mapping for load balance. The complex graphs usually have skewed degree distribution that a small part of vertices may have a very large degree than others. As GPUs run in a Single Instruction Multiple Threads (SIMT) model, threads in a warp are issued the same instruction. An uneven workload distribution among the threads within a kernel call may harm the performance significantly [70]. Some work, such as MapGraph [70] and GunRock [194], proposes workload mapping strategies to overcome the load imbalance issue.

6.3 Future Research Directions

A possible short-term future research is to find some computational kernels that have a graph-based pattern and also have a large impact on the performance of the whole system. With different features, these kernels might need customized optimization techniques.

For general graph-based applications, one future work is to integrate existing optimization into a graph-oriented compiler. The objective of this compiler is to generate optimized code for different target applications and platforms with various input data. First, the compiler can automatically apply those optimization techniques based on the application requirement. Second, the compiler can simplify the deployment procedure for different

target platforms. Third, the compiler has a pipeline with an intermediate representation that can expose potential optimization opportunities.

Another line of future research is to continue the exploration of deep neural network applications on tiny devices such as MCUs. The possible goal is to design a framework that can provide an end-to-end solution for model deployment. Moreover, accuracy is also an important metric that has not been discussed in this dissertation. How to guarantee accuracy while also improving the inference latency is still an open research question.

Bibliography

[1] Friendster network dataset – KONECT, April 2017.

[2] Graph500: Benchmark specification, June 2017.

[3] Livejournal network dataset – KONECT, April 2017.

[4] Pokec network dataset – KONECT, April 2017.

[5] Twitter (www) network dataset – KONECT, April 2017.

[6] Faiss library, 2021. https://github.com/facebookresearch/faissm.

[7] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Experimental Algorithms*, Panos M. Pardalos and Steffen Rebennack, editors, pages 230–241, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] Takuya Akiba. Pruned labeling algorithms: Fast, exact, dynamic, simple and general indexing scheme for shortest-path queries. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, pages 1339–1340, New York, NY, USA, 2014. Association for Computing Machinery.

[9] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast Exact Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 349–360. ACM, 2013.

[10] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *2006 47th annual IEEE symposium on foundations of computer science (FOCS'06)*, pages 459–468. IEEE, 2006.

[11] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal lsh for angular distance. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS)*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, volume 28 of *NIPS'15*, pages 1225–1233, Cambridge, MA, USA, 2015. MIT Press.

[12] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. *Proceedings of the VLDB Endowment*, 9(4):288–299, 2015.

[13] Arduino. Arduino nano. `https://store.arduino.cc/products/arduino-nano`, 2022. Accessed: 2022-10-01.

[14] Akhil Arora, Sakshi Sinha, Piyush Kumar, and Arnab Bhattacharya. Hd-index: Pushing the scalability-accuracy boundary for approximate knn search in high-dimensional spaces. *Proceedings of the VLDB Endowment*, 11(8):906–919, 2018.

[15] Sunil Arya and David M Mount. Approximate nearest neighbor queries in fixed dimensions. In *SODA*, volume 93, pages 271–280, 1993.

[16] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In *International Conference on Similarity Search and Applications (SISAP)*, pages 34–49. Springer, 2017.

[17] Artem Babenko and Victor Lempitsky. The inverted multi-index. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(6):1247–1260, 2014.

[18] Artem Babenko and Victor Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[19] Maxim Babenko, Andrew V. Goldberg, Haim Kaplan, Ruslan Savchenko, and Mathias Weller. On the complexity of hub labeling (extended abstract). In *Mathematical Foundations of Computer Science 2015*, pages 62–74. Springer Berlin Heidelberg, 2015.

[20] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems*, 3:517–532, 2021.

[21] Dmitry Baranchuk, Dmitry Persiyanov, Anton Sinitsin, and Artem Babenko. Learning to route in similarity graphs. In *Proceedings of the 36th International Conference on Machine Learning*, Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, volume 97 of *Proceedings of Machine Learning Research*, pages 475–484, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[22] KG Renga Bashyam and Sathish Vadhiyar. Fast scalable approximate nearest neighbor search for high-dimensional data. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 294–302. IEEE, 2020.

[23] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, 1990.

[24] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature Biotechnology*, 33(6):623–630, 2015.

[25] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. SlimSell: A Vectorizable Graph Representation for Breadth-First Search. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 32–41. IEEE, 2017.

[26] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 93–104. ACM, 2017.

[27] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

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

[29] André R Brodtkorb, Trond R Hagen, and Martin L Sætra. Graphics processing unit (gpu) programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.

[30] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. {TAO}:{Facebook's} distributed data store for the social

graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[31] Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. *IEEE Transactions on Computers*, 70(8):1253–1268, 2021.

[32] Neil Butcher, Stephen L Olivier, Jonathan Berry, Simon D Hammond, and Peter M Kogge. Optimizing for knl usage modes when data doesn't fit in mcdram. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP)*, pages 1–10, 2018.

[33] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning Graph Representations with Global Structural Information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 891–900. ACM, 2015.

[34] Alessandro Capotondi, Manuele Rusci, Marco Fariselli, and Luca Benini. Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(5):871–875, 2020.

[35] James Carbone. Stm32 32-bit arm cortex mcus. `https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html`, 2022. Accessed: 2022-10-01.

[36] João Manuel Paiva Cardoso, José Gabriel de Figueired Coutinho, and Pedro C Diniz. *Embedded computing for high performance: Efficient mapping of computations using customization, code transformations and compilation.* Morgan Kaufmann, 2017.

[37] Tim Carnes, Chandrashekhar Nagarajan, Stefan M. Wild, and Anke van Zuylen. Maximizing influence in a competitive social network: A follower's perspective. In *Proceedings of the Ninth International Conference on Electronic Commerce*, ICEC '07, pages 351–360, New York, NY, USA, 2007. Association for Computing Machinery.

[38] Umit V Catalyurek, Erik G Boman, Karen D Devine, Doruk Bozdag, Robert Heaphy, and Lee Ann Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *2007 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–11. IEEE, 2007.

[39] Gregory J Chaitin. Register allocation & spilling via graph coloring. *ACM Sigplan Notices*, 17(6):98–101, 1982.

[40] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. Pannotia: Understanding Irregular GPGPU Graph Applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 185–195. IEEE, 2013.

[41] Linchuan Chen, Xin Huo, Bin Ren, Surabhi Jain, and Gagan Agrawal. Efficient and simplified parallel graph processing over cpu and mic. In *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 819–828. IEEE, 2015.

[42] Linchuan Chen, Peng Jiang, and Gagan Agrawal. Exploiting Recent SIMD Architectural Advances for Irregular Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 47–58. ACM, 2016.

[43] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. *SPTAG: A library for fast approximate nearest neighbor search*, 2018.

[44] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[45] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and S Yu Philip. Fast computation of reachability labeling for large graphs. In *International Conference on Extending Database Technology*, pages 961–979. Springer, 2006.

[46] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S Yu. Fast computing reachability labelings for large graphs with high compression rate. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 193–204, 2008.

[47] Jiefeng Cheng, Jeffrey Xu Yu, and Philip S. Yu. Graph pattern matching: A join/semijoin approach. *IEEE Trans. Knowl. Data Eng.*, 23(7):1006–1021, 2011.

[48] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the 13th annual ACM-SIAM Symposium on Discrete algorithms*, pages 937–946, 2002.

[49] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems, Boston, MA, USA, September 15-19, 2016*, Shilad Sen, Werner Geyer, Jill Freyne, and Pablo Castells, editors, pages 191–198. ACM, 2016.

[50] Leonardo Dagum and Ramesh Menon. OpenMP: an Industry Standard API for Shared-memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[51] Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram.

Google news personalization: Scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, pages 271–280, 2007.

[52] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 253–262, New York, NY, USA, 2004. Association for Computing Machinery.

[53] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A Communication-Optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 752–768. ACM, 2018.

[54] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems*, 3:800–811, 2021.

[55] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. Work-efficient Parallel GPU Methods for Single-Source Shortest Paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 349–359. IEEE, 2014.

[56] DW Dearholt, N Gonzales, and G Kurup. Monotonic search networks for computer vision databases. In *Twenty-Second Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 548–553. IEEE, 1988.

[57] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. Robust exact distance queries on massive networks. *Microsoft Research, USA, Tech. Rep*, 2, 2014.

[58] Shiyuan Deng, Xiao Yan, KW Ng Kelvin, Chenyu Jiang, and James Cheng. Pyramid: A general framework for distributed similarity search on large-scale datasets. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 1066–1071. IEEE, 2019.

[59] Q. Dong, K. Lakhotia, H. Zeng, R. Karman, V. Prasanna, and G. Seetharaman. A fast and efficient parallel algorithm for pruned landmark labeling. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, Sep. 2018.

[60] Qing Dong, Kartik Lakhotia, Hanqing Zeng, Rajgopal Karman, Viktor Prasanna, and Guna Seetharaman. A fast and efficient parallel algorithm for pruned landmark labeling. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[61] Marat Dukhan. The indirect convolution algorithm. *arXiv preprint arXiv:1907.02129*, 2019.

[62] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. Return of the lernaean hydra: Experimental evaluation of data series approximate similarity search. *Proceedings of the VLDB Endowment*, 13(3):403–420, 2019.

[63] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. Pixie: A system for recommending 3+ billion items to 200+ million users in real-time. In *Proceedings of the 2018 World Wide Web Conference (WWW*, pages 1775–1784, 2018.

[64] David Elliott, Carlos E Otero, Steven Wyatt, and Evan Martino. Tiny transformers for environmental sound classification at the edge. *arXiv preprint arXiv:2103.12157*, 2021.

[65] Damir Ferizovic. *Parallel Pruned Landmark Labeling for Shortest Path Queries on Unit-Weight Networks*. PhD thesis, National Research Center, 2015.

[66] Adam Fidel, Nancy M Amato, Lawrence Rauchwerger, et al. Kla: A new algorithmic paradigm for parallel graph computations. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 27–38. IEEE, 2014.

[67] Jesun Sahariar Firoz, Marcin Zalewski, Thejaka Kanewala, and Andrew Lumsdaine. Synchronization-avoiding graph algorithms. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 52–61. IEEE, 2018.

[68] Cong Fu and Deng Cai. Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph. *arXiv preprint arXiv:1609.07228*, 2016.

[69] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment (VLDB)*, 12(5):461–474, January 2019.

[70] Zhisong Fu, Michael Personick, and Bryan Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[71] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. Xbfs: exploring runtime optimizations for breadth-first search on gpus. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 121–131. ACM, 2019.

[72] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and

Luca Benini. Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors. *Philosophical Transactions of the Royal Society A*, 378(2164):20190155, 2020.

[73] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2946–2953, 2013.

[74] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th international conference on Experimental algorithms*, pages 319–333, 2008.

[75] Alan George, John R Gilbert, and Joseph WH Liu. *Graph theory and sparse matrix computation*, volume 56. Springer Science & Business Media, 2012.

[76] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques, PACT*, pages 345–354. ACM, 2012.

[77] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. *Proceedings of the VLDB Conference (VLDB)*, 99(6):518–529, 1999.

[78] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, volume 12, page 2, 2012.

[79] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural

graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, October 2012. USENIX Association.

[80] GreenWaves. Gap8 next generation processor for smart sensors. `https://greenwaves-technologies.com/gap8_mcu_ai/`, 2022. Accessed: 2022-10-01.

[81] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-passing Interface*, volume 1. MIT Press, 1999.

[82] Ivana Guarneri, Giuseppe Messina, Arcangelo Bruna, and Davide Giacalone. A deep learning short commands recognition for mcu in robotics applications. In *2021 7th International Conference on Automation, Robotics and Applications (ICARA)*, pages 43–47. IEEE, 2021.

[83] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing*, page 14, May 2012.

[84] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 1312—-1317. AAAI Press, 2011.

[85] Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.

[86] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT),*, pages 233–245. IEEE, 2017.

[87] Ben Harwood and Tom Drummond. Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5713–5722, June 2016.

[88] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1223–1231, 2013.

[89] Johannes Hoffart, Stephan Seufert, Dat Ba Nguyen, Martin Theobald, and Gerhard Weikum. Kore: Keyphrase overlap relatedness for entity disambiguation. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM)*, pages 545–554, 2012.

[90] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P Sadayappan. Multigraph: Efficient graph processing on gpus. In *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 27–40. IEEE, 2017.

[91] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient Parallel Graph Exploration on Multi-core CPU and GPU. In *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 78–88. IEEE, 2011.

[92] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 9(1):1–12, 2015.

[93] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. Association for Computing Machinery.

[94] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, volume 32, pages 13771–13781. Curran Associates, Inc., 2019.

[95] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[96] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *European conference on computer vision*, pages 304–317. Springer, 2008.

[97] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[98] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861–864. IEEE, 2011.

[99] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A distributed multi-gpu system for fast graph processing. *Proceedings of the VLDB Endowment*, 11(3):297–310, 2017.

[100] Peng Jiang, Linchuan Chen, and Gagan Agrawal. Reusing Data Reorganization for Efficient SIMD Parallelization of Adaptive Irregular Applications. In *Proceedings of the 2016 International Conference on Supercomputing*, page 16. ACM, 2016.

[101] R. Jin, N. Ruan, Y. Xiang, and V. E. Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *SIGMOD*, 2012.

[102] Ruoming Jin, Zhen Peng, Wendell Wu, Feodor Dragan, Gagan Agrawal, and Bin Ren. Pruned landmark labeling meets vertex centric computation: A surprisingly happy marriage! *arXiv preprint arXiv:1906.12018*, 2019.

[103] Ruoming Jin, Ning Ruan, Bo You, and Haixun Wang. Hub-accelerator: Fast and exact shortest path computation in large social networks. *CoRR*, abs/1305.0507, 2013.

[104] Ruoming Jin and Guan Wang. Simple, fast, and scalable reachability oracle. *Proceedings of the VLDB Endowment*, 6(14):1978–1989, 2013.

[105] Zhongming Jin, Debing Zhang, Yao Hu, Shiding Lin, Deng Cai, and Xiaofei He. Fast and accurate hashing via iterative nearest neighbors expansion. *IEEE Transactions on Cybernetics*, 44(11):2167–2177, 2014.

[106] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

[107] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy G. Mattson, and José E. Moreira. Mathematical foundations of the graphblas. In *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*, pages 1–9, 2016.

[108] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*, volume 22. SIAM, 2011.

[109] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: Vertex-centric Graph Processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 239–252. ACM, 2014.

[110] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. Gts: A fast and scalable graph processing method based on streaming topology to gpus. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 447–461, New York, NY, USA, 2016. Association for Computing Machinery.

[111] Seongyun Ko and Wook-Shin Han. TurboGraph++: A Scalable and Fast Graph Analytics System. In *Proceedings of the 2018 International Conference on Management of Data*, pages 395–410. ACM, 2018.

[112] Jim Kukunas. *Power and performance: Software analysis and optimization*. Morgan Kaufmann, 2015.

[113] Brian Kulis and Kristen Grauman. Kernelized locality-sensitive hashing for scalable image search. In *2009 IEEE 12th International Conference on Computer Vision (ICCV)*, pages 2130–2137. IEEE, 2009.

[114] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, October 2012. USENIX Association.

[115] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2012.

[116] Liangzhen Lai, Naveen Suda, and Vikas Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601*, 2018.

[117] Kartik Lakhotia, Rajgopal Kannan, Qing Dong, and Viktor Prasanna. Planting trees for scalable and efficient canonical hub labeling. *Proc. VLDB Endow.*, 13(4):492–505, December 2019.

[118] Quoc V. Le and Tomás Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 1188–1196. JMLR.org, 2014.

[119] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, Friedhelm Meyer auf der Heide and Cynthia A. Phillips, editors, pages 303–314. ACM, 2010.

[120] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. Improving approximate nearest neighbor search through learned adaptive early termination. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 2539–2554, New York, NY, USA, 2020. Association for Computing Machinery.

[121] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data – experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1475–1488, 2020.

[122] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. Scaling distance labeling on small-world networks. In *Proceedings of the 2019*

International Conference on Management of Data, SIGMOD '19, pages 1060–1077, New York, NY, USA, 2019. Association for Computing Machinery.

[123] Ye Li, Man Lung Yiu, Ngai Meng Kou, et al. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment*, 11(4):445–457, 2017.

[124] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. Mcunetv2: Memory-efficient patch-based inference for tiny deep learning. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, volume 34, pages 2346–2358. Curran Associates, Inc., 2021.

[125] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, and Song Han. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems*, 33:11711–11722, 2020.

[126] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. On-device training under 256kb memory. *arXiv preprint arXiv:2206.15472*, 2022.

[127] Peng-Cheng Lin and Wan-Lei Zhao. Graph based nearest neighbor search: Promises and failures. *arXiv preprint arXiv:1904.02077*, 2019.

[128] Changxi Liu, Biwei Xie, Xin Liu, Wei Xue, Hailong Yang, and Xu Liu. Towards Efficient SpMV on Sunway Many-core Architectures. In *Proceedings of the 32th ACM on International Conference on Supercomputing, ICS*, volume 18, pages 12–15, 2018.

[129] Hang Liu and H Howie Huang. Enterprise: Breadth-First Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 68. ACM, 2015.

[130] Hang Liu and H Howie Huang. Simd-x: Programming and processing of graph algorithms on gpus. *arXiv preprint arXiv:1812.04070*, 2018.

[131] Hang Liu, H Howie Huang, and Yang Hu. iBFS: Concurrent Breadth-First Search on GPUs. In *Proceedings of the 2016 International Conference on Management of Data*, pages 403–416. ACM, 2016.

[132] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep 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)*, pages 273–282, 2013.

[133] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A New Framework for Parallel Machine Learning. *arXiv preprint arXiv:1408.2041*, 2014.

[134] Qin Lv, Moses Charikar, and Kai Li. Image similarity search with compact data structures. In *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management (CIKM)*, pages 208–217, 2004.

[135] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 527–543. ACM, 2017.

[136] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[137] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel:

a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, 2010.

[138] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[139] Yury A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020.

[140] Adam McLaughlin and David A Bader. Scalable and High Performance Betweenness Centrality on the GPU. In *Proceedings of the International Conference for High performance computing, networking, storage and analysis*, pages 572–583. IEEE Press, 2014.

[141] Clara Meister, Tim Vieira, and Ryan Cotterell. Best-first beam search. *Transactions of the Association for Computational Linguistics*, 8:795–809, 2020.

[142] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. A pattern based algorithmic autotuner for graph processing on gpus. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 201–213, 2019.

[143] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU Graph Traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

[144] Hongyu Miao and Felix Xiaozhu Lin. Enabling large neural networks on tiny microcontrollers with swapping. *arXiv preprint arXiv:2101.08744*, 2021.

[145] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *1st International Conference*

on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings, Yoshua Bengio and Yann LeCun, editors, 2013.

[146] Marius Muja and David G Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *International Conference on Computer Vision Theory and Applications (VISAPP)*, 2(331–340):2, 2009.

[147] Maxim Naumov, Alysson Vrielink, and Michael Garland. Parallel depth-first search for directed acyclic graphs. In *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–8, 2017.

[148] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[149] Priyanka Nigam, Yiwei Song, Vijai Mohan, Vihan Lakshman, Weitian Allen Ding, Ankit Shingavi, Choon Hui Teo, Hao Gu, and Bing Yin. Semantic product search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis, editors, pages 2876–2885. ACM, 2019.

[150] Pierre-Emmanuel Novac, Ghouthi Boukli Hacene, Alain Pegatoquet, Benoît Miramond, and Vincent Gripon. Quantization and deployment of deep neural networks on microcontrollers. *Sensors*, 21(9):2984, 2021.

[151] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 709–724, 2018.

[152] Sreepathi Pai and Keshav Pingali. A Compiler for Throughput Optimization

of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19. ACM, 2016.

[153] Jongsoo Park, Ganesh Bikshandi, Karthikeyan Vaidyanathan, Ping Tak Peter Tang, Pradeep Dubey, and Daehyun Kim. Tera-Scale 1D FFT With Low-Communication Algorithm and Intel® Xeon Phi™ Coprocessors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 34. ACM, 2013.

[154] Shishir G Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph Gonzalez. Poet: Training neural networks on tiny devices with integrated rematerialization and paging. In *International Conference on Machine Learning*, pages 17573–17583. PMLR, 2022.

[155] Scott Beamer Krste Asanovic David Patterson. Direction-Optimizing Breadth-First Search. *SC12, November*, pages 10–16, 2012.

[156] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. GraphPhi: Efficient Parallel Graph Processing on Emerging Throughput-oriented Architectures. In *2018 International Conference on Parallel Architecture and Compilation (PACT)*. ACM, 2018.

[157] James Philbin, Ondrej Chum, Michael Isard, Josef Sivic, and Andrew Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *2007 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8. IEEE, 2007.

[158] Jon J Pimentel, Brent Bohnenstiehl, and Bevan M Baas. Hybrid hardware/software floating-point implementations for optimized area and throughput tradeoffs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(1):100–113, 2016.

[159] Liudmila Prokhorenkova and Aleksandr Shekhovtsov. Graph-based nearest neighbor search: From practice to theory. In *Proceedings of the 37th International Conference on Machine Learning*, Hal Daumé III and Aarti Singh, editors, volume 119 of *Proceedings of Machine Learning Research*, pages 7803–7813, Virtual, 13–18 Jul 2020. PMLR.

[160] PULP. Pulp platform. `https://pulp-platform.org/`, 2022. Accessed: 2022-10-01.

[161] Qiongwen Xu, Xu Zhang, Jin Zhao, Xin Wang, and T. Wolf. Fast shortest-path queries on large-scale graphs. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, 2016.

[162] Kun Qiu, Yuanyang Zhu, Jing Yuan, Jin Zhao, Xin Wang, and Tilman Wolf. ParaPLL: Fast Parallel Shortest-path Distance Query on Large-scale Weighted Graphs. In *Proceedings of the 47th International Conference on Parallel Processing*, page 2. ACM, 2018.

[163] Abdul Quamar, Amol Deshpande, and Jimmy Lin. NScale: Neighborhood-Centric Large-Scale Graph Analytics in the Cloud. *The VLDB Journal–The International Journal on Very Large Data Bases*, 25(2):125–150, 2016.

[164] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* " O'Reilly Media, Inc.", 2007.

[165] Neil Robertson and Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322, 1986.

[166] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[167] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[168] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2, August 2009.

[169] Jagan Sankaranarayanan, Houman Alborzi, and Hanan Samet. Distance join queries on spatial networks. In *Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*, GIS '06, 2006.

[170] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Hopi: An efficient connection index for complex xml document collections. In *Advances in Database Technology - EDBT 2004*, Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, editors, pages 237–255, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[171] Robert Searles, Stephen Herbein, and Sunita Chandrasekaran. A portable, high-level graph analytics framework targeting distributed, heterogeneous systems. In *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*, pages 79–88. IEEE, 2016.

[172] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. GraphReduce: Processing Large-scale Graphs on Accelerator-based Systems. In *2015 SC-International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, pages 1–12. IEEE, 2015.

[173] Hyunseok Seo, Jinwook Kim, and Min-Soo Kim. Gstream: A graph streaming processing method for large-scale graphs on gpus. In *Proceedings of the 20th*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, page 253–254, New York, NY, USA, 2015. Association for Computing Machinery.

[174] P. Shiralkar, A. Flammini, F. Menczer, and G. L. Ciampaglia. Finding streams in knowledge graphs to support fact checking. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 859–864, Nov 2017.

[175] Julian Shun and Guy E Blelloch. Ligra: a Lightweight Graph Processing Framework for Shared Memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[176] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8. IEEE, 2008.

[177] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[178] Avinash Sodani. Knights landing (knl): 2nd generation intel® xeon phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24. IEEE, 2015.

[179] Jyothish Soman, Kothapalli Kishore, and PJ Narayanan. A Fast GPU Algorithm for Graph Connectivity. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[180] SparkFun. Sparkfun edge development board - apollo3 blue. `https://www.sparkfun.com/products/15170`, 2022. Accessed: 2022-10-01.

[181] Milan Stanic, Oscar Palomar, Ivan Ratkovic, Milovan Duric, Osman Unsal, Adrian Cristal, and Mateo Valero. Evaluation of vectorization po-

tential of graph500 on intel's xeon phi. In *2014 International Conference on High Performance Computing & Simulation (HPCS)*, pages 47–54. IEEE, 2014.

[182] STMicroelectronics. Tight supply and less price erosion is expected for 32-bit mcus. `https://electronics-sourcing.com/2021/03/08/tight-supply-and-less-price-erosion-is-expected-for-32-bit-mcus/`, 2021. Accessed: 2022-10-01.

[183] STMicroelectronics. 32f469idiscovery, discovery kit with stm32f469ni mcu. `https://www.st.com/en/evaluation-tools/32f469idiscovery.html`, 2022. Accessed: 2022-10-01.

[184] STMicroelectronics. Stm32f469ni, high-performance advanced line, arm cortex-m4 core with dsp and fpu, 2 mbytes of flash memory, 180 mhz cpu, art accelerator, chrom-art accelerator, fmc with sdram, dual qspi, tft, mipi-dsi. `https://www.st.com/en/microcontrollers-microprocessors/stm32f469ni.html`, 2022. Accessed: 2022-10-01.

[185] STMicroelectronics. X-cube-ai, ai expansion pack for stm32cubemx. `https://www.st.com/en/embedded-software/x-cube-ai.html`, 2022. Accessed: 2022-10-01.

[186] Danny Sullivan. Faq: All about the google rankbrain algorithm. `https://searchengineland.com/faq-all-about-the-new-google-rankbrain-algorithm-234440`, 2018.

[187] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. Graphgrind: Addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*, ICS '17, New York, NY, USA, 2017. Association for Computing Machinery.

[188] Jiawen Sun, Hans Vandierendonck, and Dimitrios S Nikolopoulos. Graph-

grind: addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*, pages 1–10, 2017.

[189] Narayanan Sundaram, Nadathur Rajagopalan Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *arXiv preprint arXiv:1503.07241*, 2015.

[190] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[191] Hans Vandierendonck. Graptor: Efficient pull and push style vectorized graph processing. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[192] Runhui Wang and Dong Deng. Deltapq: Lossless product quantization code compression for high dimensional similarity search. *Proceedings of the VLDB Endowment*, 13(13):3603–3616, 2020.

[193] Xiaying Wang, Michele Magno, Lukas Cavigelli, and Luca Benini. Fann-on-mcu: An open-source toolkit for energy-efficient neural network inference at the edge of the internet of things. *IEEE Internet of Things Journal*, 7(5):4403–4417, 2020.

[194] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 11. ACM, 2016.

[195] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. Analyticdb-v: A hybrid analytical engine towards

query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment*, 13(12):3152–3165, 2020.

[196] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828. ACM, 2016.

[197] Mathias Weller. Optimal hub labeling is np-complete. *CoRR*, abs/1407.8373, 2014.

[198] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-Coalesced Memory Accesses on GPU. In *ACM SIGPLAN Notices*, volume 48, pages 57–68. ACM, 2013.

[199] Xiang Wu, Ruiqi Guo, Ananda Theertha Suresh, Sanjiv Kumar, Daniel N Holtmann-Rice, David Simcha, and Felix Yu. Multiscale quantization for fast similarity search. In *Advances in Neural Information Processing Systems (NIPS)*, pages 5745–5755, 2017.

[200] Yubao Wu, Ruoming Jin, and Xiang Zhang. Fast and unified local search for random walk based k-nearest-neighbor query in large graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of Data*, pages 1139–1150. ACM, 2014.

[201] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. CVR: Efficient Vectorization of SpMV on X86 Processors. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 149–162. ACM, 2018.

[202] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric

framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[203] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 974–983, 2018.

[204] Maxim Zemlyanikin, Alexander Smorkalov, Tatiana Khanova, Anna Petrovicheva, and Grigory Serebryakov. 512kib ram is enough! live camera face recognition dnn on mcu. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pages 0–0, 2019.

[205] Eddy Z Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly Elimination of Dynamic Irregularities for GPU Computing. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 369–380. ACM, 2011.

[206] Jialiang Zhang, Soroosh Khoram, and Jing Li. Efficient large-scale approximate nearest neighbor search on opencl FPGA. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 4924–4932. IEEE Computer Society, 2018.

[207] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware Graph-Structured Analytics. *ACM SIGPLAN Notices*, 50(8):183–193, 2015.

[208] Minjia Zhang and Yuxiong He. Grip: Multi-store capacity-optimized high-performance nearest neighbor search for vector search engine. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, pages 1673–1682, New York, NY, USA, 2019. Association for Computing Machinery.

[209] Qi Zhang, Jihua Kang, Yeyun Gong, Huan Chen, Yaqian Zhou, and Xu-

anjing Huang. Map search via a factor graph model. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management (CIKM)*, pages 69–78, 2013.

[210] Yanhao Zhang, Pan Pan, Yun Zheng, Kang Zhao, Yingya Zhang, Xiaofeng Ren, and Rong Jin. Visual search at alibaba. *CoRR*, abs/2102.04674, 2021.

[211] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. Optimizing ordered graph algorithms with graphit. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, pages 158–170, New York, NY, USA, 2020. Association for Computing Machinery.

[212] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. Optimizing ordered graph algorithms with graphit. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 158–170, New York, NY, USA, 2020. Association for Computing Machinery.

[213] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance dsl for graph analytics. *arXiv preprint arXiv:1805.00923*, 2018.

[214] Jianlong Zhong and Bingsheng He. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014.

[215] Wenyong Zhong, Jianhua Sun, Hao Chen, Jun Xiao, Zhiwen Chen, Chang Cheng, and Xuanhua Shi. Optimizing graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):1149–1162, 2016.

[216] TX Zhou and Huston Collins. Why tinyml is a giant opportunity. `https://venturebeat.com/ai/why-tinyml-is-a-giant-opportunity/`, 2020. Accessed: 2022-10-01.

[217] Lei Zou, Lei Chen, and M. Tamer Özsu. Distance-join: pattern match query in a large graph database. *Proc. VLDB Endow.*, 2(1):886–897, August 2009.

VITA

Zhen Peng

Zhen Peng is a Ph.D. Candidate in the Department of Computer Science at William & Mary under the supervision of Professor Bin Ren. Zhen's research interests lie in parallel computing with an emphasis on the design and optimization of parallel solutions for graph-based applications. His Ph.D. research has been published in PACT 2018, ICS 2020, and PPoPP 2023. Additionally, his co-authored paper has appeared in CGO 2020. Zhen worked as an intern with Kuaishou U.S. R&D Center in the summer of 2021 and Pacific Northwest National Laboratory in the summer of 2022, respectively. Before joining William & Mary, he received his Bachelor of Engineering in Computer Science and his Master of Engineering in Computer Software and Theory at Huaqiao University, China, in 2013 and 2016, respectively.