GraphIt: Optimizing the Performance and Improving the Programmability of Graph Algorithms

by

Yunming Zhang

B.S., Rice University (2013)
M.S., Rice University (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author . . . .
Department of Electrical Engineering and Computer Science
July 31, 2020

Certified by . . . .
Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A.Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

GraphIt: Optimizing the Performance and Improving the Programmability of Graph Algorithms

by

Yunming Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on July 31, 2020, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

In recent years, large graphs with billions of vertices and trillions of edges have emerged in many domains, such as social network analytics, machine learning, physical simulations, and biology. However, optimizing the performance of graph applications is notoriously challenging due to irregular memory access patterns and load imbalance across cores. We need new performance optimizations to improve hardware utilization and require a programming system that allows domain experts to easily write high-performance graph applications.

In this thesis, I will present our work on GraphIt, a new domain-specific language that consistently achieves high performance across different algorithms, graphs, and architectures, while offering an easy-to-use high-level programming model that supports both unordered and ordered graph algorithms. GraphIt decouples algorithms from performance optimizations (schedules) for graph applications to make it easy to explore a large space of cache, non-uniform memory access, load balance, and data layout optimizations. GraphIt achieves up to 4.8x speedup over state-of-the-art graph frameworks on CPUs, while reducing the lines of code by up to one order of magnitude.

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I want to thank Professor Saman Amarasinghe for his guidance over the years and Professor Julian Shun for his significant help on the GraphIt project. Shoaib Kamil has also provided valuable feedback and countless comments on the GraphIt project. I also want to express my gratitude to Professor Matei Zaharia for his guidance in the early years of my PhD. I want to thank Professor Michael Carbin for his help and feedback with my defense and thesis.

I would like to also acknowledge contributions from other students and postdocs. Vladimir Kiriansky has been instrumental in the development of the cache optimizations and provided invaluable mentorship in my first few years at MIT. Mengjiao Yang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, and Riyadh Baghdadi have contributed significantly to GraphIt's cache and NUMA optimizations, program structure optimizations, code generation, runtime library, and benchmarking for bucketing, which are discussed in this thesis. Changwan Hong, Tugsbayasgalan Manlaibaatar, Claire Hsu, Charith Mendis, and Kenny Yang have also contributed significantly to various other features of GraphIt and its optimizations. Stephen Chou provided valuable help in building the parser for GraphIt. I want to thank the members of the COMMIT group for all the great discussions we have had over the years.

I want to thank my parents, Hong Yun and Xianglin Zhang, for raising me and providing me with the opportunity to study in the US. I am grateful to my dear wife, Mengcheng Wang, for her love and support throughout my PhD. I would not be here without the strong support of my family members.

Finally, I want to thank God for his grace and mercy. I am here today not because of my own work but largely due to his grace and plan. I want to thank the MIT Graduate Christian Fellowship, especially everyone from the SP3 small group, for their fellowship during my time at MIT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently, large graphs with billions of vertices and trillions of edges have emerged in many domains, such as social network analytics, machine learning, and biology. Extracting information from these graphs often involves running algorithms to identify important vertices, finding connections between vertices, and detecting communities of interest. Speeding up these algorithms leads to many significant impact on these domains, such as enhancing the efficiency of data analytics applications and improving the quality of web services [84, 87, 36, 35].

Modern hardware is designed for dense computations rather than sparse computations, such as graph algorithms, which have very low computation density and a large amount of irregular memory accesses and are often memory-bounded. The programmers must perform heroic software transformations and optimizations to achieve high performance for graph algorithms running on modern hardware. However, the performance bottlenecks of graph algorithms depend not only on the algorithm and the underlying hardware, but also on the size and structure of the graph [12]. As a result, choosing the right software performance optimizations can have a substantial effect on the performance of the graph algorithms. Programmers must experiment with different combinations of a large set of techniques that make tradeoffs between locality, work-efficiency, and parallelism, to develop the best implementation for a specific algorithm and type of graph. Existing graph frameworks do not provide the programmers with the flexibility to try out the different optimizations. These

Algorithm 1 PageRank

```
procedure PageRank(Graph $G$)
    parallel for v : G.vertexArray do
        for u : G.edgeArray[v] do
            G.newRank[v] += G.rank[u] / G.degree[u]
    end parallel for
```

frameworks are also difficult to use for domain experts with a limited background in parallel computing to use.

This thesis focuses on *GraphIt*, a domain-specific language (DSL) that decouples algorithm from performance optimizations for graph computations. GraphIt proposes a unified scheduling space for graph computations, allowing programmers to easily search through a large space of performance optimizations. With this novel approach, GraphIt consistently achieves high performance across different algorithms, graphs, and architectures. Moreover, GraphIt offers an easy-to-use high-level programming model for programmers without significant parallel programming experience.

1.1 Motivating Example

We use PageRank [77] in Algorithm 1 as an example here to illustrate the performance bottleneck of graph algorithms and the impact of different graphs and optimization strategies. PageRank is an important graph algorithm that ranks the vertices based on their importance in the topological structure. PageRank iteratively updates the rank of each vertex based on the rank and degree of its neighbors. The performance characteristics of PageRank can generalize to numerous graph applications. The algorithm does very little computation per byte accessed, and a large fraction of their memory requests (G.rank[u] and G.degree[u]) are irregular. Irregular accesses to a working set that does not fit in the cache make the entire cache hardware subsystem ineffective. Without effective use of the cache to mitigate the processor-DRAM gap, CPUs are stalled on high-latency irregular accesses to DRAM. We find that many graph algorithms spend 60%–80% of their cycles stalled on memory access. Figure 1-1 breaks down the ratio of cycles stalled on irregular memory accesses (blue) vs other cycles (green) for different PageRank implementations. For social networks, such as Twitter [55], the last level cache (LLC) miss rate of PageRank reaches 40%.

Different optimizations have a substantial influence on the performance of PageRank by improving the efficiency of memory access. Figure 1-1 illustrates the performance of PageRank with different software optimizations with cycles normalized to a parallel pull-based PageRank implementation. The optimizations (described in details in Chapter 3) significantly improve the locality of the graph traversals.

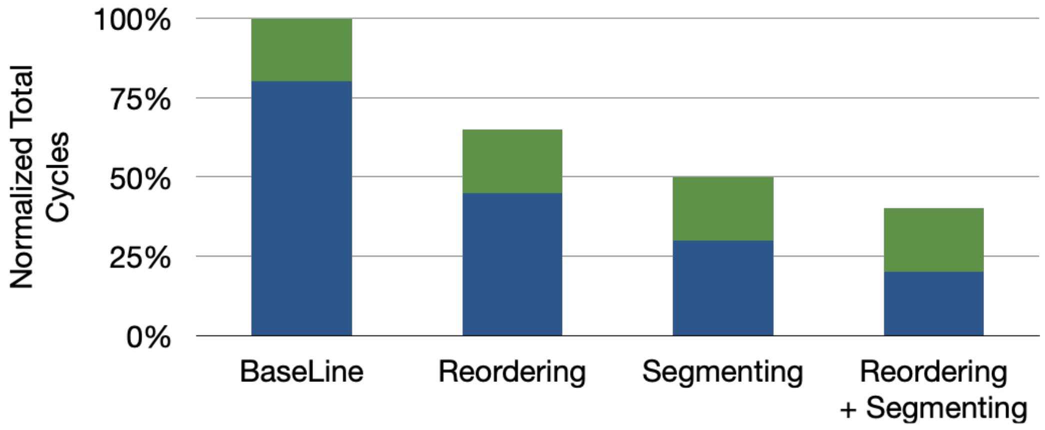


Figure 1-1: Cycles of PageRank with different software optimizations normalized over by a baseline parallel push-based implementation. The portion of cycles highlighted in blue is attributed to irregular memory accesses. The green portion represents all other cycles.

Programmers must iterate over multiple implementations of the same algorithm to identify the best combination of optimizations for a specific algorithm and input data. Existing graph frameworks perform well for a subset of algorithms for specific types of input, but have suboptimal performances on algorithms with different bottlenecks and graphs of assorted sizes and structures [12, 86]. This performance inconsistency exists because each framework was designed to support only a limited set of optimization techniques, and does not allow for easy exploration of the large space of optimizations. It is infeasible to write hand-optimized implementations for every combination of algorithm and input type. A compiler approach that generates efficient implementations from high-level specifications is therefore a good fit. However, existing graph DSLs [46, 57, 2] do not support the composition of optimizations or expose comprehensive performance tuning capabilities for programmers.

Composing optimizations is quite challenging for graph processing frameworks for a few reasons. The framework first has to map out a comprehensive space of performance optimizations. Furthermore, the framework must develop an internal representation of the optimizations that the compiler can analyze to ensure correctness and perform code generation after multiple transformations. The framework also must support global program transformations that change various parts of the program. All the transformations and optimizations must not add extra runtime overhead. Finally, the framework must not sacrifice its programmability in the process of achieving high performance. As a result, existing frameworks have focused on a fixed set of optimizations instead of searching through a large space of optimizations.

1.2 GraphIt

To tackle the challenges of writing high-performance graph applications, we introduce GraphIt[1] [116, 114], a new graph DSL that provides a new high-level programming model and produces efficient implementations with performance that is competitive with or faster than the state-of-the-art frameworks for a diverse set of algorithms running on graphs with varied sizes and structures. GraphIt achieves good performance by enabling programmers to easily find the best combination of optimizations for their specific algorithm and input graph. Furthermore, GraphIt provides an easy-to-use programming model for programmers.

GraphIt makes it possible to easily and productively explore the space of optimizations by separating algorithm specifications from the choice of performance optimizations for graph applications. This design is inspired by previous work, such as Halide [80] for image processing and CHILL [23] for loop transformations. GraphIt is the first to map out the complex space of performance optimizations for graph computations and proposes a new high-level algorithm specification that covers both unordered and ordered graph algorithms.

[1]The GraphIt compiler is available under the MIT license at `http://graphit-lang.org/`

Programmers specify an algorithm using an *algorithm language* based on high-level operators on sets of vertices and edges, similar to Ligra [91]. The algorithm language simplifies expressing algorithms and exposes opportunities for optimizations by separating edge processing logic from edge traversal, edge filtering, vertex deduplication, and synchronization logic. We also introduced data structures and operators, including priority queue and priority update operators, to support ordered graph algorithms, where vertices are processed by their priorities. The algorithm language also hides the low-level implementation details, such as atomic synchronization, physical data layout (e.g., the array of structs (AoS), the struct of arrays (SoA), or bitvectors), and parallelization schemes.

Performance optimizations are specified using a separate *scheduling language*. We formulate graph optimizations, including edge traversal direction, data layout, parallelization, cache, non-uniform memory access (NUMA), bucket updates, and kernel fusion optimizations, as tradeoffs between locality, parallelism, and work-efficiency. The scheduling language enables programmers to easily search through the complicated tradeoff space by composing a large set of edge traversal, vertex data layout, active vertex process ordering, and program structure optimizations.

In addition to the algorithm and scheduling languages in the frontend of the compiler, GraphIt also introduces novel scheduling representations for edge traversal, vertex data layout, and program structure optimizations for the compiler's midend and backend. Inspired by the iteration space theory for dense loops [105, 76], we introduce an abstract *graph iteration space* model to represent, compose, and ensure the validity of edge traversal optimizations. We encode the graph iteration space in the compiler's intermediate representation of the compiler to guide program analyses and code generation. We designed the graph iteration space to be a multidimensional vector, similar to the iteration space vector for dense loops. We expand the vector with tags to represent the performance optimizations for each level of traversal.

The separation of the algorithm and schedule enables GraphIt to search for high-performance schedules automatically. The large scheduling space and long running time of the applications make it costly to do an exhaustive search. We demonstrate that it is possible to discover high-performance schedules in much less time using autotuning. Programmers familiar with graph optimizations can leverage their expertise to tune performance using the scheduling language directly. GraphIt achieves up to 4.8x speedup over state-of-the-art graph frameworks, while reducing the lines of code by up to one order of magnitude.

1.3 Contributions

This thesis makes the following contributions:

- Performance Optimizations
  - A systematic analysis and characterization of the fundamental tradeoffs between locality, work-efficiency, and parallelism in graph optimizations (Chapter 3).
  - Introduction of three novel performance optimizations, frequency-based clustering, compressed sparse row (CSR) segmenting, and bucket fusion (Chapter 3).
- Language Design and Implementation
  - A high-level programmer-friendly algorithm language that separates edge processing logic from edge traversal, synchronization, updated vertex tracking, active vertex process ordering, and deduplication logic, covering both unordered and ordered graph algorithms (Chapter 4).
  - A new scheduling language that allows programmers to explore the tradeoff space by composing edge traversal, vertex data layout, bucketing, and program structure optimizations for both unordered and ordered graph algorithms (Chapter 4).
  - A novel scheduling representation, the graph iteration space model, that can represent, combine and reason about the validity of various edge traversal optimizations (Chapter 5).
  - A compiler that leverages program analyses on the algorithm language and an intermediate representation that encodes the graph iteration space to generate efficient and valid implementations for different combinations of optimizations.

(Chapter 5).

- A comprehensive evaluation of GraphIt that demonstrates that it is faster than the next fastest state-of-the-art framework on 12 applications by up to 4.8×, while reducing the lines of code by up to one order of magnitude. (Chapter 6).

1.4 Outline

Chapter 2 discusses the graph algorithms and hardware background used throughout the thesis. Chapter 3 describes the performance optimizations and their tradeoffs. Chapter 4 introduces the programming model of GraphIt. Chapter 5 presents the design and implementation of the compiler. We evaluate the performance of GraphIt in Chapter 6. We discuss the limitations and future directions for GraphIt in Chapter 7. Finally, we survey the related work in Chapter 8 and conclude the work in Chapter 9.

Chapter 2

Background

In this chapter, we introduce the graph algorithms and present two motivating graph algorithms, PageRankDelta and Δ-stepping, which are used throughout the thesis. We also analyze the hardware performance bottlenecks of graph computations.

2.1 Definitions and Preliminaries

Graph frameworks typically store graphs in the Compressed Sparse Row (CSR) format. Assuming the graph has V vertices and E edges, CSR format would create a vertex array, `G.vertexArray` or `G.vertices`, of of length $O(V)$ and an edge array, `G.edgeArray`, of size $O(E)$. The vertex array stores the indices of the first neighbor of each vertex in the Edge Array and use that to access the neighbor list of each vertex or with the `G.getOutNgh(v)` API. Application specific data is stored as separate arrays. We use the term `frontier` to describe the current set of active vertices.

2.1.1 Graph Algorithms Overview

In this thesis, we define a graph algorithm as an algorithm that takes a single graph as input and performs some computations based on the input graph. This definition is just used in the scope of this thesis. In this thesis, we classify graph algorithms broadly into two main categories, *unordered* and *ordered* graph algorithms.

Ordered algorithms process active vertices following a dynamic priority-based ordering, potentially reducing redundant work. For example, single source shortest paths (SSSP) with the delta stepping algorithm [70] uses the shortest distance to the starting vertex as the priority for each vertex. Active vertices are processed in the order of their priorities. Another example is the peeling-based k-core algorithm [64]. The computations on the vertices are ordered by the degree of each node. The degree is updated dynamically as we remove the edges from the graph during the peeling procedure.

By contrast, unordered algorithms process active vertices in an arbitrary order, improving parallelism while potentially performing a significant amount of redundant work. For example, active vertices in PageRank or label propagation can be processed in any order in parallel. The processing order of the active vertices is not dictated by any priority values.

Not all of the graph applications have both unordered and ordered versions. Most of the graph applications have only an unordered algorithm. For example, PageRank only has an unordered algorithm.

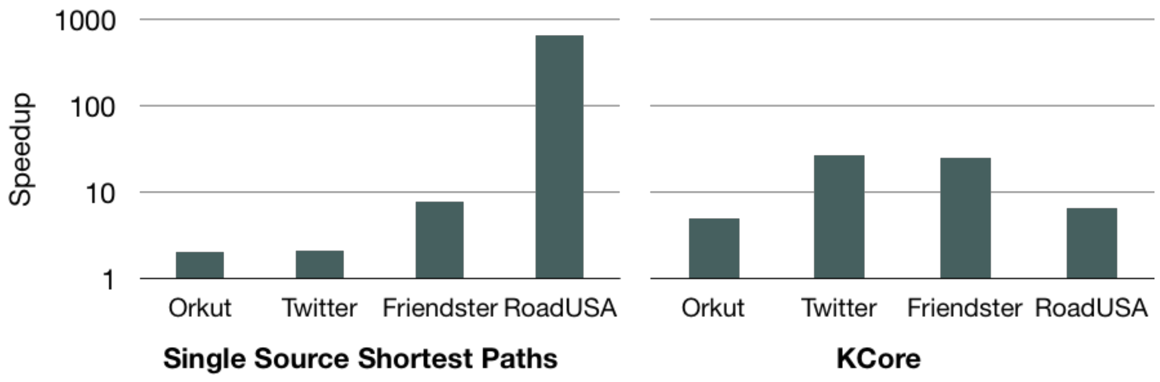


Figure 2-1: Speedup of ordered algorithms for single-source shortest path and k-core over the corresponding unordered algorithms implemented in our framework on a 24-core machine.

Some important graph problems can be implemented using either *ordered* or *unordered* parallel algorithms. Optimized ordered graph algorithms are up to two orders of magnitude faster than the unordered versions [32, 44, 14, 45], as shown in Figure 2-1. For example, computing SSSP on graphs with non-negative edge weights can be implemented using either the Bellman-Ford algorithm [15] (an unordered algorithm) or the Δ-stepping algorithm [70] (an ordered algorithm).[1] Bellman-Ford updates the shortest path distances to all active vertices on every iteration. On the other hand, Δ-stepping reduces the number of vertices that need to be processed every iteration by updating the path distances to the vertices that are closer to the source vertex first, before processing the vertices that are farther away.

2.1.2 Preliminaries for Ordered Graph Algorithms

We first define *ordered graph processing*, which is used throughout the thesis. Each vertex has a priority p_v. Initially, the users can explicitly initialize the priorities of vertices, with the default priority being a null value, $\emptyset$. These priorities change dynamically throughout the execution. However, the priorities can only change *monotonically*, that is they can only be increased, or only be decreased. We say that a vertex is *finalized* if its priority can no longer be updated. The vertices are processed and finalized based on the sorted priority ordering. By default, the ordered execution will stop when all vertices with non-null priority values are finalized. Alternatively, the user can define a customized stop condition, for example to halt once a certain vertex has been finalized.

We define *priority coarsening* as an optimization to coarsen the priority value of the vertex to p'_v by dividing the original priority by a coarsening factor Δ such that $p'_v = \lfloor p_v/\Delta \rfloor$. The optimization is inspired by Δ-stepping for SSSP, and enables greater parallelism at the cost of losing some algorithmic work-efficiency. Priority coarsening is used in algorithms that tolerate some priority inversions, such as A* search, SSSP, and point-to-point shortest path (PPSP), but not in k-core and SetCover, where the priority order is required for correctness.

[1]In this thesis, we define Δ-stepping as an ordered algorithm, in contrast to previous work [44] which defines Δ-stepping as an unordered algorithm.

```
Rank = $\{0, \ldots, 0\}$                                      ▷ Length $V$ array
DeltaSum = $\{0.0, \ldots, 0.0\}$                              ▷ Length $V$ array
Delta = $\{1/V, \ldots, 1/V\}$                                 ▷ Length $V$ array
procedure PageRankDelta(Graph $G$, $\alpha$, $\epsilon$)
    Frontier = { $G$.vertices }
    for round $\in \{1, \ldots,$MaxIter$\}$ do
        NextFrontier = {}                              ▷ Initialize the next frontiner
        parallel for src : Frontier do
            for dst : $G$.getOutNgh[src] do
                AtomicAdd(DeltaSum[dst], Delta[src]/G.OutDegree[src])
        end parallel for
        parallel for v : $G$.vertices do
            if round == 1 then
                BaseScore = $(1.0 - \alpha)/V$
                Delta[v] = $\alpha \cdot$ (DeltaSum[v])+ BaseScore
                Delta[v] $-$= $1/V$
            else
                Delta[v] = $\alpha \cdot$ (DeltaSum[v])
            Rank[v] += Delta[v]
            DeltaSum[v] = 0
            if |Delta[v]|> $\epsilon\cdot$Rank[v] then
                NextFrontier.add(v)
        end parallel for
        Frontier = NextFrontier
```

Figure 2-2: PageRankDelta (SparsePush).

2.2 Graph Algorithms

In this section, we describe two graph algorithms, PageRankDelta and Delta Stepping. PageRankDelta is an unordered graph algorithm and Delta Stepping is an ordered graph algorithm. The two algorithms are used throughout the thesis to demonstrate the performance optimizations and compiler mechanisms.

2.2.1 PageRankDelta

PageRankDelta [91] is a variant of the standard PageRank algorithm [77] that computes the importance of vertices in a graph. It is an unordered algorithm. It maintains an array of ranks, and on each iteration, updates the ranks of all vertices based on the ranks of their neighbors weighted by their neighbors' out-degrees. PageRankDelta speeds up the computation by updating only the ranks of vertices whose ranks have changed significantly from the previous iteration.

The pseudocode for PageRankDelta is shown in Fig. 2-2, where V is the number of vertices in the graph, $0 \leq \alpha \leq 1$ is the damping factor that determines how heavily to weight the neighbors' ranks during the update, and $\epsilon \geq 0$ is a constant that determines whether a vertex's rank has changed sufficiently. For simplicity, we update the ranks of vertices for MaxIter number of iterations, although the code can easily be modified to terminate based on a convergence criterion. The algorithm maintains the set of vertices whose ranks (stored in the Rank array) have changed significantly from the previous iteration in the variable Frontier (represented as a sparse array). We will refer to this as the *active set* of vertices, or the *frontier*. Initially all vertices are in the active set (Line 5). On each iteration, each vertex in the active set sends its Delta (change in Rank value) from the previous iteration to its out-neighbors by incrementing the DeltaSum values of its neighbors (Lines 8–10). Because vertices are processed in parallel, the updates to DeltaSum must be atomic. Then in parallel, all vertices compute their own Delta and Rank values based on their DeltaSum value and α (Lines 12–19). Delta is computed differently for the first iteration. If the Delta of the vertex is larger than ϵ times its Rank, then the vertex is active for the next

```
Dist = $\{\infty, \ldots, \infty\}$                                    ▷ Length $|V|$ array
procedure SSSP WITH Δ-STEPPING(Graph $G$, $\Delta$, startV)
    $B$ = new LazyBucket(Dist, $\Delta$, startV);            ▷ Initialize a new lazy bucket
    Dist[startV] = 0
    while ¬empty $B$ do
        ▷ Get the next bucket containing vertices of the smallest priority
        minBucket = $B$.getMinBucket()
        ▷ Create a new buffer to hold the updates to the buckets
        buffer = new BucketUpdateBuffer();
        ▷ Process each vertex in the current min bucket
        parallel for src : minBucket do
            for e : G.getOutEdge[src] do
                ▷ Update the distances of the neighbors
                Dist[e.dst] = min(Dist[e.dst], Dist[src] + e.weight)
                ▷ Compute the priority of the dst vertex and append it to the buffer
                buffer.syncAppend(e.dst, $\lfloor$Dist[e.dst]/$\Delta\rfloor$)
        end parallel for
        ▷ Reduce the bucket updates of the same vertex
        buffer = buffer.reduceBucketUpdates();
        ▷ Perform the final updates to the buckets based on the reduced updates
        B.bulkUpdateBuckets(buffer);
```

Figure 2-3: Δ-stepping for single-source shortest paths (SSSP) with the lazy bucket update approach.

iteration and is added to the next frontier (Lines 21–23).

2.2.2 Delta Stepping

Δ-stepping (shown in Figure 2-3) is an ordered algorithm that finds the shortest paths distance from the source vertex to all other vertices. In particular, it maintains buckets containing vertices at distances in $[i\Delta,\ (i+1)\Delta)$ for integer values of i starting at 0, and processes all vertices in a smaller bucket until convergence before moving to the next bucket. The algorithm finishes when the priorities of all vertices are finalized. Δ-stepping reduces the number of vertices that need to be processed every iteration by updating the path distances to the vertices that are closer to the source vertex first, before processing the vertices farther away.

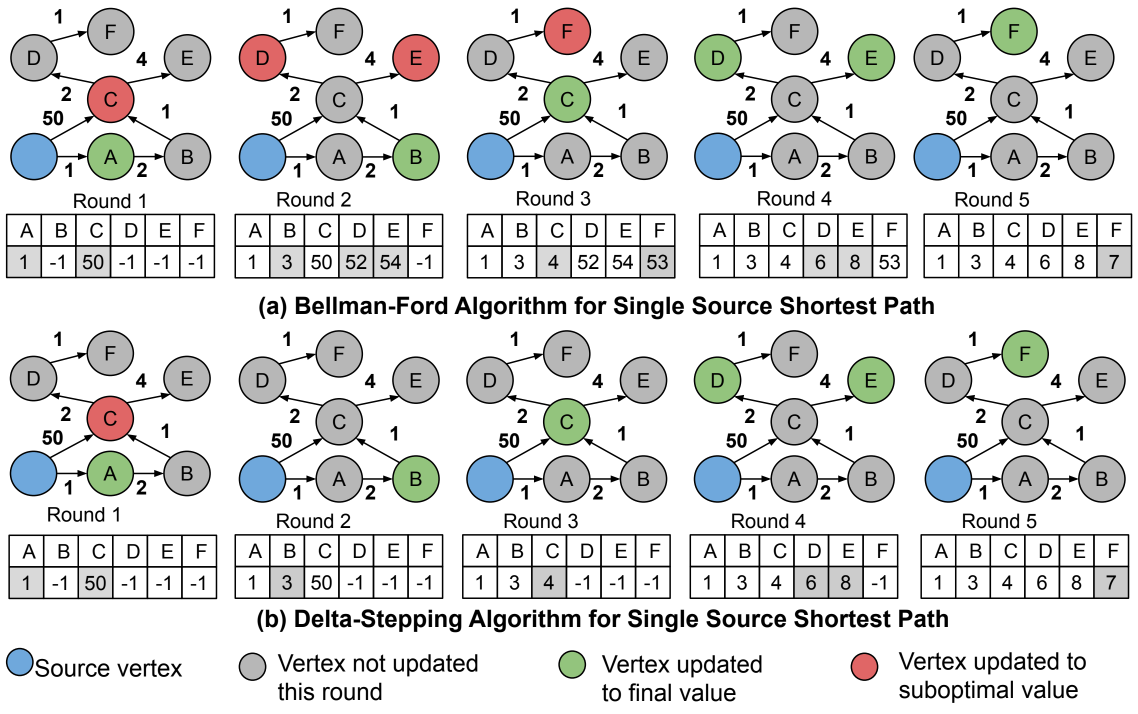
Bellman-Ford vs Δ-stepping. We show the unordered Bellman-Ford algorithm for SSSP in Figure 2-4. In the Bellman-Ford algorithm, all of the active vertices in the frontier are processed in parallel without a priority-based ordering as shown

```
 1 Dist = $\{\infty, \ldots, \infty\}$                                    ▷ Length $|V|$ array
 2 procedure SSSP WITH THE BELLMAN-FORD ALGORITHM(Graph $G$, $\Delta$, startV)
 3     Dist[startV] = 0
 4     n = G.numVertices()
 5     frontier = { startV }
 6     rounds =0
 7     while ¬empty Frontier do
 8         nextFrontier = {}
 9         ▷ Process vertices in the frontier in parallel
10         parallel for src : frontier do
11             for e : G.getOutEdge[src] do
12                 if Dist[e.dst] > Dist[src] + e.weight then
13                     ▷ Update the distance of the neighbor
14                     Dist[e.dst] = Dist[src] + e.weight
15                     nextFrontier.atomicAdd(e.dst)
16         end parallel for
17         rounds++
18         frontier = nextFrontier
19         ▷ Check if there is a cycle with negative edge weight
20         if rounds == n then
21             print "Negative Weight Cycle Found"
22             break
```

Figure 2-4: The Bellman-Ford algorithm for single-source shortest paths (SSSP).

in Figure 2-4 Line 10. The algorithm keeps track of vertices that are updated in the current round (Figure 2-4 Line 15) and uses them in the next round (Figure 2-4 Line 18). The algorithm ends when the frontier becomes empty (Figure 2-4 Line 7).

We use Figure 2-5 to demonstrate the difference between the Bellman-Ford (unordered algorithm) and the Δ-stepping (ordered algorithm) algorithms for SSSP. Bellman-Ford updates the distance to vertex C in round 1, and then immediately propagates updates to vertices D, E, and F in both rounds 2 and 3. However, these updates (highlighted in red) constitute wasted work because the correct shortest path distance for vertex C is set through a three-hop path through vertices A and B. Δ-stepping only propagate updates to vertices D, E, and F after the distance to vertex C is finalized in round 3. The Δ-stepping algorithm sacrifices some parallelism, but avoids redundant updates, leading to a significant speedup over the Bellman-Ford algorithm on many graphs.


Round 1
Round 2
Round 3
Round 4
Round 5
(a) Bellman-Ford Algorithm for Single Source Shortest Path
Round 1
Round 2
Round 3
Round 4
Round 5
(b) Delta-Stepping Algorithm for Single Source Shortest Path
Source vertex
Vertex not updated this round
Vertex updated to final value
Vertex updated to suboptimal value

(a) Bellman-Ford, Round 1

A	B	C	D	E	F
1	-1	50	-1	-1	-1

(a) Bellman-Ford, Round 2

A	B	C	D	E	F
1	3	50	52	54	-1

(a) Bellman-Ford, Round 3

A	B	C	D	E	F
1	3	4	52	54	53

(a) Bellman-Ford, Round 4

A	B	C	D	E	F
1	3	4	6	8	53

(a) Bellman-Ford, Round 5

A	B	C	D	E	F
1	3	4	6	8	7

(b) Delta-Stepping, Round 1

A	B	C	D	E	F
1	-1	50	-1	-1	-1

(b) Delta-Stepping, Round 2

A	B	C	D	E	F
1	3	50	-1	-1	-1

(b) Delta-Stepping, Round 3

A	B	C	D	E	F
1	3	4	-1	-1	-1

(b) Delta-Stepping, Round 4

A	B	C	D	E	F
1	3	4	6	8	-1

(b) Delta-Stepping, Round 5

A	B	C	D	E	F
1	3	4	6	8	7

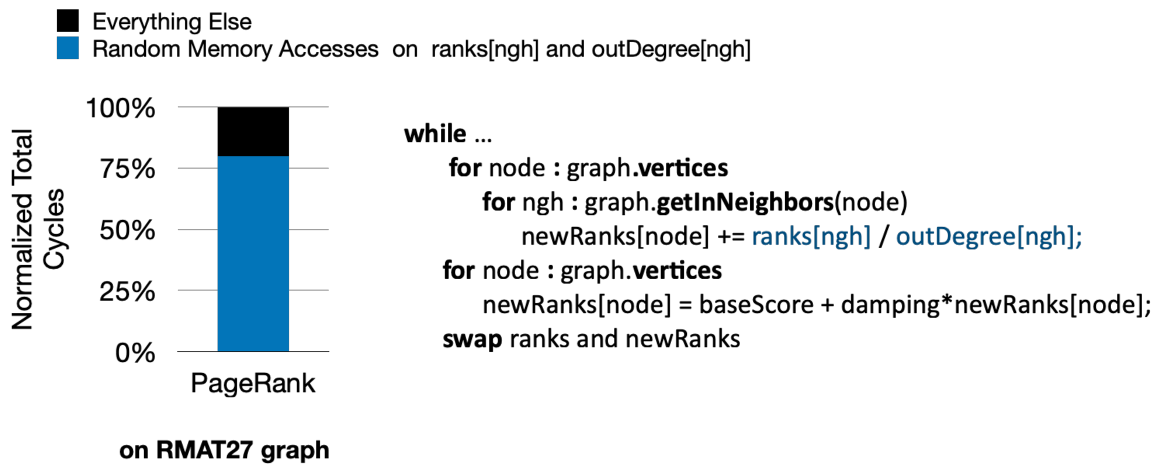
Figure 2-5: SSSP implemented with Bellman-Ford and Δ-stepping ($\Delta = 30$). The distances are shown in the tables. Bellman-Ford needs 10 updates, whereas Δ-stepping needs only seven updates.

2.3 Hardware Performance Bottleneck

We use PageRank listed in Algorithm 1 as a running example to illustrate the performance bottlenecks and motivate our optimizations for graph processing. PageRank iteratively updates the rank of each vertex based on the rank and degree of its neighbors. The performance characteristics of PageRank can generalize to numerous graph applications.

In the case of PageRank, vertex data is stored as arrays `newRank`, `rank` and `degree` of length $O(V)$. The algorithm sequentially reads size $O(E)$ data. By going over every vertex in order, the algorithm issues sequential read requests to `G.edgeArray` and sequential writes requests to `newRank`. The algorithm randomly reads $O(E)$ times from size $O(V)$ vertex data, including `rank` and `degree`. These read requests are random because we cannot predict the values of `u`.

This pattern of sequentially accessed edge data and randomly accessed vertex data is common in representative graph applications. For example, collaborative filtering



```
while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh] / outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Figure 2-6: Cycles Breakdown of running PageRank on the rmat27 graph. The cycles in the blue part are attributed to the random memory accesses for neighbors' ranks and outDegree.

must randomly read each vertex's latent factors, and Betweenness Centrality needs to randomly access the active frontier and number of paths through each vertex.

Memory Access Stalls. Graph applications have poor cache hit rates and are largely stalled on memory accesses because the working set of realistic graphs is much larger than the last level cache (LLC) of current machines. For example, the Twitter graph [55] has 41 million vertices and 1.5 billion edges. The `rank` and `degree` arrays, which together form the working set that is randomly accessed, are 656 MB (assuming 64-bit doubles) and are many times larger than the 30 to 55 MB LLC of the current CPUs. Even though a higher than expected hit rate exists due to the power-law degree distribution and the community structures in the graph [12], we still find the LLC miss rate for PageRank to be more than 45%. As a result of the high cache miss rates, our performance profile demonstrates that the graph applications are spending 60% to 80% of their cycles stalled on memory access, as depicted in Figure 2-6. Random memory access becomes the major bottleneck because random access to DRAM is 6x to 8x more expensive than random access to LLC or sequential accesses to DRAM. Sequential access to DRAM effectively uses all memory bandwidth, and benefits from hardware prefetchers to further reduce latency.

Load Balance. Load balance is another major challenge for graph applications to reach scalable performance. The number of neighbors of each vertex has (degree of each vertex) are often distributed in a power-law fashion for many graphs. A small number of vertices are adjacent to a large number of edges. As a result, simple approaches to statically partition the graph by vertex for parallelism often lead to poor performance. However, partitioning by edges provides better parallelism but potentially incurs a large synchronization overhead.

2.4 Chapter Summary

In this chapter, we first defined ordered and unordered graph algorithms and described two algorithms, PageRankDelta and Delta Stepping in detail. We also analyzed the performance bottlenecks of the graph algorithms, including memory access stalls and load balance. We use PageRankDelta and Delta Stepping to illustrate the various performance optimizations in Chapter 3, the programming model of GraphIt in Chapter 4, and compiler designs in Chapter 5. Chapter 3 discusses how the performance optimizations address the performance bottlenecks of graph algorithms.

Chapter 3

Optimizations

Writing high-performance graph applications is challenging because no set of optimizations is good for all applications as we indicated in Chapter 1. To attain high performance, programmers must choose a combination of performance optimizations, which makes tradeoffs between locality, work-efficiency, and parallelism, which is best suited for their specific algorithm and graph. No single set of optimizations works well across all the different algorithms and graphs. In this chapter, we first analyze the existing optimizations to map out the tradeoff space between locality, work-efficiency, and parallelism. We then introduce a few novel optimizations that further expand the optimization space, including CSR Segmenting, frequency-based clustering, and bucket fusion.

3.1 Performance Tradeoff Space

In this section, we first describe the tradeoff space for optimizing graph programs. Although the effects of various optimizations are well-known to expert programmers, we believe that we are the first to characterize the optimization tradeoff space for graph optimizations. Our tradeoff space includes three properties of graph programs—locality, work-efficiency, and parallelism.

Locality refers to the amount of spatial and temporal reuse in a program. For example, increasing the amount of temporal reuse improves locality due to increasing the number of cache hits. In a NUMA system, locality also refers to the memory location relative to the processor. Increasing the ratio of local memory accesses to remote memory accesses improves locality. Other optimizations that place data accessed together in the same cache line significantly improve spatial locality, such as using an array of structures versus using a struct of arrays.

Work-efficiency is the inverse of the number of instructions, where each instruction is weighted according to the number of cycles that it takes to execute assuming a cold cache. Reducing the number of instructions improves work-efficiency. We can improve work-efficiency by reducing the runtime overhead or using a more efficient algorithm. For example, some optimizations need to create auxiliary data structures, such as bitvector and cache-blocked graphs, to improve locality. These data structures require many extra instructions to build and access. Using these data structures increases the amount of work and decreases the work-efficiency. In terms of algorithmic efficiency, ordered graph algorithms use fewer instructions than their unordered counterparts by enforcing a priority-based ordering for active vertices.

Parallelism refers to the relative amount of work that can be executed independently by different processing units, which is often affected by the load balance and synchronization among processing units. Increasing parallelism improves performance by taking advantage of more processing units, and helping to hide the latency of DRAM requests. However, exploiting parallelism often causes extra work, such as synchronization overhead and redundant computations. Additionally, using more parallelism than what the hardware can support is also unproductive. As a result, parallelism can sometimes hurt the performance of the program.

We use PageRankDelta and Δ-stepping (described in Chapter 2) as examples to illustrate the effects of various optimizations on these three metrics. Each optimization can affect multiple properties in the tradeoff space, either positively or negatively. The complex tradeoff space motivates the design of GraphIt's scheduling language and compiler, which can be used to easily search for points in the tradeoff space that achieve high performance.

Table 3.1: Effect of optimizations on the different properties of the tradeoff space relative to the baseline SparsePush. For each property, ↑ means positive impact on performance, ↓ means negative impact on performance, ↕ means it could increase or decrease depending on various factors (described in the text), and no entry means no effect on performance.

Optimization	Locality	Work-efficiency	Parallelism
DensePull		↕	↕
DensePush		↕	↑
DensePull-SparsePush		↕	↕
DensePush-SparsePush		↕	↑
edge-aware-vertex-parallel		↓	↑
edge-parallel		↓	↑
bitvector	↑	↓	
vertex data layout	↕	↓	
cache partitioning	↑	↓	
NUMA partitioning	↑	↓	↓
kernel fusion	↑	↓	

```
parallel for src : Frontier do
    for dst : $G$.getOutNgh[src] do
        AtomicAdd(DeltaSum[dst],Delta[src] / G.OutDegree[src])
end parallel for
```

Figure 3-1: SparsePush

3.2 Graph Traversal Optimizations

Traversing through all the edges or a subset of edges in a graph is a fundamental operator in many graph algorithms. Many different strategies can optimize the performance of graph traversals, such as tuning the direction of the traversal and the degree of parallelization, and using different data layout strategies to represent the vertex sets. In this section, we describe the tradeoffs of the graph traversal optimizations listed in Table 3.1 with PageRankDelta (Table 8.1 describes which optimizations are supported by which frameworks and DSLs). Table 3.1 contains the effect of the optimizations relative to the baseline in Fig. 2-2, which we refer to as *SparsePush*.

```
parallel for dst : $G$.vertices do
    for src : $G$.getInNgh[dst] do
        if src $\in$ Frontier then
            DeltaSum[dst] += Delta[src] / G.OutDegree[src]
end parallel for
```

Figure 3-2: DensePull

```
parallel for src : $G$.vertices do
    if src $\in$ Frontier then
        for dst : $G$.getOutNgh[src] do
            AtomicAdd(DeltaSum[dst],Delta[src] / G.OutDegree[src])
end parallel for
```

Figure 3-3: DensePush

3.2.1 Direction Optimization and Frontier Data Structure

Figure 2-2 lines 8–10 of SparsePush shown in Figure 3-1 iterate over the outgoing neighbors of each vertex, and update the DeltaSum value of the neighbor. *DensePull* (Fig. 3-5) is a different traversal mode where each vertex iterates over its incoming neighbors that are in the active set, and updates its own DeltaSum value. DensePull increases parallelism relative to SparsePush because it loops over all vertices in the graph. This increases work compared to SparsePush, which only loops over vertices in the active set. The update to the DeltaSum array no longer requires atomics since an entry will not be updated in parallel, and this reduces synchronization overhead. Instead of performing random writes as in SparsePush, DensePull performs random reads and mostly sequential writes, which are cheaper. For some algorithms (e.g., breadth-first search), the inner loop over the in-neighbors in DensePull can exit early to reduce overall work. Therefore, the overall number of edges traversed could increase or decrease. A detailed performance study of the two traversal methods is found in [11] and [18]. We can further use *bitvectors* instead of boolean arrays to keep track of vertices on the frontier for the DensePull direction. A dense frontier implemented using a bitvector improves the spatial locality but requires extra work to compress the boolean array.

DensePush (Fig. 3-3) loops through all vertices and checks whether each one is on

```
if Frontier.size() < threshold then
    perform SparsePush
else
    perform DensePull
```

Figure 3-4: DensePull-SparsePush

```
if Frontier.size() < threshold then
    perform SparsePush
else
    perform DensePush
```

Figure 3-5: DensePush-SparsePush

the frontier instead of only looping over frontier vertices as in SparsePush. Although iterating over all vertices reduces work-efficiency, this could be offset by not having to maintain the frontier in a sparse format. Parallelism increases because there is more parallel work when looping over all vertices.

Hybrid traversal modes, including *DensePull-SparsePush*(Figure 3-4) and *DensePush-SparsePush* (Figure 3-5), use different directions (SparsePush, DensePull, and DensePush) in different iterations based on the size of the active set to improve work-efficiency [11, 91]. In PageRankDelta, the number of vertices in the frontier gradually decreases as the ranks of vertices converge. In the early iterations, DensePull is preferred due to lower synchronization overheads and avoidance of random writes. As the frontier shrinks, SparsePush is preferred due to the fewer number of vertices that need to be traversed. *DensePull-SparsePush* computes the sum of out-degrees of the frontier vertices and uses DensePull if the sum is above some threshold and uses SparsePush otherwise. However, computing the sum of out-degrees of the vertices in the active set in every iteration incurs significant overhead if one direction is always better than the other.

GraphIt is able to support all of these traversal directions, whereas the existing frameworks only support a subset of them. GraphIt also supports both bitvectors and boolean arrays for the frontier representation in dense traversals, as well as the sparse array representation for sparse traversals. We show the performance of different traversal directions in Section 6.2.4

Algorithm 2 Preprocessing for CSR Segmenting

```
Input: Number of vertices per segment N, Graph G
for v : G.vertices do
    for inEdge : G.inEdges(v) do
        segmentID← inEdge.src / N
        subgraphs[segmentID].addInEdge(v,inEdge.src)
for subgraph : subgraphs do
    subgraph.sortByDestination()
    subgraph.constructIdxMap()
    subgraph.constructBlockIndices()
    subgraph.constructIntermBuf()
```

3.2.2 Parallelization

For each traversal mode, there are different methods for parallelization. The parallelization shown in Fig. 3-1, 3-3 and 3-5 processes the vertices in parallel. The *vertex-parallel* approach works well on algorithms and inputs where the workload of each vertex is similar. However, if the degree distribution is skewed and the workload of each vertex is proportional to the number of incident edges, this approach can lead to significant load imbalance. For these workloads, an edge-aware vertex-parallel scheme (*edge-aware-vertex-parallel*) can be more effective. This approach breaks up the vertices into a number of vertex chunks, where each chunk has approximately the same number of edges. However, this scheme reduces work-efficiency due to having to compute the sum of degrees of vertices in each chunk. Finally, we can parallelize across all edges, instead of just vertices, by parallelizing the inner loop of the edge-traversal code computing DeltaSum in Fig. 3-1 and 3-5. This method (*edge-parallel*) improves parallelism but reduces work-efficiency due to the overhead of work-stealing in the inner loop and atomic updates needed for synchronization. For graphs with a regular degree distribution, using static parallelism instead of work-stealing parallelism can sometimes reduce runtime overhead and improve performance. GraphIt supports all three modes of parallelism, whereas the existing frameworks only support one or two.

Algorithm 3 Parallel Segment Processing

```
for subgraph : subgraphs do
    parallel for v : subgraph.Vertices do
        for inEdge: subgraph.inEdges(v) do
            Process inEdge
    end parallel for
```

3.2.3 Cache and NUMA Optimizations

CSR Segmenting [115] is a graph partitioning technique that keeps random accesses within the last level cache (LLC) to improve locality. This optimization first partitions the vertices into p segments $(V_0, V_1, \ldots, V_{p-1})$, which correspond to the range of source vertexsets in the pull mode or the destination vertexsets in the push mode for each Segmented Subgraph (SSG). For the pull mode, incoming edges (src, dst) are assigned to SSG_i if $src \in V_i$, and sorted by dst. For the push mode, outgoing edges (src, dst) are assigned to SSG_i if $dst \in V_i$, and sorted by src. Each SSG is processed before moving on to the next. V_i controls the range of random memory accesses through segmenting the original graph.

The preprocessing (partitioning) algorithm is presented in Algorithm 2. The first step is to construct the subgraphs based on the segments. We first divide the vertices into segments such that the data for each segment fits in the cache. For each segment S, we construct a new subgraph consisting of edges whose sources are in the segment. To do this, we compute the segmentID(subgraphID) of each inEdge by dividing the sourceID of the inEdge by the number of vertices in each segment N and then add the edge to the subgraph. The edges in each subgraph are sorted by their destinations ($sortByDestination$). This step takes no time since the original graph in CSR is already sorted by destination. Then, a CSR representation will be constructed for each subgraph. The algorithm also creates an array, $intermBuf$, to hold the intermediate result for each destination vertex v. Additionally, we create an index mapping, $idxMap$, to map the local index of destination vertices in the subgraph to their global index in the original graph. Finally, we create an index of blocks that stores block starts and ends used in the cache-aware merge. This preprocessing phase can be done in parallel by building each subgraph separately from the original CSR.

After the preprocessing is done, the system processes each subgraph in turn as shown in Algorithm 3. Within each subgraph, we parallelize the computation across different vertices. If we fit V_i into LLC, we can significantly reduce the number of random DRAM accesses when processing each SSG. Cache partitioning improves locality but sacrifices work-efficiency due to the vertex data replication from the graph partitioning and merging partial results [115, 13, 75]. Fine-tuning the number of SSGs can reduce this overhead. Most existing frameworks do not support cache partitioning.

Moreover, NUMA partitioning improves locality by minimizing the slow inter-socket memory accesses [112, 94, 118]. This optimization partitions the graph into a set of Segmented Subgraphs (SSGs) in the same way as cache partitioning in order to limit the range of random memory access. While the cache partitioning optimization processes one SSG at a time across all sockets, NUMA partitioning executes multiple SSGs in parallel on different sockets. Each SSG and the threads responsible for processing the subgraph are bound to the same NUMA socket. The intermediate results collected on each socket are merged at the end of each iteration. As with cache partitioning, NUMA partitioning improves locality but reduces work-efficiency due to vertex data replication from graph partitioning and the additional merge phase. Parallelism might also decrease in highly skewed graphs due to the workload imbalance among SSGs [94]. For algorithms with performance bottlenecked on load imbalance instead of inter-socket memory accesses, simply using an interleaved allocation across sockets can result in better performance. GraphIt can do both cache and NUMA optimizations hierarchically.

We study the results of these optimizations in Section 6.2.3 and Section 6.2.4. We use the performance counters in the machines to demonstrate improved cache miss rates and reduced inter-socket traffic.

3.2.4 Vertex Data Layout Optimizations

The layout of vertex data can significantly affect the locality of memory accesses. Random accesses to the same index of two separate arrays (e.g., the Delta and

OutDegree arrays in PageRankDelta) can be changed into a single random access to an array of structs to improve spatial locality. However, grouping together fields that are not always accessed together expands the working set and hurts the spatial locality of the data structures. Vertex data layout optimizations reduce work-efficiency due to the extra overhead for reorganizing the data. GraphIt supports data representations of both arrays of structs (AoSs) and structs of arrays (SoAs). The users can use both array of structs and structs of arrays for different data in the same program.

3.2.5 Program Structure Optimizations

When two graph kernels have the same traversal pattern (they process the same vertices/edges on each iteration), we can fuse together the edge traversals and transform their data structures into an array of structs. We refer to this optimization as *kernel fusion*. This improves spatial locality by enabling the program to access the fields of the two kernels together when traversing the edges. Additional work is incurred for performing the AoS-SoA optimization, but this is usually small compared to the rest of the algorithm. We demonstrate the effect of the program structure optimizations in Section 6.2.5.

3.3 Frequency-Based Clustering

Frequency-Based Clustering [115] is an out-degree based graph reordering technique, to further boost cache line utilization and keep frequently accessed vertices in fast cache. Frequency-Based Clustering reorganizes the physical layout of the vertex data structures to improve cache utilization. It reduces overall cycles stalled on memory by serving more random requests in fast storage.

We make three key observations on graph access patterns to motivate frequency-based clustering. First, each random read in graph applications often only uses a small portion of the cache line. For PageRank, the size of the vertex data is 8 bytes for a rank represented as a double, using only 1/8 of a common 64 byte cache line. Because there is little spatial locality, the other elements in the cache line are often

Algorithm 4 Frequency-based Clustering

```
1  Input:  Number of vertices n, Number of edges m, Graph G stored in a CSR format
   with a vertex array (G.vertexArray) and an edge array (G.edgeArray)
2  ▷ mapping from the new reordered vertexIDs to the old vertexIDs
3  vmap ← vector<int>(n)
4  ▷ mapping from the old vertexIDs to the new reordered vertexIDs
5  rmap ← vector<int>(n)
6  newVertexArray ← vector<int>(n)
7  newEdgeArray ← vector<int>(m)
8  for $v : G.vertices$ do
9      vmap[v] ← v
10 ▷ Sort the vertices based on their degree if their degrees are higher than the threshold.
   This maintains their original ordering if the degrees are lower than the threshold.
11 func comp = [](int a, int b) return G.degree[a]/threshold < G.degree[b]/threshold
12 stableSort(vMap.begin(), vMap.end(), comp)
13 for $i : 0, ..., N$ do ▷ Set up the reverse mapping from the old vertexIDs to new reordered
   vertexIDs
14     rmap[vmap[i]] ← i
15 edgeIdx ← 0
16 for $i : 0, ..., N$ do                                    ▷ Create a new relabeled CSR
17     newVertexArray[i] ← edgeIdx
18     oldV ← vMap[i]
19     for $j : 0, ..., G.degree[newV]$ do
20         offset ← G.VertexArray[oldV]
21         newEdgeArray[edgeIdx++] ← rmap[G.edgeArray[offset+j]]
```

not used. This is true for many other graph applications, such as label propagation, which reads an integer type vertex label. Second, certain vertices are much more likely to be accessed than others in power law distributed graphs, where a small number of vertices have a large number of edges attached to them [55]. Thus, a large percentage of random read requests will concentrate on a small subset of vertices. These skewed out-degree graphs include social networks, web graphs, and many networks in biology. Because of the above observations, if we store the vertices in a random order, each high out-degree vertex will likely be on a different cache line in the vertex data array (e.g., `rank` in PageRank). The cache line will be "polluted" by the data from low out-degree vertices when it is brought in. A third observation is that the original ordering of vertices in real world graphs often exhibit some locality. Vertices that are referenced together are sometimes placed close to each other due to the communities existing in these graphs. For example, PageRank on the original ordering of vertices on Twitter graph [55] is 50% faster than a random ordering. Thus, it is also important to utilize the original ordering for improved performance.

Design and Implementation: We designed frequency-based clustering to group the vertices that are frequently referenced, while preserving the natural ordering in the real-world graphs. We present frequency-based clustering in Algorithm 4. We use out-degrees to select the frequently accessed vertices because many graph algorithms use only pull-based implementations, or spend a significant portion of the execution time in the pull phase. To preserve the original ordering in real world graphs, we cluster together only vertices with out-degree above the average degree of nodes (this threshold can be tuned). This thresholding allows us to maintain some of the locality in the original ordering, yet still offering a clustering of high-out-degree vertices that maximizes the effectiveness of L1, L2, and L3 caches.

We use a parallel stable sort based on the vertices' out-degree/threshold to cluster together frequently referenced vertices (Algorithm 4 Line 12). Next, we create a mapping from the old vertex index to the newly sorted vertex index (Algorithm 4 Line 13) and use the mapping to update the vertex index in the `G.edgeArray`.

Load balance is critical to achieving high performance with frequency-based clustering. The thread responsible for the part of the vertex array containing high out-degree vertices may perform much more work than other threads. We implemented a work-estimating load balancing scheme that partitions the vertex array based on the number of edges within each task, which reflects how many random reads it makes to the `rank` array. The task then processes its range of vertices if the cost is sufficiently small, or divides it into two sub-tasks otherwise.

Performance Impact We show the performance improvements of frequency-based clustering in Table 3.2. The optimization is most effective on graphs with random order (RMAT graphs) and are more effective on topology-driven algorithms, such as PageRank. The performance of frequecy-based clustering is studied in greater details in [115] and [10].

Table 3.2: Speedups of Frequency-Based Clustering on BFS, PageRank, Betweenness Centrality (BC). The experiments are run on a 2-socket, 24-core Intel Xeon E5-2695 v2 CPU. The system has 128 GB of DDR3 memory and 30 MB last level cache on each socket.

BFS	BaseLine (sec)	Frequency-based Clustering (sec)	Speedup
Live Journal	0.33	0.35	0.94
Twitter	3.18	3.16	1.01
RMAT25	1.42	1.18	1.20
RMAT27	7.02	4.82	1.46
PageRank	BaseLin (sec)	Frequency-based Clustering (sec)	Speedup
Live Journal	0.02	0.02	1.0
Twitter	0.75	0.624	1.20
RMAT25	0.33	0.20	1.66
RMAT27	1.54	0.94	1.64
BC	BaseLine (sec)	Frequency-based Clustering (sec)	Speedup
Live Journal	1.19	1.21	0.98
Twitter	17.5	16.90	1.04
RMAT25	11.1	8.02	1.38
RMAT27	42.8	25.4	1.69

3.4 Bucketing Optimizations

The priority-based extension to GraphIt uses a *bucketing* data structure [32, 14] to maintain the execution ordering for the ordered graph algorithms. Each bucket stores active vertices of the same priority, and the buckets are sorted in priority order. The program processes one bucket at a time in priority order and dynamically moves active vertices to new buckets when their priorities change. Updates to the bucket structure can be implemented using either an *eager bucket update* [14] approach or a *lazy bucket update* [32] approach.

With eager bucket updates, buckets are immediately updated when the priorities of active vertices change. Lazy bucketing buffers the updates and later performs a single bucket update per vertex.

Bucketing incurs high synchronization overheads, slowing down algorithms that spend most of their time on bucket operations. We introduce a new performance optimization, *bucket fusion*, which drastically reduces synchronization overheads for eager bucket updates. In an ordered algorithm, a bucket can be processed in multiple rounds under a bulk synchronous processing execution model. In every round, the current bucket is emptied and vertices whose priorities are updated to the current

```
 1 Dist = $\{\infty, \ldots, \infty\}$                                ▷ Length $|V|$ array
 2 procedure SSSP WITH Δ-STEPPING(Graph $G$, $\Delta$, startV)
 3     $B$ = new LazyBucket(Dist, $\Delta$, startV);
 4     Dist[startV] = 0
 5     while ¬empty $B$ do
 6         minBucket = $B$.getMinBucket()
 7         buffer = new BucketUpdateBuffer();
 8         ▷ Process vertices in the next smallest bucket in parallel
 9         parallel for src : minBucket do
10             for e : G.getOutEdge[src] do
11                 ▷ Relax the edge
12                 Dist[e.dst] = min(Dist[e.dst], Dist[src] + e.weight)
13                 ▷ Add the bucket update to the buffer
14                 buffer.syncAppend(e.dst, $\lfloor$Dist[e.dst]/$\Delta\rfloor$)
15         buffer = buffer.reduceBucketUpdates();
16         ▷ Apply the reduced bucket updates
17         B.bulkUpdateBuckets(buffer);
```

Figure 3-6: Δ-stepping for single-source shortest paths (SSSP) with the lazy bucket update approach.

bucket's priority are added to the bucket. The algorithm moves on to the next bucket when no more vertices are added to the current bucket. The key idea of bucket fusion is to *fuse consecutive rounds that process the same bucket*. Using bucket fusion in GraphIt results in 1.2×–3× speedup on road networks with large diameters compared to the existing work.

We use Δ-stepping for single-source shortest paths (SSSP) as a running example to illustrate the performance tradeoffs between two different bucket update strategies, including the lazy and eager bucket update approaches, in Section 3.4.1 and Section 3.4.2. We also introduce our new bucket fusion optimization that can significantly improve the efficiency of the eager bucket approach on road networks in Section 3.4.3.

3.4.1 Lazy Bucket Update

We first consider using the lazy bucket update approach for the Δ-stepping algorithm, with the pseudocode illustrated in Figure 3-6. The algorithm constructs a bucketing data structure in Line 3, which groups the vertices into *buckets* according to their priorities. It then repeatedly extracts the bucket with the minimum priority (Line 6), and finishes the computation once all of the buckets have been processed (Lines 5–17). To process a bucket, the algorithm iterates over each vertex in the bucket, and updates the priority of its outgoing neighbor destination vertices by updating the neighbor's distance (Line 12). With priority coarsening, the algorithm computes the new priority by dividing the distance by the coarsening factor, Δ. The corresponding bucket update (the vertex and its updated priority) is added to a buffer with a synchronized append (Line 14). The `syncAppend` can be implemented using atomic operations, or with a prefix sum to avoid atomics. The buffer is later reduced so that each vertex will only has one final bucket update (Line 15). Finally, the buckets are updated in bulk with `bulkUpdateBuckets` (Line 17).

The lazy bucket update approach can be very efficient when a vertex changes buckets multiple times within a round. The lazy approach buffers the bucket updates, and makes a single insertion to the final bucket. Furthermore, the lazy approach can be combined with other optimizations such as the histogram-based reduction of priority updates to further reduce runtime overheads. However, the lazy approach adds additional runtime overhead from maintaining a buffer (Line 7), and performing reductions on the buffer (Line 15) at the end of each round. These overheads can incur a significant cost in cases where there are only a few updates per round (e.g., in SSSP on large diameter road networks).

3.4.2 Eager Bucket Update

Another approach for implementing Δ-stepping is to use an eager bucket update approach (shown in Figure 3-7) that directly updates the bucket of a vertex when its priority changes. The algorithm is naturally implemented using thread-local buckets, which are updated in parallel across different threads (Line 13). Each thread works on a disjoint subset of vertices in the current bucket (Line 14). Using thread-local buckets avoids atomic synchronization overheads on bucket updates (Lines 3 and 17–19). To extract the next bucket, the algorithm first identifies the smallest priority across all threads and then has each thread copy over its local bucket of that priority to a global minBucket (Line 11). If a thread does not have a local bucket of the next smallest

```
Dist = {$\infty, \dots, \infty$}                                    ▷ Length $|V|$ array
procedure SSSP WITH $\Delta$-STEPPING(Graph $G$, $\Delta$, startV)
    $B$ = new ThreadLocalBuckets(Dist, $\Delta$, startV);
    ▷ Initialize the thread-local buckets for each thread
    for threadID : threads do
        $B$.append(new LocalBucket());
    Dist[startV] = 0
    ▷ While there are still vertices left to be updated
    while $\neg$empty $B$ do
        ▷ Get a global frontier created from the buckets of the next smallest priority across
all threads
        minBucket = $B$.getGlobalMinBucket()
        ▷ Each thread processes a portion of the global frontier
        parallel for threadID : threads do
            for src : minBucket.getVertices(threadID) do
                for e : G.getOutEdge[src] do
                    ▷ Relax the edge weights
                    Dist[e.dst] = min(Dist[e.dst], Dist[src] + e.weight)
                    ▷ Update the appropriate bucket
                    $B$[threadID].updateBucket(e.dst, $\lfloor$Dist[e.dst]/$\Delta\rfloor$)
```

Figure 3-7: Δ-stepping for SSSP with the eager bucket update approach.

priority, then it will skips the copying process. Copying local buckets into a global bucket helps redistribute the work among threads for better load balancing.

Compared to the lazy bucket update approach, the eager approach saves instructions and one global synchronization needed for reducing bucket updates in the buffer (Figure 2-3, Line 19). However, it potentially needs to perform multiple bucket updates per vertex in each round. We show detailed performance comparison of lazy and eager approaches in Section 6.3.5.

3.4.3 Eager Bucket Fusion Optimization

In this section, we explain the motivation, design, and implementation of our new bucket fusion optimization for eager bucketing. The bucket fusion optimization can achieve more than 3× speed up for many ordered graph algorithms running on graphs with large diameters, such as road networks. This optimization has since been integrated into the popular GAP benchmark suite [14].

A major challenge in bucketing is that many buckets need to be processed, resulting

```
Dist = $\{\infty, \ldots, \infty\}$                              ▷ Length $|V|$ array
procedure SSSP WITH Δ-STEPPING(Graph $G$, $\Delta$, startV)
    $B$ = new ThreadLocalBuckets(Dist, $\Delta$, startV);
    for threadID : threads do
        $B$.append(new LocalBucket());
    Dist[startV] = 0
    while $\neg$empty $B$ do
        minBucket = $B$.getMinBucket()
        parallel for threadID : threads do
            for src : minBucket.getVertices(threadID) do
                for e : G.getOutEdge[src] do
                    Dist[e.dst] = min(Dist[e.dst], Dist[src] + e.weight)
                    $B$[threadID].updateBucket(e.dst, $\lfloor$Dist[e.dst]/$\Delta\rfloor$)
            ▷ While there are still vertices of the current priority left to be processed in
    the thread-local bucket
            while $B$[threadID].currentLocalBucket() is not empty do
                currentLocalBucket = $B$[threadID].currentLocalBucket()
                ▷ Check if the local bucket's size is smaller than the threshold
                if currentLocalBucket.size() < threshold then
                    ▷ Process the local bucket immediately
                    for src : currentLocalBucket do
                        for e : G.getOutEdge[src] do
                            Dist[e.dst] = min(Dist[e.dst], Dist[src] + e.weight)
                            $B$[threadID].updateBucket(e.dst, $\lfloor$Dist[e.dst]/$\Delta\rfloor$)
                else break
```

Figure 3-8: Δ-stepping for single-source shortest paths with the eager bucket update approach and the bucket fusion optimization.

in thousands or even tens of thousands of processing rounds. Because each round requires at least one global synchronization, reducing the number of rounds while maintaining priority ordering can significantly reduce synchronization overhead.

Often in practice, many consecutive rounds process a bucket of the same priority. For example, in Δ-stepping, the priorities of vertices that are higher than the current priority can be lowered by edge relaxations to the current priority in a single round. Thus, the same priority bucket may be processed again in the next round. The process repeats until no new vertices are added to the current bucket. This pattern is common in ordered graph algorithms that use priority coarsening. We observe that rounds processing the same bucket can be fused without violating priority ordering.

Based on this observation, we propose a novel bucket fusion optimization for the eager bucket update approach that allows a thread to execute the next round processing the current bucket without synchronizing with other threads. We illustrate bucket fusion using the Δ-stepping algorithm in Figure 3-8. The same optimization can be applied in other applications, such as weighted breadth-first search (wBFS), A^* search and point-to-point shortest path. This algorithm extends the eager bucket update algorithm (Figure 3-7) by adding a while loop inside each local thread (Figure 3-8, Line 15). The while loop executes if the current local bucket is non-empty. If the current local bucket's size is below a certain threshold, the algorithm immediately processes the current bucket without synchronizing with other threads (Figure 3-8, Line 18). If the current local bucket is large, it will be copied over to the global bucket and distributed across other threads. The threshold is important to avoid creating straggler threads that process too many vertices, leading to load imbalance. The bucket processing logic in the while loop (Figure 3-8, Lines 20–23) is the same as the original processing logic (Figure 3-8, Lines 10–13). This optimization is hard to apply for the lazy approach since a global synchronization is needed before bucket updates.

Bucket fusion is particularly useful for road networks where multiple rounds frequently process the same bucket. For example, bucket fusion reduces the number of rounds by more than 30× for SSSP on the RoadUSA graph, leading to more than 3× speedup by significantly reducing the amount of global synchronization (details in Section 6.2).

3.5 Chapter Summary

In this chapter, we described various performance optimizations for graph traversal and bucketing data structures. We analyzed the tradeoffs between the different optimizations and provided some guidelines for selecting the optimizations depending on the performance bottlenecks of the graph algorithms. Chapter 4 describes the interface for the programmer to combine the different optimizations using the scheduling language. Chapter 5 provides more implementation details for the various optimizations in the GraphIt compiler. Specifically, Section 5.1 formalizes the space of graph traversal optimizations Finally, Chapter 6 evaluates the performances of the various optimizations.

Chapter 4

Language Design

The performance bottlenecks of graph algorithms are diverse and depend on the size and structure of the input graph. No single set of performance optimizations performs well across all graphs and algorithms. Therefore, to achieve high performance across different graphs and graphs, GraphIt introduces a novel design that decouples algorithm specification from optimizations for the graph algorithms. The design is inspired by the Halide programming language [80]'s decoupled design for the image processing domain. By separating the algorithm specifications from the optimizations, GraphIt is the first graph processing framework that allows the programmers to easily navigate the complex performance tradeoff space described in Chapter 3 for optimizations, while providing an easy-to-use high-level programming model for the domain experts. GraphIt provides an algorithm language and a separate scheduling language. The algorithm language can express a variety of ordered and unordered graph algorithms, while exposing opportunities for optimizations. The scheduling language allows the user to try different combinations of optimizations. In this chapter, we describe the design of the algorithm and scheduling language and use PageRankDelta (Figure 4-1) and Δ-stepping(Figure 4-2) to showcase the language.

```
element Vertex end
element Edge end
const edges : edgeset{Edge}(Vertex,Vertex) = load(argv[1]);
const vertices : vertexset{Vertex} = edges.getVertices();
const damp : double = 0.85;
const base_score : double = (1.0 - damp)/vertices.size();
const epsilon : double = 0.1;
const OutDegree : vector{Vertex}(int) = edges.getOutDegrees();
Rank : vector{Vertex}(double) = 0;
DeltaSum : vector{Vertex}(double) = 0.0;
Delta : vector{Vertex}(double) = 1.0/vertices.size();
func updateEdge(src : Vertex, dst : Vertex)
    DeltaSum[dst] += Delta[src]/OutDegree[src];
end
func updateVertexFirstRound(v : Vertex) -> output : bool
    Delta[v] = damp * (DeltaSum[v]) + base_score;
    Rank[v] += Delta[v];
    Delta[v] = Delta[v] - 1.0/vertices.size();
    output = fabs(Delta[v] > epsilon*Rank[v]);
    DeltaSum[v] = 0;
end
func updateVertex(v : Vertex) -> output : bool
   Delta[v] = DeltaSum[v] * damp;
   Rank[v] += Delta[v];
   DeltaSum[v] = 0;
   output = fabs(Delta[v]) > epsilon * Rank[v];
end
func main()
    var V : int = vertices.size();
    var Frontier : vertexset{Vertex} = new vertexset{Vertex}(V);
    for i in 1:maxIters
        #s1# edges.from(frontier).apply(updateEdge);
        if i == 1
            Frontier = vertices.filter(updateVertexFirstRound);
        else
            Frontier = vertices.filter(updateVertex);
        end
    end
end
```

Figure 4-1: GraphIt code for PageRankDelta.

Table 4.1: Algorithm Language API.

Set Operators	**Return Type**	**Description**
`size()`	int	Returns the size of the set.
Vertexset operators		
`filter(func f)`	vertexset	Filters out vertices where `f`(vertex) returns true.
`apply(func f)`	none	Applies `f`(vertex) to every vertex.
Edgeset operators		
`from(vertexset vset)`	edgeset	Filters out edges whose source vertex is in the input vertexset.
`to(vertexset vset)`	edgeset	Filters out edges whose destination vertex is in the input vertexset.
`filter(func f)`	edgeset	Filters out edges where `f`(edge) returns true.
`srcFilter(func f)`	edgeset	Filters out edges where `f`(source) returns true.
`dstFilter(func f)`	edgeset	Filters out edges where `f`(destination) returns true.
`apply(func f)`	none	Applies `f`(source, destination) to every edge.
`applyModified(func f, vector vec, [bool disable_deduplication])`	vertexset	Applies `f`(source, destination) to every edge. Returns a vertexset that contains destination vertices whose entry in the vector vec has been modified in `f`. The programmer can optionally disable deduplication within modified vertices. Deduplication is enabled by default.
`applyUpdatePriority(func f)`	none	Applies `f`(src, dst) to every edge. The `f` function updates priorities of vertices.
Priority Queue Operators		
`new priority_queue( bool coarsen_priority, string priority_dir, vector priority_vector, [Vertex start_vertex])`	priority_queue	The constructor for the priority queue. It specifies a) whether priority coarsening is allowed or not, b) whether higher or lower priority is executed first, c) the vector that is used to compute the priority values, and d) an optional start vertex.
`pq.dequeueReadySet()`	vertexset	Returns a bucket with all the vertices that are currently ready to be processed.
`pq.finished()`	bool	Checks if there is any bucket left to process.
`pq.finishedVertex(Vertex v)`	bool	Checks if a vertex's priority is finalized (finished processing).
`pq.getCurrentPriority()`	priority_type	Returns the priority of the current bucket.
`pq.updatePriorityMin(Vertex v, ValT new_val)`	void	Decreases the value of the priority of the specified vertex `v` to the new_val.
`pq.updatePriorityMax(Vertex v, ValT new_val)`	void	Increases the value of the priority of the specified vertex `v` to the new_val.
`pq.updatePrioritySum(Vertex v, ValT sum_diff, ValType min_threshold)`	void	Adds sum_diff to the priority of the Vertex `v`. The user can specify an optional minimum threshold so that the priority does not go below the threshold.

```
element Vertex end
element Edge end
const edges : edgeset{Edge}(Vertex,Vertex, int)=load(argv[1]);
const dist : vector{Vertex}(int) = INT_MAX;
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    var new_dist : int = dist[src] + weight;
    pq.updatePriorityMin(dst, dist[dst], new_dist);
end

func main()
    var start_vertex : int = atoi(argv[2]);
    dist[start_vertex] = 0;
    pq = new priority_queue
        {Vertex}(int)(true, "lower_first", dist, start_vertex);
    while (pq.finished() == false)
        var bucket : vertexset{Vertex} = pq.dequeueReadySet();
        #s1# edges.from(bucket).applyUpdatePriority(updateEdge);
        delete bucket;
    end
end
```

Figure 4-2: GraphIt algorithm for Δ-stepping for SSSP.

4.1 Algorithm Language

GraphIt aims to capture the algorithm at a high-level with vertex, edge sets, priority queues, and user-defined functions (UDFs). The high-level algorithm specification also enables opportunities for optimizations such as direction optimization, eager bucket update, eager update with bucket fusion, lazy bucket update, and other optimizations. The operators in GraphIt hide low-level implementation details such as atomic synchronization, deduplication, bucket updates, and priority coarsening for both ordered and unordered graph algorithms We designed the GraphIt language as a standalone language instead of an embedded domain-specific language (embedded DSL) because many optimizations require global transformations across multiple data structures and functions. Using a stand-alone language also allows GraphIt to provide simpler syntax involving many different components, such as UDFs and loops. The set-based design in the algorithm language is inspired by the Simit programming language [53] for simulations and the Ligra graph processing framework [91].

4.1.1 Data Model

GraphIt's data model consists of elements, vertexsets and edgesets, and vertex and edge data. These abstract data types and structures help GraphIt achieve physical independence, enabling the compiler to switch between different low-level implementations. The high-level data model also allows GraphIt to avoid analyzing complex general-purpose data structures with pointers.

We illustrate GraphIt's data model with the PageRankDelta example in Figure 4-1. The programmer first defines vertex and edge element types (Vertex and Edge on Lines 1–2 of Figure 4-1). GraphIt supports multiple types of user-defined vertices and edges, which is important for algorithms that work on multiple graphs. After defining element types, the programmer can construct vertexsets and edgesets. Lines 3–4 of Figure 4-1 present the definitions of an edgeset, *edges*, and vertexset, *vertices*. Each element of the edgeset is of Edge type (specified between "{ }"), and the source and destination of the edge is of Vertex type (specified between "( )"). The edgeset declaration supports edges with different types of source and destination vertices (e.g., in a bipartite graph). *vertices* uses the `getVertices` method on the edgeset to obtain the union of the source and destination vertices of the *edges*. Data for vertices and edges are defined as vectors associated with an element type denoted using the { } syntax (Lines 8–11).

4.1.2 Language Constructs and Operators

The language constructs of GraphIt aim to represent the algorithm at a high level, allowing opportunities for edge traversal and vertex data layout optimizations. At the same time, we want to free the programmer from specifying low-level implementation details, such as synchronization, priority-queue implementation, and deduplication logic. To achieve these goals, the algorithm language (presented in Table 4.1) separates the edge processing logic from other components, such as the edge traversal, priority-based ordering, edge filtering (`from`, `to`, `srcFilter`, and `dstFilter`), atomic synchronization, and modified vertex deduplication and tracking logic (`apply` and `applyModified`). Each component can be scheduled with optimizations separately. The compiler has the freedom to fuse together the different components or keep them separated depending on the performance optimizations specified by the user.

In the GraphIt code for PageRankDelta (Figure 4-1), the `from` operator (Line 32) ensures that only edges whose source vertex is in the frontier are traversed, and the `apply` operator uses the `updateEdge` function on the selected edges to compute DeltaSum, corresponding to Lines 8–10 of Algorithm 2-2. This separation enables the compiler to generate complex code for different traversal modes and parallelization optimizations, while inserting appropriate data access and synchronization instructions for the `updateEdge` function. `#s1#` is a label used in the scheduling language (explained in Section 4.2). Lines 34 and 36 of Figure 4-1 compute the updated Delta and Rank values by applying updateVertexFirstRound and updateVertex functions on every vertex. Vertices with Delta greater than epsilon of their Rank are returned as the next frontier, corresponding to Lines 12–23 of Algorithm 2-2. As shown in Table 4.1, GraphIt provides various operators on vertexsets and edgesets to express graph algorithms with different traversal and update logic. The `applyModified` operator tracks which vertices have been updated during the edge traversal and outputs a vertexset containing just those vertices. By default, `applyModified` ensures that each vertex is added only once to the output vertexset. However, the programmer can optionally disable deduplication for algorithms that are guaranteed to insert each vertex only once (e.g., BFS) for better performance.

We demonstrate how GraphIt simplifies the expression of the algorithm by showing Ligra's implementation of the edge update function in Figure 4-3 (note that the 16 lines of Ligra code shown correspond to only three lines in GraphIt's implementation in Fig. 4-1). Ligra requires the programmer to specify edge processing (Lines 8–9, 12–14), edge filtering (Line 16), deduplication and modification tracking (Lines 10 and 15), and synchronization logic (Lines 12–14). GraphIt only requires the programmer to specify the edge processing logic in this case.

GraphIt also supports traditional control flow constructs such as `for`, `while`, and `if` for expressing fixed iteration loops, loops until convergence, and conditional control

```
template <class vertex>
struct PR_F {
  vertex* V;
  double* Delta, *nghSum;
  PR_F(vertex* _V, double* _Delta, double* _nghSum) :
    V(_V), Delta(_Delta), nghSum(_nghSum) {}
  inline bool update(uintE s, uintE d){
    double oldVal = nghSum[d];
    nghSum[d] += Delta[s]/V[s].getOutDegree();
    return oldVal == 0;}
  inline bool updateAtomic (uintE s, uintE d) {
    volatile double oldV, newV;
    do { oldV = nghSum[d]; newV = oldV + Delta[s]/V[s].getOutDegree();
    } while(!CAS(&nghSum[d],oldV,newV));
    return oldV == 0.0;}
  inline bool cond (uintE d) { return cond_true(d); }};
```

Figure 4-3: Ligra's PageRankDelta edge update function, corresponding to Lines 12–14 of Figure 4-1 in GraphIt's PageRankDelta example.

flow. After setting up a new `vertexset` called Frontier, Line 31 in Figure 4-1 uses a `for` loop to iterate the maxIters times. An alternative implementation could use a while loop that iterates until the ranks of all vertices stabilize.

We use Δ-stepping in Figure 4-2 to demonstrate the priority-based operators. The algorithm specification first sets up the `edgeset` data structures (Lines 1–3), and sets the distances to all the vertices in `dist` to INT_MAX to represent ∞ (Line 4). It declares the global priority queue, `pq`, on Line 5. This priority queue can be referenced in user-defined functions and the main function. The user then defines a function, `updateEdge`, that processes each edge (Lines 7–10). In `updateEdge`, the user computes a new distance value, and then updates the priority of the destination vertex using the `updatePriorityMin` operator defined in Table 4.1. In other algorithms, such as k-core, the user can use `updatePrioritySum` when the priority is decremented or incremented by a given value. The `updatePrioritySum` can detect if the change to the priority is a constant, and use this fact to perform more optimizations. The priority update operators, `updatePriorityMin` and `updatePrioritySum`, hide bucket update operations, allowing the compiler to generate different code for lazy and eager bucket update strategies.

In the main function, programmers use the constructor of the priority queue (Figure 4-2 Lines 15–16) to specify algorithmic information, such as the priority ordering, support for priority coarsening, and the direction in which priorities change (documented in Table 4.1). The abstract priority queue hides low-level bucket implementation details and provides a mapping between vertex data and their priorities. The user specifies a `priority_vector` that stores the vertex data values used for computing priorities. In SSSP, we use the `dist` vector and the coarsening parameter (Δ specified using the scheduling language) to perform priority coarsening. The while loop (Figure 4-2 Line 17) processes vertices from a bucket until all buckets are finished processing. In each iteration of the while loop, a new bucket is extracted with `dequeueReadySet` (Figure 4-2 Line 18). The edgeset operator on Line 19 uses the `from` operator to keep only the edges that come out of the vertices in the bucket. Then it uses `applyUpdatePriority` to apply the `updateEdge` function to outgoing edges of the bucket. Label (`#s1#`) is later used by the scheduling language to configure optimization strategies.

4.1.3 Guide for Writing an Algorithm Specification

In this section, we provide a short guide for writing algorithm specifications in GraphIt in addition to the PageRankDelta and Delta Stepping examples described in this chapter. The first step is to define the data structures used in the algorithm. Specifically, the vertex and edge set types should be declared early in the program. The programmer also needs to define the data associated with each vertex using the vectors, such as the rank and the outdegree in the PageRankDelta example.

The next step is to write the graph processing logic. It is important to frame the graph processing steps as operations on the vertex and edge sets. It is easy to utilize GraphIt's vertex and edge set filtering and processing operators. To use the vertex and edge set processing operators in GraphIt, the programmer must also define a series of UDFs that can be passed as arguments.

The UDFs specify the logic to process or filter a single edge or a single vertex. Within a UDF, the user can only access vertices that are passed in as function arguments or global variables. The UDFs used for filtering must return a boolean variable. The UDFs used for processing a vertex or edge with the `apply` operator do not need to return anything.

Finally, the output can be stored in vectors or with just a scalar value. We also provide support for Python Bindings that can convert between python and GraphIt data structures. The user can also use the generated C++ program directly as external functions in C++.

4.2 Scheduling Language

After specifying the algorithm using the language described in Section 4.1, programmers can explore different combinations of optimizations using GraphIt's scheduling language. We designed the scheduling language to be flexible enough to represent the various optimizations outlined in Chapter 3. Additionally, the scheduling language has to be modular so that different optimizations can be combined together to form new optimizations. In this section, we describe the design of the scheduling language functions and demonstrate how they work with PageRankDelta.

We use labels (`#label#`) in algorithm specifications to identify the statements on which optimizations apply. Programmers can assign a label on the left side of a statement and later reference it in the scheduling language. Figure 4-4 shows a simple schedule for the PageRankDelta implementation in Figure 4-1. The programmer adds label **`s1`** to the `edgeset` operation statement. After the `schedule` keyword, the programmer can make a series of calls to scheduling functions.

Compared to Halide's scheduling language, which can only schedule functions, GraphIt's scheduling language is more flexible and can configure optimizations for individual operators within functions, loops, and even data structures. This allows GraphIt to incorporate more program transformations at a finer level and include physical data layout optimizations.

We designed GraphIt's scheduling language functions (shown in Table 4.2) to allow programmers to compose the edge traversal direction, frontier data structure, parallelization, cache, NUMA, vertex data layout, and program structure optimizations discussed in Chapter 3. The `configApplyDirection` function allows programmers to configure directions used for traversal. The programmer can use

```
  ...
  for i in 1:maxIters
    #s1# edges.from(frontier).apply(updateEdge);
    ...
  end
...
schedule:
program->configApplyDirection("s1", "DensePull-SparsePush");
```

Figure 4-4: Scheduling PageRankDelta.

Table 4.2: GraphIt scheduling language functions. The default option for an operator is shown in bold. Optional arguments are shown in [ ]. If the optional direction argument is not specified, the configuration is applied to all relevant directions. We use a default grain size of 256 for parallelization.

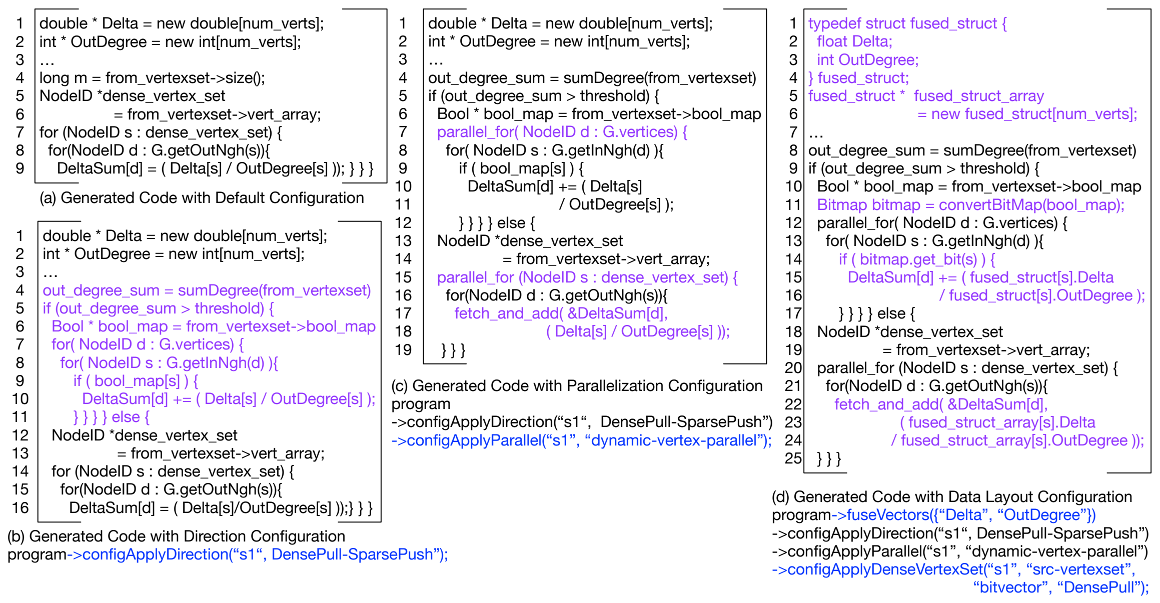
Apply Scheduling Functions	**Descriptions**
`program->configApplyDirection(label, config);`	Config options: **SparsePush**, DensePush, DensePull, DensePull-SparsePush, DensePush-SparsePush
`program->configApplyParallelization (label, config, [grainSize], [direction]);`	Config options: **serial**, dynamic-vertex-parallel, static-vertex-parallel, edge-aware-dynamic-vertex-parallel, edge-parallel
`program->configApplyDenseVertexSet (label, config, [vertexset], [direction])`	Vertexset options: **both**, src-vertexset, dst-vertexset Config Options: **bool-array**, bitvector
`program->configApplyNumSSG(label, config, numSegments, [direction]);`	Config options: **fixed-vertex-count** or edge-aware-vertex-count
`program->configApplyNUMA(label, config, [direction]);`	Config options: **serial**, static-parallel, dynamic-parallel
`program->fuseFields({vect1, vect2, ...})`	Fuses multiple arrays into a single array of structs.
`program->fuseForLoop(label1, label2, fused_label)`	Fuses together multiple loops.
`program->fuseApplyFunctions(label1, label2, fused_func)`	Fuses together two edgeset apply operators. The fused apply operator replaces the first operator.
`program->configApplyPriorityUpdate (label,config);`	Config options: eager_with_fusion, eager_no_fusion, lazy_constant_sum, and **lazy**.
`program-> configApplyPriorityUpdateDelta (label,config);`	Configures the Δ parameter for coarsening the priority range.
`program->configBucketFusionThreshold (label, config);`	Configures the threshold for the bucket fusion optimization.
`program->configNumBuckets (label,config);`	Configures the number of buckets that are materialized for the lazy bucket update approach.

the `configDenseVertexSet` function to switch between bitvector and boolean array for source and destination vertexsets. The `configApplyNumSSG` function configures the number of segmented subgraphs and how the subgraphs are partitioned (fixed-vertex-count and edge-aware-vertex-count). Setting the right number of segments and partitioning configuration allows random accesses to be restricted to a NUMA node or last level cache with balanced load as described in Chapter 3. Moreover, `configApplyNUMA` configures the SSGs to be executed in parallel with static or dynamic NUMA node assignment (static-parallel and dynamic-parallel), ensuring the random memory accesses are restricted to the local NUMA node, while maintaining good parallel scalability. Finally, vertex data vectors can be fused together into an array of structs with `fuseFields`.

To combine different optimizations, the programmer first chooses a direction for traversal. Then the programmer can use the other scheduling functions to pick one option for the parallelization, graph partitioning, NUMA, and dense `vertexset` optimizations for the current direction. The programmer can configure each direction separately using the optional direction argument for hybrid directions (DensePush-SparsePush or DensePull-SparsePush). If no direction argument is specified, then the configuration applies to both directions.

4.2.1 Scheduling PageRankDelta

Figure 4-5 illustrates different schedules for PageRankDelta. Figure 4-5(a) starts with the pseudocode generated from the default schedule that performs a serial SparsePush traversal. Figure 4-5(b) adds a hybrid traversal code that first computes the sum of out-degrees and uses it to determine whether to perform a DensePull or a SparsePush traversal. This allows the implementation to pick the traversal mode that minimizes the number of edges that need to be traversed, improving work-efficiency. Figure 4-5(c) adds dynamic-vertex-parallelism to both directions in the generated code by parallelizing the loops and inserting synchronization codes for SparsePush. Figure 4-5(d) adds the vertex data layout and bitvector optimizations. Fusing together the vectors Delta and OutDegree with the `fuseFields` function improves spatial locality of memory accesses since the two vectors are always accessed together. This optimization changes the declaration and access points for the arrays. Finally, for the DensePull



```
double * Delta = new double[num_verts];
int * OutDegree = new int[num_verts];
...
long m = from_vertexset->size();
NodeID *dense_vertex_set
              = from_vertexset->vert_array;
for (NodeID s : dense_vertex_set) {
  for(NodeID d : G.getOutNgh(s)){
    DeltaSum[d] = ( Delta[s] / OutDegree[s] ); } } }
```

(a) Generated Code with Default Configuration

```
double * Delta = new double[num_verts];
int * OutDegree = new int[num_verts];
...
out_degree_sum = sumDegree(from_vertexset)
if (out_degree_sum > threshold) {
  Bool * bool_map = from_vertexset->bool_map
  for( NodeID d : G.vertices) {
    for( NodeID s : G.getInNgh(d) ){
      if ( bool_map[s] ) {
        DeltaSum[d] += ( Delta[s] / OutDegree[s] );
      } } } } else {
  NodeID *dense_vertex_set
              = from_vertexset->vert_array;
  for (NodeID s : dense_vertex_set) {
    for(NodeID d : G.getOutNgh(s)){
      DeltaSum[d] = ( Delta[s]/OutDegree[s] );} } }
```

(b) Generated Code with Direction Configuration
program->configApplyDirection("s1", DensePull-SparsePush");

```
double * Delta = new double[num_verts];
int * OutDegree = new int[num_verts];
...
out_degree_sum = sumDegree(from_vertexset)
if (out_degree_sum > threshold) {
  Bool * bool_map = from_vertexset->bool_map
  parallel_for( NodeID d : G.vertices) {
    for( NodeID s : G.getInNgh(d) ){
      if ( bool_map[s] ) {
        DeltaSum[d] += ( Delta[s]
                    / OutDegree[s] );
      } } } } else {
  NodeID *dense_vertex_set
              = from_vertexset->vert_array;
  parallel_for (NodeID s : dense_vertex_set) {
    for(NodeID d : G.getOutNgh(s)){
      fetch_and_add( &DeltaSum[d],
                    ( Delta[s] / OutDegree[s] ));
} } }
```

(c) Generated Code with Parallelization Configuration
program
->configApplyDirection("s1", DensePull-SparsePush")
->configApplyParallel("s1", "dynamic-vertex-parallel");

```
typedef struct fused_struct {
  float Delta;
  int OutDegree;
} fused_struct;
fused_struct * fused_struct_array
                    = new fused_struct[num_verts];
...
out_degree_sum = sumDegree(from_vertexset)
if (out_degree_sum > threshold) {
  Bool * bool_map = from_vertexset->bool_map
  Bitmap bitmap = convertBitMap(bool_map);
  parallel_for( NodeID d : G.vertices) {
    for( NodeID s : G.getInNgh(d) ){
      if ( bitmap.get_bit(s) ) {
        DeltaSum[d] += ( fused_struct[s].Delta
                    / fused_struct[s].OutDegree );
  } } } } else {
  NodeID *dense_vertex_set
              = from_vertexset->vert_array;
  parallel_for (NodeID s : dense_vertex_set) {
    for(NodeID d : G.getOutNgh(s)){
      fetch_and_add( &DeltaSum[d],
                ( fused_struct_array[s].Delta
              / fused_struct_array[s].OutDegree ));
} } }
```

(d) Generated Code with Data Layout Configuration
program->fuseVectors({"Delta", "OutDegree"})
->configApplyDirection("s1", DensePull-SparsePush")
->configApplyParallel("s1", "dynamic-vertex-parallel")
->configApplyDenseVertexSet("s1", "src-vertexset",
"bitvector", "DensePull");

Figure 4-5: Each subfigure shows pseudocode generated from applying the schedule in the caption to the GraphIt PageRankDelta code with labels from Figure 4-1 and Figure 4-4. The options in the caption highlighted in blue are newly added scheduling commands relative to the previous subfigure and the code highlighted in purple is pseudocode updated due to the new schedules.

direction, the source `vertexset` specified in `from` can be dynamically compressed into a bitvector to reduce the working set size, further improving spatial locality.

4.2.2 Scheduling Program Structure Optimizations

To support program structure optimizations, we introduce scoped labels, which allow labels to function even after complex program transformations, and scheduling functions for fusing together loops and edgeset `apply` operators. Figure 4-6 shows two iterative edgeset `apply` operators (Lines 2 and 5) that can be fused together into a single iterative edgeset `apply` operator. GraphIt first performs loop fusion, creating a new loop (**l3**), and destroying the two old loops (**l1** and **l2**). Now, it would be difficult if we wanted to schedule the first edgeset `apply` operator in the **l3** loop as the original loops **l1** and **l2** have been removed from the program. Since both edgeset `apply` operators have **s1** as their labels, it is difficult to identify them individually. To address this, we introduce scoping to the labels. The two `apply` operators will obtain the labels **l1:s1** and **l2:s1**, respectively.

```
#l1# for i in 1:10
    #s1# edges.apply(func1);
end
#l2# for i in 1:10
    #s1# edges.apply(func2);
end
schedule:
program->fuseForLoop("l1", "l2", "l3")
->fuseApplyFunctions("l3:l1:s1", "l3:l2:s1", "fusedFunc")
->configApplyDirection("l3:l1:s1", "DensePull");
```

Figure 4-6: GraphIt loop and function fusion

We also need a name node, which enforces a named scope for the label. Loops **l1** and **l2** are replaced with the name nodes with labels **l1** and **l2**, respectively. The resulting pseudocode is displayed in Figure 4-7. This enables the user to reference the first edgeset `apply` as **l3:l1:s1** and the second edgeset `apply` as **l3:l2:s1**. After the loops are fused together, we can use `fuseApplyFunctions` to create a new edgeset `apply` to replace the **l3:l1:s1** statement, which can be further configured (Figure 4-8). The new edgeset `apply` function, `fusedFunc`, concatenates the statements in the original functions, `func1` and `func2`. In Section 6.2, we demonstrate that the fusion of multiple iterative kernels with similar traversal patterns (eigenvector centrality and PageRank), and the vertex data vectors they access boosts the performance of the application by up to 60% as shown in Section 6.2.5.

```
#l3# for i in 1:10
   #l1# namenode
       #s1# edges.apply(func1);
   end
   #l2# namenode
       #s1# edges.apply(func2);
   end
end
```

Figure 4-7: Pseudocode after loop fusion

```
#l3# for i in 1:10
   #l1# namenode
       #s1# edges.apply(fused_func);
   end
end
```

Figure 4-8: Pseudocode after function fusion

4.2.3 Scheduling Δ-stepping

The scheduling language allows users to specify different optimization strategies for the ordered graph algorithms as well. We extend the scheduling language of GraphIt

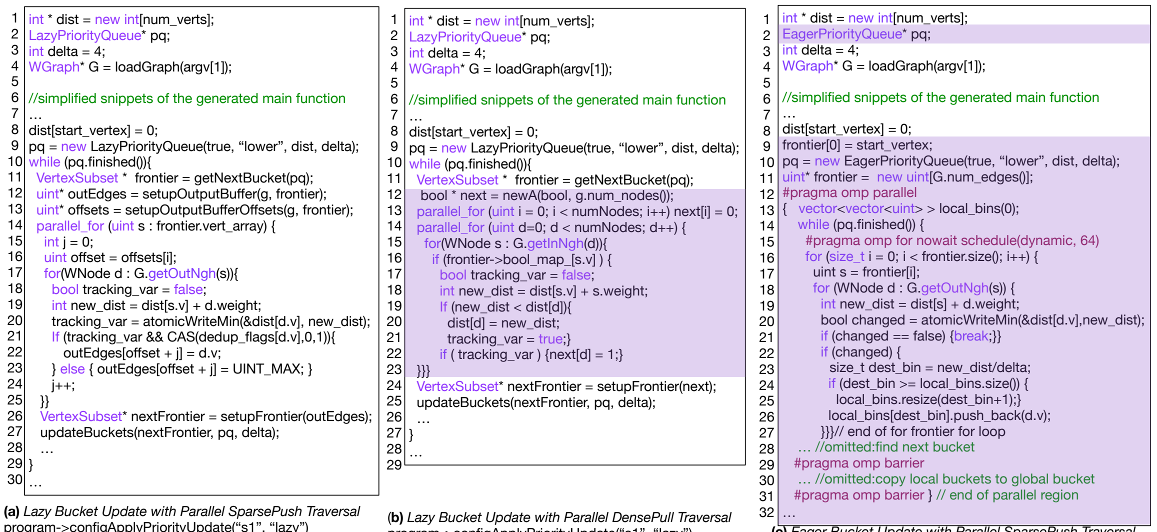
```
  ...
    while (pq.finished() == false)
        var bucket : vertexsubset = pq.dequeueReadySet();
        #s1# edges.from(bucket).applyUpdatePriority(updateEdge);
        delete bucket;
    end
...
schedule:
program->configApplyPriorityUpdate("s1", "lazy")
->configApplyPriorityUpdateDelta("s1", "4")
->configApplyDirection("s1", "SparsePush")
->configApplyParallelization("s1","dynamic-vertex-parallel");
```

Figure 4-9: GraphIt scheduling specification for Δ-stepping.

with new optimizations, such as the eager and lazy bucket update strategies, and eager bucket fusion optimizations. Users can also tune other parameters, such as the coarsening factor for priority coarsening. The scheduling API extensions are listed in Table 4.2.

Figure 4-9 illustrates a set of schedules for Δ-stepping. GraphIt uses labels (`#label#`) to identify the algorithm language statements for which the scheduling language commands are applied. The programmer adds the label **s1** to the `edgeset applyUpdatePriority` statement. After the `schedule` keyword, the programmer calls the scheduling functions. The `configApplyPriorityUpdate` function allows the programmer to use the lazy bucket update optimization. The programmer can use the original GraphIt scheduling language to configure the direction of edge traversal (`configApplyDirection`) and the load balance strategies (`configApplyParallelization`). Direction optimizations can be combined with lazy priority update schedules. Another API, `configApplyUpdateDelta`, is used to set the delta for priority coarsening.

Users can change the schedules to generate code with different combinations of optimizations as illustrated in Figure 4-10. Figure 4-10(a) presents the code generated by combining the lazy bucket update strategy and other edge traversal optimizations from the GraphIt scheduling language. The scheduling function `configApplyDirection` configures the data layout of the frontier and direction of the edge traversal (`SparsePush` indicates the sparse frontier and push direction). Figure 4-10(b) displays the code generated when we combine a different traversal direction (`DensePull`) with the lazy bucketing strategy. Figure 4-10(c) reveals the code generated with the eager bucket update strategy. Code generation is explained in Section 5.2.



```
int * dist = new int[num_verts];
LazyPriorityQueue* pq;
int delta = 4;
WGraph* G = loadGraph(argv[1]);

//simplified snippets of the generated main function
…
dist[start_vertex] = 0;
pq = new LazyPriorityQueue(true, "lower", dist, delta);
while (pq.finished()){
  VertexSubset *  frontier = getNextBucket(pq);
  uint* outEdges = setupOutputBuffer(g, frontier);
  uint* offsets = setupOutputBufferOffsets(g, frontier);
  parallel_for (uint s : frontier.vert_array) {
    int j = 0;
    uint offset = offsets[i];
    for(WNode d : G.getOutNgh(s)){
      bool tracking_var = false;
      int new_dist = dist[s.v] + d.weight;
      tracking_var = atomicWriteMin(&dist[d.v], new_dist);
      If (tracking_var && CAS(dedup_flags[d.v],0,1)){
        outEdges[offset + j] = d.v;
      } else { outEdges[offset + j] = UINT_MAX; }
      j++;
   }}
   VertexSubset* nextFrontier = setupFrontier(outEdges);
   updateBuckets(nextFrontier, pq, delta);
   …
}
…
```

(a) *Lazy Bucket Update with Parallel SparsePush Traversal*
program->configApplyPriorityUpdate("s1", "lazy")
->configApplyPriorityUpdateDelta("s1", 4)
->configApplyDirection("s1", SparsePush")
->configApplyParallelization("s1", "dynamic-vertex-parallel");

```
int * dist = new int[num_verts];
LazyPriorityQueue* pq;
int delta = 4;
WGraph* G = loadGraph(argv[1]);

//simplified snippets of the generated main function
…
dist[start_vertex] = 0;
pq = new LazyPriorityQueue(true, "lower", dist, delta);
while (pq.finished()){
  VertexSubset *  frontier = getNextBucket(pq);
   bool * next = newA(bool, g.num_nodes());
  parallel_for (uint i = 0; i < numNodes; i++) next[i] = 0;
  parallel_for (uint d=0; d < numNodes; d++) {
    for(WNode s : G.getInNgh(d)){
      if (frontier->bool_map_[s.v] ) {
        bool tracking_var = false;
        int new_dist = dist[s.v] + s.weight;
        If (new_dist < dist[d]){
          dist[d] = new_dist;
          tracking_var = true;}
        if ( tracking_var ) {next[d] = 1;}
  }}}
  VertexSubset* nextFrontier = setupFrontier(next);
  updateBuckets(nextFrontier, pq, delta);
  …
}
…
```

(b) *Lazy Bucket Update with Parallel DensePull Traversal*
program->configApplyPriorityUpdate("s1", "lazy")
->configApplyPriorityUpdateDelta("s1", 4)
->configApplyDirection("s1", DensePull")
->configApplyParallelization("s1", "dynamic-vertex-parallel");

```
int * dist = new int[num_verts];
EagerPriorityQueue* pq;
int delta = 4;
WGraph* G = loadGraph(argv[1]);

//simplified snippets of the generated main function
…
dist[start_vertex] = 0;
frontier[0] = start_vertex;
pq = new EagerPriorityQueue(true, "lower", dist, delta);
uint* frontier =  new uint[G.num_edges()];
#pragma omp parallel
{   vector<vector<uint> > local_bins(0);
    while (pq.finished()) {
      #pragma omp for nowait schedule(dynamic, 64)
      for (size_t i = 0; i < frontier.size(); i++) {
        uint s = frontier[i];
        for (WNode d : G.getOutNgh(s)) {
          int new_dist = dist[s] + d.weight;
          bool changed = atomicWriteMin(&dist[d.v],new_dist);
          if (changed == false) {break;}}
          if (changed) {
            size_t dest_bin = new_dist/delta;
            if (dest_bin >= local_bins.size()) {
              local_bins.resize(dest_bin+1);}
            local_bins[dest_bin].push_back(d.v);
          }}}// end of for frontier for loop
    … //omitted:find next bucket
   #pragma omp barrier
    … //omitted:copy local buckets to global bucket
   #pragma omp barrier } // end of parallel region
…
```

(c) *Eager Bucket Update with Parallel SparsePush Traversal*
program->configApplyPriorityUpdate("s1", "eager")
->configApplyPriorityUpdateDelta("s1", 4)
->configApplyDirection("s1", SparsePush")
->configApplyParallelization("s1", "dynamic-vertex-parallel");

Figure 4-10: Simplified generated C++ code for Δ-stepping for single-source shortest paths (SSSP) with different schedules. Changes in the schedules for (b) and (c) compared to (a) are highlighted in blue. Changes in the generated code are highlighted with a purple background. The `parallel_for` is translated to `cilk_for` or `#pragma omp parallel for`. Struct `WNode` has two fields, `v` and `weight`, where `v` stores the vertex ID and `weight` stores the edge weight.

4.2.4 Guide for Manually Tuning the Schedules

While the user can use the autotuner in GraphIt to determine the best schedules, we still provide some general guidelines for manually selecting a set of schedules. The full set of schedules we used for different applications is listed in Figure 6.3.

The first step is to select a direction from SparsePush, DensePush, DensePull, SparsePush-DensePull with `configApplyDirection`. In general, the direction is often decided by the size of the frontiers. For large social networks that generate frontiers with varying sizes, SparsePush-DensePull is usually a good choice. When the frontier size is fixed for the algorithm, it is often the best strategy to pick just one direction. For PageRank, DensePull is always the best direction because the frontiers always contain all of the vertices. For road networks, SparsePush often is the best direction because the frontiers are usually quite small.

The next step is to select a parallelization strategy with `configApplyParallelization`. Usually, the dynamic-vertex-parallel option is the best. For road networks, it is sometimes better to switch to the static-vertex-parallel option. Edge-aware-dynamic-vertex-parallel is only better for PageRank and Collaborative Filtering in some cases, where the frontiers are large and there are high degrees of parallelism.

The user must select a layout for the dense vertex set. This can be done with the `configApplyDenseVertexset` API. If a DensePull direction is used, then you can potentially use the bitvector option for the vertexset when the graph has a large number of vertices (currently only working for the pull direction).

If a DensePull direction is used, then one can use `configNumSSG` (currently only working for the pull direction) to partition the graph for cache efficiency. This is mostly useful for applications that spend considerable time processing all the edges (PageRank, PageRankDelta, and Collaborative Filtering) on large social networks. This optimization is usually bad for applications that only touch a subset of vertices (BFS, SSSP, and betweenness centrality). Calculations for the number of segments is based on the LLC size.

The user can also use `fuseFields` to fuse fields that are accessed together into an array of structures to reduce the number of random accesses. This schedule is only needed for PageRankDelta so far.

For ordered graph algorithms, such as Δ-stepping, it is important to determine whether to use an eager or lazy bucket update strategy. If there are a large number of redundant bucket updates in each round, such as k-core, then the lazy update strategy is a good option. In contrast, if only a few redundant updates are needed per round, then the eager update strategy is more efficient. Most of the ordered graph algorithms can benefit from the eager bucket fusion optimization introduced in Section 5.2.6. However, it is the most effective on the road networks, where the synchronization overhead is high.

Overall, the tuning of the schedules should be a dynamic and iterative process. Many optimization ideas that are expected to work might not actually generate high performance implementatioins. It is important for the programmers to quickly measure the performance and make hypotheses to guide the next steps. This process should be repeated until a high-performance schedule is found.

4.3 Chapter Summary

In this chapter, we described the design of the algorithm and the scheduling language of GraphIt. The decoupling allows the programmer to navigate a complex tradeoff space for performance optimizations, while providing a user-friendly programming model. We demonstrated how GraphIt can be used to implement and optimize PageRankDelta (an unordered graph algorithm) and Δ-stepping (an ordered graph algorithm). In Chapter 5, we describe the implementation of the GraphIt compiler to demonstrate how the compiler generates high-performance implementations based on the algorithm and scheduling specifications.

Chapter 5

Compiler Design and Implementation

Unlike many graph processing libraries, programs in GraphIt are analyzed, optimized, and code generated using a full compiler stack. In this chapter, we focus on the scheduling representation, program analyses and transformations, code generation algorithms, and optimized runtime libraries that GraphIt leverages to produce high-performance C++ code based on the high-level algorithm and optimization specifications described in Chapter 3. GraphIt uses the novel graph iteration space to represent the various graph traversal optimizations specified with the scheduling language, such as the traversal direction and parallelization optimizations. The compiler uses the program analyses and transformations to ensure the correctness and efficiency of the generated programs. The code generation algorithm generates high-performance C++ programs with optimized edge traversals and UDFs based on the scheduling representations. Finally, the runtime libraries provide a series of optimized low-level routines, such as prefix sum and the buckets for maintaining the priority-based ordering, which can be used by the generated program.

5.1 Scheduling Representation

The schedules for an optimized PageRankDelta implementation become even more complex than those shown in Figure 4-5(d) as we further combine NUMA and cache optimizations. It is challenging to reason regarding the validity of all the possible

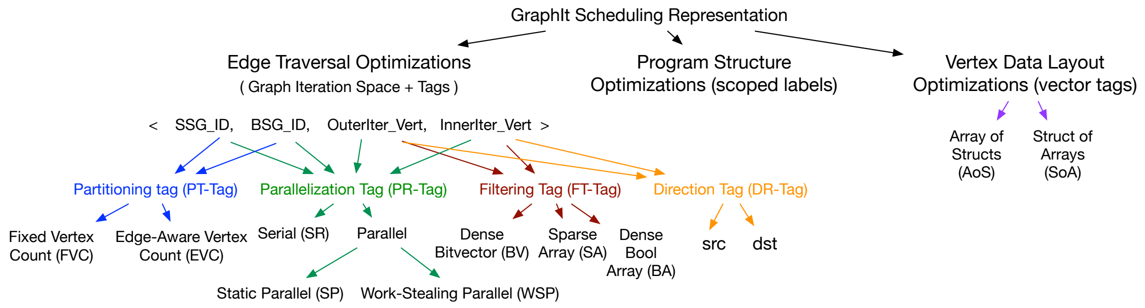

GraphIt Scheduling Representation
Edge Traversal Optimizations
(Graph Iteration Space + Tags)
Program Structure Optimizations (scoped labels)
Vertex Data Layout Optimizations (vector tags)
Array of Structs (AoS)
Struct of Arrays (SoA)
< SSG_ID, BSG_ID, OuterIter_Vert, InnerIter_Vert >
Partitioning tag (PT-Tag)
Parallelization Tag (PR-Tag)
Filtering Tag (FT-Tag)
Direction Tag (DR-Tag)
Fixed Vertex Count (FVC)
Edge-Aware Vertex Count (EVC)
Serial (SR)
Parallel
Dense Bitvector (BV)
Sparse Array (SA)
Dense Bool Array (BA)
src
dst
Static Parallel (SP)
Work-Stealing Parallel (WSP)

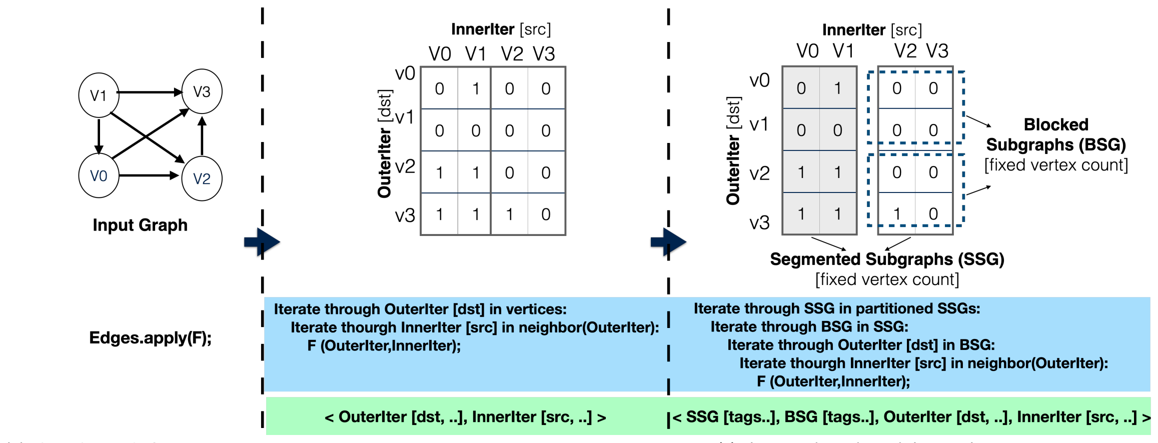
Figure 5-1: GraphIt's scheduling representation for edge traversal, vertex data layout, and program structure optimizations. The tags of the graph iteration space represent the direction and performance optimization choices for each vertex data vector and each dimension of the graph iteration space.

combinations of optimizations and to generate code for them. GraphIt relies on multiple scheduling representations, specifically the graph iteration space, the vertex data vector tags, and the scoped labels, to model combinations of edge traversal, vertex data layout, and program structure optimizations. Figure 5-1 presents the full space of optimizations.

5.1.1 Graph Iteration Space

Motivation. The graph iteration space is an abstract model for edge traversals that represents the edge traversal optimizations specified by the scheduling commands in Table 4.2. The model simplifies the design of the compiler by representing different combinations of optimizations as multi-dimensional vectors. This representation enables the compiler to easily combine different optimizations, reason about validity through dependence analysis, and generate nested loop traversal code. The graph iteration space also defines the space of edge traversal optimizations supported by GraphIt, revealing new combinations of optimizations that have not been explored in prior work.

The concept of graph iteration space is inspired by the traditional iteration spaces in dense nested loops [105, 76]. The dense iteration space represents a nested loop as a multi-dimensional vector, where each dimension represents one level of the loop. Graph Iteration Space extends the dense iteration space concept to the sparse nested


V1
V3
V0
V2
Input Graph
Edges.apply(F);
InnerIter [src]
V0 V1 V2 V3
OuterIter [dst]
v0
v1
v2
v3
Blocked Subgraphs (BSG) [fixed vertex count]
Segmented Subgraphs (SSG) [fixed vertex count]

```
Iterate through OuterIter [dst] in vertices:
    Iterate thourgh InnerIter [src] in neighbor(OuterIter):
        F (OuterIter,InnerIter);
```

< OuterIter [dst, ..], InnerIter [src, ..] >

```
Iterate through SSG in partitioned SSGs:
    Iterate through BSG in SSG:
        Iterate through OuterIter [dst] in BSG:
            Iterate thourgh InnerIter [src] in neighbor(OuterIter):
                F (OuterIter,InnerIter);
```

< SSG [tags..], BSG [tags..], OuterIter [dst, ..], InnerIter [src, ..] >

(a) Algorithm code for traversing Edges and apply function F to every edge

(b) The pseudo code and the graph iteration space for traversing Edges in pull direction using only two dimensions

(c) The pseudo code and the graph iteration space for traversing Edges in pull direction using all four dimensions

Figure 5-2: Representing an edge traversal as nested loops and corresponding graph iteration spaces. Subfigure (c) illustrates the four dimensions for the graph iteration space, assuming using pull direction and the fixed vertex count partitioning strategy for both segmented subgraph (SSG) and blocked subgraphh (BSG) dimensions.

$$\left\{ \langle\, S \text{ [tags]},\ B \text{ [tags]},\ O \text{ [tags]},\ I \text{ [tags]} \,\rangle \;\middle|\; \begin{array}{l} \text{src\_set} = \text{filtered } src \text{ vertexset of } \mathbf{F},\ \text{dst\_set} = \text{filtered} \\ dst \text{ vertexset of } \mathbf{F} \\ O \in \text{src\_set} \wedge I \in \text{dst\_set or } O \in \text{dst\_set} \wedge I \in \text{src\_set} \\ \text{ssg\_set} = \text{subgraphs created by segmenting the graph} \\ \text{based on InnerIter} \\ \text{bsg\_set} = \text{subgraphs created by blocking the graph based} \\ \text{on OuterIter} \\ S \text{ (Segmented Subgraph ID)} \in \text{ssg\_set} \\ B \text{ (Blocked Subgraph ID)} \in \text{bsg\_set} \\ \langle\, O,\ I \,\rangle \in \text{edges within the subgraph } (B \text{ or } S) \end{array} \right\}$$

Figure 5-3: Definition of the graph iteration space with four dimensions. S, B, O, and I are abbreviations for segmented subgraph (SSG), blocked subgraph (BSG), OuterIter, and InnerIter.

loops for edge traversals.

Definition. We assume that we have an operation that traverses the edges and applies an UDF, **F**, on an edgeset Edges as shown in Figure 5-2(a). A graph iteration space defines the set of directed edges on which **F** is applied and the strategy of traversing the edges. First, we represent the graph as an adjacency matrix, where a column in a row has a value of one if the column represents a neighbor of the current row (the top part of Figure 5-2(b)). With this representation, we can traverse through all edges using dense two-level nested for loops that iterate through every row and every column. The traversal can be viewed as a traditional 2-D iteration space. Unlike the dense iteration space, the edge traversal only happens when an edge exists from the source to the destination. Thus, we can eliminate unnecessary traversals and make the loops sparse by iterating only through columns with nonzero values in each row (the blue part in Figure 5-2(b)). We define the row iterator variable as *OuterIter*, and the column iterator variable as *InnerIter*. The green part of Figure 5-2(b) indicates that a 2-D graph iteration space vector is used to represent this two-level nested traversal. The two-level nested for loops can be further blocked and segmented into up to four dimensions, as illustrated in Figure 5-2(c). The dimensions of the graph iteration space encode the nesting level of the edge traversal, and the tags for each dimension specify the strategy used to iterate through that dimension. We provide more details of the graph iteration space below.

Graph Iteration Space Dimensions. The graph iteration space in GraphIt uses four dimensions, defined in Figure 5-3 and illustrated in Figure 5-2. The dimensions are $\langle$ SSG, BSG, OuterIter, InnerIter $\rangle$ and are abbreviated as $\langle$ S, B, O, I $\rangle$. Unused dimensions are marked with $\perp$.

OuterIter (O) and InnerIter (I) in Figure 5-3 are the vertex IDs of an edge (Figure 5-2(b)). The ranges of O and I dimensions depend on the direction. For the push direction, O is in the filtered source vertexset (src_set) and I is in the filtered destination vertexset (dst_set). For the pull direction, O is in the dst_set and I is in the src_set. The OuterIter dimension sequentially accesses vertices, while the InnerIter dimension has a random access pattern for both the push and pull directions becuse the neighbor vertex IDs are not sequential. The edge (O, I) is in the edgeset of the subgraph identified by BSG and SSG.

The BSG (Blocked Subgraph ID) dimension identifies a Blocked Subgraph (BSG) in the Blocked Subgraphs Set (bsg_set). The bsg_set is created by partitioning the graph by the OuterIter dimension as illustrated in the top part of Figure 5-2(c). This partitioning transforms the loops that traverse the edges without changing the graph data structure. The graph can be partitioned with a grain size on the number of OuterIter vertices or on the total number of edges per BSG, depending on the schedule. This dimension controls the different strategies for parallelization optimizations.

The SSG (Segmented Subgraph ID) identifies a Segmented Subgraph (SSG) in the

< S [PR-tag, (PT-Tag, count)], B [PR-tag, (PT-Tag, count)], O [DR-Tag, PR-Tag, FT-Tag], I [DR-Tag, PR-Tag, FT-Tag] >

Figure 5-4: Graph Iteration Space Tags: Direction Tags (DR-Tag), Partitioning Tags (PT-Tag), Parallelization Tags (PR-Tag), and Filtering Tags (FT-Tag) (explained in Figure 5-1) specify direction and optimization strategy for each dimension, and are shown in square brackets next to each dimension.

Table 5.1: Mapping between GraphIt's scheduling language functions to the relevant dimensions and tags (highlighted in bold) of the graph iteration space.

Apply Scheduling Functions	**Graph Iteration Space Dimensions and Tags Configured**
`program->configApplyDirection (label, config);`	$\langle$ S [tags], B [tags], $\boldsymbol{O}$ **[direction tag, filtering tag]**, $\boldsymbol{I}$ **[direction tag, filtering tag]** $\rangle$. Note, for hybrid directions (e.g. DensePull-SparsePush), two graph iteration space vectors are created, one for each direction.
`program->configApplyParallelization (label, config, [grainSize], [direction]);`	$\langle$ S [tags], $\boldsymbol{B}$ **[partitioning tag, parallelization tag]**, O [tags], I [tags] $\rangle$
`program->configApplyDenseVertexSet (label, config, [vertexset], [direction])`	$\langle$ S [tags], B [tags], $\boldsymbol{O}$ **[filtering tag]**, $\boldsymbol{I}$ **[filtering tag]** $\rangle$
`program->configApplyNumSSG(label, config, numSegments, [direction]);`	$\langle$ $\boldsymbol{S}$ **[partitioning tag]**, B [tags], O [tags], I [tags] $\rangle$
`program->configApplyNUMA(label, config, [direction]);`	$\langle$ $\boldsymbol{S}$ **[parallelization tag]**, B [tags], O [tags], I [tags] $\rangle$

Segmented Subgraphs Set (ssg_set). The ssg_set is created by partitioning the graph by the InnerIter dimension as demonstrated in the top part of Figure 5-2(c). The partitioning transforms both the graph data structure and the loops that traverse the edges. Details of the partitioning scheme are described in Section 3.2.3. This dimension controls the range of random accesses, enabling cache and NUMA optimizations. The ordering of the dimensions ensures that the graph is segmented into SSGs before each SSG is blocked into BSGs.

Graph Iteration Space Tags. Each dimension is annotated with tags to specify the direction and optimization strategies (Figure 5-1 illustrates the tags in GraphIt). There are four types of tags: Direction Tags (DR-Tag), Partitioning Tags (PT-Tag), Parallelization Tags (PR-Tag), and Filtering Tags (FT-Tag). We show tags for each dimension within square brackets in Figure 5-4. Table 5.1 shows the mapping between

Table 5.2: The schedules applied for PageRankDelta and the generated graph iteration space vectors with tags, following the examples in Figure 4-5. Additional scheduling commands are added from row to row and the affected dimensions and tags in the graph iteration space are highlighted in bold. ⊥ is an unused dimension. Note that configApplyNumSSG uses an integer parameter (X) which is dependent on the data and hardware system.

PageRankDelta Schedules and the corresponding Graph Iteration Space
`program->configApplyDirection("s1", "SparsePush");`
⟨ ⊥, ⊥, *O* [*src*, Serial, SparseArray], *I* [*dst*, Serial] ⟩
`program->configApplyDirection("s1","DensePull-SparsePush");`
runtime decision between two graph iteration space vectors ⟨ ⊥, ⊥, ***O*** [***dst***, **Serial**], ***I*** [***src***, **Serial**, **Dense Bool Array**] ⟩ and ⟨ ⊥, ⊥, *O* [*src*, Serial, SparseArray], *I* [*dst*, Serial] ⟩

`program->configApplyDirection("s1","DensePull-SparsePush");`
`program->configApplyParallel("s1","dynamic-vertex-parallel");`

runtime decision between two graph iteration space vectors ⟨ ⊥, ***B*** [**Work-Stealing Parallel, (Fixed Vertex Count, 1024)**], *O* [*dst*, Serial], *I* [*src*, Serial, BA] ⟩ and ⟨ ⊥, ***B*** [**Work-Stealing Parallel, (Fixed Vertex Count, 1024)**], *O* [*src*, Serial, SparseArray], *I* [*dst*, Serial] ⟩

`program->configApplyDirection("s1","DensePull-SparsePush");`
`program->configApplyParallel("s1","dynamic-vertex-parallel");`
`program->configApplyDenseVertexSet("s1","src-vertexset", bitvector","DensePull");`

runtime decision between two graph iteration space vectors ⟨ ⊥, *B* [Work-Stealing Parallel, (Fixed Vertex Count, 1024)], *O* [*dst*, Serial], *I* [*src*, Serial, **BitVector**] ⟩ and ⟨ ⊥, *B* [Work-Stealing Parallel, (Fixed Vertex Count, 1024)], *O* [*src*, Serial, SparseArray], *I* [*dst*, Serial] ⟩

`program->configApplyDirection("s1","DensePull-SparsePush");`
`program->configApplyParallel("s1","dynamic-vertex-parallel");`
`program->configApplyDenseVertexSet("s1","src-vertexset", "bitvector","DensePull");`
`program->configApplyNumSSG("s1","fixed-vertex-count",X, "DensePull");`

runtime decision between two graph iteration space vectors ⟨ ***S*** [**Serial**, (**Fixed Vertex Count, num_vert / X**)], *B* [Work-Stealing Parallel, (Fixed Vertex Count, 1024)], *O* [*dst*, Serial], *I* [*src*, Serial, BitVector] ⟩ and ⟨ ⊥, *B* [Work-Stealing Parallel, (Fixed Vertex Count, 1024)], *O* [*src*, Serial, SparseArray], *I* [*dst*, Serial] ⟩

scheduling language commands from Section 4.2 and the corresponding graph iteration space vector and tags.

Direction Tags specify whether the traversal is in push or pull direction. In the push direction, the OuterIter is tagged as *src* and InnerIter tagged as *dst*; in the pull direction, the tags are reversed. Partitioning Tags specify the strategy used for partitioning the SSG or BSG dimensions. For example, the default fixed vertex count (FVC) partitioning strategy will partition the graph based on a fixed number of InnerIter or OuterIter vertices as shown in Figure 5-2(c). Depending on the input, this scheme may lead to an unbalanced number of edges in each SSG or BSG. Alternatively, the edge-aware vertex count (EVC) scheme partitions each subgraph with a different number of InnerIter or OuterIter vertices to ensure each subgraph has a similar number of edges. The EVC tag is used when the users specify the edge-aware-dynamic-vertex-parallel option with `configApplyParalllelization` or the edge-aware-vertex-count option with `configApplyNumSSG`.

Parallelization Tags control whether to iterate through the dimension using serial (SR), static-partitioned parallel (SP), or dynamic work-stealing parallel (WSP) execution strategies. The PR-Tag for the BSG dimension controls the parallelization strategy across different BSGs within a SSG. Tagging the SSG dimension as parallel enables NUMA optimizations by executing multiple SSGs in different sockets in parallel. If work-stealing is enabled, threads on one socket can steal unprocessed SSGs from another socket to improve load balance.

Filtering Tags on the OuterIter and InnerIter dimensions control the underlying data structure. Filtering is implemented with sparse arrays (SA), dense boolean arrays (BA), or dense bitvectors (BV). The sparse arrays contain all of the vertices that pass the filtering, while the dense boolean arrays or the bitvectors set the value to true or the bit to one for each vertex that passes the filtering.

Graph Iteration Spaces for PageRankDelta. Table 5.2 continues to use PageRankDelta as an example to illustrate how scheduling language commands generate graph iteration space vectors and tags. The first row shows that the SparsePush schedule maps to a graph iteration space vector with only two dimensions used ($\perp$ means the dimension is unused). The direction tags for the OuterIter and InnerIter dimensions, *src* and *dst*, indicate that this graph iteration space is for the push direction. Going from SparsePush to DensePull-SparsePush creates a new graph iteration space vector for the pull direction. A runtime threshold on the size of the source `vertexset` is used to determine which vector is executed. The `configApplyParallelization` function configures the BSG dimension with work-stealing parallelism (WSP) and uses the default 1024 grainsize. The fourth row demonstrates that `configDenseVertexSet` sets the filtering tag for the innerIter dimension to bitvector (BV) in the graph iteration space vector for the pull direction. Finally, `configNumSSG` sets up the SSG dimension to partition the graph for cache locality. In the fixed-vertex-count configuration (FVC), the InnerIter range for each SSG is computed by dividing the total number of vertices by the number of SSGs specified with X.

Generalizing Graph Iteration Spaces. The graph iteration space concept can be generalized to expand the space of supported optimizations. In GraphIt, we restrict the graph iteration space to four dimensions with fixed partitioning schemes. Adding more dimensions and/or removing constraints on how the dimensions are partitioned can potentially represent additional optimizations. For example, distributed graph computing can potentially be expressed as another dimension in graph partitioning.

5.1.2 Vertex Data Layout and Program Structure Optimizations Representation

Since the vertex data are stored as abstract vectors, they can be implemented as an array of structs or struct of arrays. We use vector tags to tag each vertex data vector as Array of Structs (AoS) or a separate array in the implicit global struct (SoA). These tags can be configured with the `fuseFields` scheduling function. Program structure optimizations update the structure of the loops and edgeset `apply` operators. We use the scoped labels (described in Section 4.2.2), which are specified in the scheduling language with `fuseForLoop` and `fuseApplyFunctions`, to represent the optimizations. These scheduling representations are used by the compiler to generate appropriate data

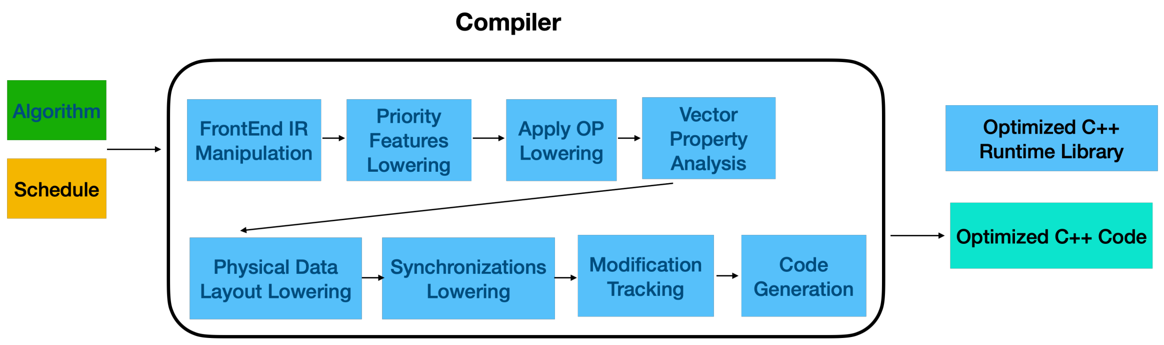


Figure 5-5: Organization of the compiler passes in GraphIt

structure declarations, initializations, and access methods as described in Section 5.2.2.

5.2 Compiler Implementation

Once the scheduling representations, such as the graph iteration space, are constructed, the compiler must apply program analyses, transformations, and code generation algorithms to finally produce high-performance implementations. This section describes the different passes of the GraphIt compiler and the autotuner , which is built on top of the compiler to automatically discover high-performance schedules.

We first present the high-level overview of the passes in Figure 5-5. The FrontEnd IR Manipulation pass enables the program structure optimizations (Section 5.2.3). The Priority Features Lowering pass, along with our optimized runtime library, perform the optimizations for the ordered graph algorithms, including bucketing (Section 5.2.4 and Section 5.2.5). The Apply Operator Lowering, the Vector Property Analysis,the Synchronization Lowering, and the Modification Tracking passes set up the code generation for the graph iteration space (Section 5.2.1). The Physical Data Layout Lowering pass applies the vertex data layout optimizations (Section 5.2.2). Finally the compiler generates optimized C++ programs for the operators specified in the algorithm, such as parallel edge traversal and processing. We also built an optimized C++ runtime library for low-level operations, such as loading graphs and switching between different vertexset data layouts.

```
# generate edgeset traversal code based on
# apply_expr: algorithmic operators. such as apply, from, to, src_filter, and dst_filter
# gis_vec_list: graph iteration space vectors generated with the scheduling language
gen-edgeset-apply (List<GraphIterSpaceVector> gis_vec_list, EdgesetApplyExpr apply_expr){
    # two graph iteration space vectors might be supplied
    # one of the two graph iteration space vectors will be selected at runtime
    if (gis_vec_list contains two graph iteration space vectors) {
      # generate a condition to select one of the two graph iteration space vectors
       print "if"; emit-gis-vector-select-condition (gis_vec_list, apply_expr); print "{";
       gen-SSG (gis_vec_list[0], apply_expr);
       print " } else { ";
       gen-SSG (gis_vec_list[1], apply_expr);
       print "}";
    } else { gen-SSG (gis_vec_list[0], apply_expr); }
}
# generate traversal code for the SSG dimension
gen-SSG (GraphIterSpaceVector gis_vec, EdgesetApplyExpr apply_expr) {
    emit-SSG-traversal-loop (gis_vec, apply_expr); # see subsection SSG Code Generation
    gen-BSG (gis_vec, apply_expr);
    emit-SSG-post-traversal-code (gis_vec, apply_expr);
}
# generate traversal code for the BSG dimension
gen-BSG (GraphIterSpaceVector gis_vec, EdgesetApplyExpr apply_expr) {
    emit-BSG-traversal-loop (gis_vec, apply_expr); # see subsection BSG Code Generation
    gen-OuterIter-InnterIter (gis_vec, apply_expr);
    emit-BSG-post-traversal-code (gis_vec, apply_expr);
}
# generate traversal code for the outerIter and innerIter
gen-OuterIter-InnterIter (GraphIterSpaceVector gis_vec, EdgesetApplyExpr apply_expr) {
    # see subsection OuterIter and InnerIter Code Generation
    emit-OuterIter-InnerIter-Nested-loops (gis_vec, apply_expr);
}
```

Figure 5-6: Code generation algorithm for the graph iteration space.

5.2.1 Code Generation for Graph Iteration Space

We first show the high-level code generation algorithm for the graph iteration space in Figure 5-6. To deal with hybrid traversal modes that have two graph iteration space vectors, such as DensePull-SparsePush, `gen-edgeset-apply` generates two implementations of edge traversal logic with additional logic to choose an implementation based on the sum of the out-degrees of active vertices (the "if", "else", and `emit-gis-vector-select-condition` shown in Figure 5-6) as described in Section 3.1. The functions `gen-SSG`, `gen-BSG`, and `gen-OuterIter-InnerIter` generate nested traversal loops for the different graph iteration space dimensions. Below, we provide more details on the code generation functions and the mechanisms to ensure the validity of

```
#s1# edges.from(Frontier).dstFilter(dstFunc).apply(applyFunc)
schedule:
program->configApplyDirection("s1","SparsePush");
```

Figure 5-7: SparsePush configuration.

```
for (int i = 0; i < Frontier.size(); i++){
  NodeID src = Frontier.vert_array[i];
  for (NodeID dst : G.getOutNghs(src)){
    if (dstFunc(dst)){
      applyFunc(src, dst); }}}
```

Figure 5-8: Generated SparsePush code.

the optimizations.

OuterIter and InnerIter Code Generation. We demonstrate how to generate traversal code for the OuterIter and InnerIter dimensions using a simple example with the SparsePush configuration shown in Fig. 5-7 (graph iteration space and tags: $\langle$ $\perp$, $\perp$, O [src, SR, SA], I [dst, SR, BA] $\rangle$; abbreviations and sets are defined in Fig. 5-3 and Fig. 5-1).

For the push direction, OuterIter is src and InnerIter is dst. Since the source (OuterIter) filtering is tagged as Sparse Array (SA), the outer loop iterates over the source `vertexset` (Frontier). The `dst` (InnerIter) filtering uses the user-defined boolean function `dstFunc`. The generated code is displayed in Fig. 5-8.

We show code generated for a DensePull traversal mode ($\langle$ $\perp$, $\perp$, O [dst, SR, BA], I [src, SR, BA] $\rangle$) in Fig. 5-9. The OuterIter is now dst and the InnerIter is src. The user-defined function `applyFunc` is applied to every edge as before. The `vertexsets` are automatically converted from the sparse array of vertices (`vert_array` shown in the SparsePush example above) to a boolean map (`bool_map` in the DensePull example). Filtering on destination vertices (`dstFilter`) is attached as an `if` statement next to the dst iterator (OuterIter).

Blocked Subgraph (BSG) Code Generation. The BSG dimension in the graph

```
for (NodeID dst = 0; dst < num_verts; dst++){
  if (dstFunc){
    for (NodeID src : G.getInNghs(dst)){
      if (Frontier.bool_map(src)){
        applyFunc(src, dst); }}}}
```

Figure 5-9: Generated DensePull code.

```
parallel_for (int BSG_ID = 0; BSG_ID < g.num_chunks; BSG_ID++){
  for (NodeID src = g.chunk_start[BSG_ID]; src < g.chunk_end[BSG_ID]; src++)
    for (NodeID dst : G.getOutNghs(src))
      applyFunc(src,dst);}
```

Figure 5-10: Generated blocked subgraph (BSG) code.

```
for (int SSG_ID = 0; SSG_ID < num_SSG; SSG_ID++){
  sg = g.SSG_list[SSG_ID];
  for (int BSG_ID = 0; BSG_ID < sg.num_chunks; BSG_ID++){
    for (NodeID dst = sg.chunk_start[BSG_ID]; dst < sg.chunk_end[BSG_ID]; dst++)
      for (NodeID src : G.getInNghs(dst))
        applyFunc(src,dst);}}
```

Figure 5-11: Generated segmented subgraph (SSG) and blocked subgraph (BSG) code.

iteration space is created by partitioning the OuterIter dimension. GraphIt uses the partitioning tag for this dimension to control the granularity and blocking strategy for load balancing, and the parallelization tag to control the mode of parallelization. Figure 5-10 shows an example of the generated code, assuming OuterIter represents src.

If the edge-aware vertex count (EVC) partitioning tag is used, the compiler generates chunks with approximately the number of edges specified by the schedule. For the fixed vertex count (FVC) partitioning tag, the compiler uses the built-in grain size in OpenMP. For parallelization tags static parallelism (SP) and dynamic work-stealing parallelism (WSP), we simply use the OpenMP pragmas to implement `parallel_for` (`pragma omp for parallel schedule (static)` and `schedule (dynamic)`).

Segmented Subgraph (SSG) Code Generation. Using the SSG dimension requires adding a loop outside of the existing traversals and changing the data layout of the graph. GraphIt generates code in the main function to create the SSGs by partitioning the graph by InnerIter. This partitioning can use a fixed range of vertices (FVC) in the InnerIter or a flexible range of vertices that takes into account the number of edges in each SSG (EVC) with an edge grain size. The random memory access range in each SSG is restricted to improve locality. Fig. 5-11 shows edge traversal code that uses both SSG and BSG dimensions ($\langle$ S [SR], B [SR], O [dst, SR], I [src, SR] $\rangle$). The segmented subgraphs are stored in `g.SSG_`list.

The cache optimization processes one SSG at a time (SR), but processes the BSGs within the SSG in parallel. The programmer can enable NUMA optimizations by specifying the parallelization tag for SSG as static parallel (SP); the compiler then assigns different SSGs to be executed on different sockets. GraphIt implements this assignment using `numa_alloc` in the main function to first allocate SSGs on different sockets, and then uses the `proc_bind` API in OpenMP to assign threads to process each socket-local subgraph. If work-stealing parallelism (WSP) is enabled for SSGs, then a socket can steal an SSG allocated on another socket if no work remains on the current socket.

To apply NUMA optimizations, we use NUMA-local (socket-local) buffers to store the intermediate results from each SSG. The compiler generates code for allocating NUMA-local buffers and changes the data references from updating global vertex data vectors to the NUMA-local buffers. The compiler also generates code for a merge phase that merges NUMA-local buffers to update the global data vectors.

Validity of Optimizations. GraphIt ensures the validity of single edge traversal optimizations by imposing a set of restrictions on the GraphIt language and using dependence analysis to insert appropriate atomic synchronization instructions.

We enforce some restrictions on read-write accesses and reduction operators for vertex data vectors across UDFs used in `srcFilter`, `dstFilter`, and edgeset `apply` functions for a given `edgeset` traversal operation. Each vertex data vector must have only one of the following properties: read-only, write-only, or reduction. Additionally, reductions are commutative and associative. With these two restrictions, transformations do not need to preserve read-after-write dependences, and transformations remain valid independent of edge traversal order. Therefore, a transformed program is valid as long as each filtered edge is processed exactly once.

To ensure that each filtered edge is processed exactly once, we insert synchronization code to vertex data vector updates by leveraging dependence analysis theory from dense loop iteration spaces [65, 60]. Dependence analysis is well-suited for GraphIt because the language prevents aliasing and each vertex data vector represents a separate data structure. Additionally, the goal of the analysis is not to automatically parallelize the loop with a correctness guarantee, but the much easier task of determining whether

```
for (int i = 0; i < Frontier.size(); i++){
  NodeID src = Frontier.vert_array[i];
  for (NodeID dst : G.getOutNghs(src)){
      DeltaSum[dst] += Delta[src]/OutDegree[src];
      }}}
```

Figure 5-12: PageRankDelta edge traversal code with SparsePush

Table 5.3: Dependence vectors for PageRankDelta edge traversal code with SparsePush

Vector Dependence	Distance Vector	Read-Write
DeltaSum	$\langle *, 0\rangle$	reduction
Delta	$\langle 0, 0\rangle$	read-only
OutDegree	$\langle 0, 0\rangle$	read-only

synchronization code is necessary for a given parallelization scheme. Accomplishing this task does not require a precise distance vector.

Figure 5-12 shows a code snippet of PageRankDelta with the SparsePush configuration ($\langle$ $\perp$, $\perp$, O [*src*, SR, SA], I [*dst*, SR] $\rangle$), the distance vector, and read-write properties of the vectors. The compiler builds a dependence vector for the OuterIter and InnerIter dimensions based as listed in Table 5.3 for the push direction. We observe that DeltaSum has a dependence with the reduction operator (it is both read from and written to). Different source nodes can update the same *dst*, and so we assign $*$ to the first element of the distance vector to denote that a dependence exists on an unknown iteration of *src*, which maps to OuterIter based on the direction tags. Given a *src*, we know that the *dst*'s are all different, and thus, there is no data dependence on the second iterator and we assign the second value of the distance vector as 0. Since Delta and OutDegree are both read-only, they have the distance vector $\langle 0, 0\rangle$ with no dependence across different iterations. Given that DeltaSum's distance vector's first element is $*$, the compiler knows that synchronization must be provided when parallelizing the OuterIter (outer loop). If only the InnerIter (inner loop) is parallelized, then no synchronization is needed.

A similar analysis works on a DensePull ($\langle$ $\perp$, $\perp$, O [*dst*, SR, BA], I [*src*, SR] $\rangle$) PageRankDelta. The code snippet and distance vectors are shown in Figure 5-13 and Table 5.4. The first element in the distance vector for DeltaSum is 0 because there is no dependence between different destination vertices and OuterIter represents *dst*. However, the value is $*$ on the second element because different sources will

```
for (NodeID dst = 0; dst < num_verts; dst++) {
  for (NodeID src : G.getInNghs(dst)){
      if (Frontier.bool_map(src))
         DeltaSum[dst] += Delta[src]/OutDegree[src];
         }}}
```

Figure 5-13: PageRankDelta edge traversal code with DensePull

Table 5.4: Dependence vectors for PageRankDelta edge traversal code with DensePull

Vector Dependence	Distance Vector	Read-Write
DeltaSum	$\langle 0, * \rangle$	reduction
Delta	$\langle 0, 0 \rangle$	read-only
OutDegree	$\langle 0, 0 \rangle$	read-only

update the same destination. Parallelizing OuterIter in this case does not require any synchronization in this case.

Because BSG is partitioned by OuterIter, parallelizing the BSG dimension would have the same effect as parallelizing OuterIter. Similarly, parallelizing SSG has the same effect as parallelizing InnerIter given that SSG is partitioned by InnerIter.

When applying NUMA optimizations to the second code snippet with the DensePull direction (parallelizing both the SSG and BSG dimensions), we have a dependence vector of $\langle *, * \rangle$ for DeltaSum. In this case, GraphIt writes the updates to DeltaSum$[dst]$ to a socket-local buffer first and later merges buffers from all sockets to provide synchronization.

For the hybrid traversal configurations, GraphIt generates two versions of the user-defined `apply` function because the synchronization requirements for the push and pull directions are different. Each version will be used in the corresponding traversal mode.

```
func vertexset_apply_f(v:Vertex)
  parent[v] = -1;
end
func main()
  vertices.apply(vertexset_apply_f);
end
```

Figure 5-14: Vertexset `apply` code.

```
for (NodeID v = 0; v < num_verts; v++) {
  fused_struct[v].parent = -1;
}
```

Figure 5-15: Generated parent initialization code with array of structs

5.2.2 Code Generation for Vertex Data Layout Optimizations

To generate code with different physical data layouts for the vertex data (array of structs or struct of arrays), GraphIt generates declaration and initialization code in the main function and updates references to vertex data in the other functions. The compiler first transforms assignments on vertex data vectors into `vertexset apply` operations that set the values of the data vectors. If the programmer specifies the `fuseField` command, GraphIt generates a new struct type, an array of structs declaration, and changes the references to the vertex data vectors in the functions to access fields of the struct instead of separate arrays. For example, the assignment statement for the parent vector `parent : vector {Vertex}(int) = -1;` is implemented by first declaring an `apply` function `vertexset_apply_f` and another `vertices.apply` statement in the main function that uses `vertexset_apply_f` as shown in Fig. 5-14. The vector access expression in the `apply` function will then be lowered from `parent[v]` to `fused_struct[v].parent`, as indicated in Figure 5-15. The correctness of the program is not affected by the vertex data layout optimization because it does not affect the execution ordering of the program.

5.2.3 Code Generation for Program Structure Optimizations

Traditional compilers with a fixed number and order of optimization passes are ill-suited for program structure optimizations, such as kernel fusion. The GraphIt compiler introduces a new schedule-driven optimization pass orchestration design that allows users to add more optimization passes and dictate the order of the added optimizations with the label-based scheduling language described in Section 4.2. Users can perform fine-grained loop fusion, loop splitting, and fusion of `apply` functions on loops and functions specified with statement labels and scheduling commands. These optimizations are implemented as passes that manipulate the frontend intermediate representation, shown as the FrontEnd IR Manipulation pass in Figure 5-5. GraphIt implements these program structure transformation schedules by adding new optimization passes, which transform the intermediate representation. These extra passes are added dynamically with the scheduling commands specified by the user.

5.2.4 Lazy Bucket Update Optimization

Next, we demonstrate how the compiler generates code for different bucketing optimizations, including lazy and eager bucket update strategies. As discussed in Section 3.4, the lazy approach buffers and reduces bucket updates to the same vertex at each round before making the one final actual bucket update, whereas the eager approach immediately updates the bucket every time. The key challenges are in how to insert low-level synchronization and deduplication instructions, and how to combine bucket optimizations with direction optimization and other optimizations in the original GraphIt scheduling language. Furthermore, the compiler has to perform global program transformations and code generation to switch between lazy and eager approaches.

To support the lazy bucket update approach, the compiler applies program analyses and transformations on the user-defined functions (UDFs). The compiler uses dependence analysis on `updatePriorityMin` and `updatePrioritySum` to determine whether write-write conflicts exist and insert atomics instructions as necessary (Figure 4-10(a) Line 20). Additionally, the compiler must insert variables to track whether a vertex's priority has been updated or not (`tracking_var` in Figure 4-10(a), Line 18). This variable is used in the generated code to determine which vertices should be added to the buffer `outEdges` (Figure 4-10(a), Line 21). Deduplication is enabled with a compare-and-swap (CAS) on deduplication flags (Figure 4-10(a), Line 21) to ensure that each vertex is inserted into the outEdges only once. Deduplication is required for correctness for applications such as k-core. In this case, the user specifies that deduplication is required in the algorithm specification for k-core to ensure no schedule generates incorrect code. Changing the schedules with different traversal directions or frontier layouts affects the code generation for edge traversal and UDFs (Figure 4-

```
func apply_f(src: Vertex, dst: Vertex)
   var k: int = pq.get_current_priority();
   pq.updatePrioritySum(dst, -1, k);
end
```

```
apply_f_transformed = [&] (uint vertex, uint count) {
   int k = pq->get_current_priority();
   int priority = pq->priority_vector[vertex];
   if (priority > k) {
     uint __new_pri = std::max(priority + (-1) * count, k);
     pq->priority_vector[vertex] = __new_pri;
     return wrap(vertex, pq->get_bucket(__new_pri));}}
```

Figure 5-16: The original (top) and transformed (bottom) user-defined function for k-core using lazy bucket with constant sum reduction.

10(b)). In the DensePull traversal direction, no atomics are needed for the destination nodes.

We built runtime libraries to manage the buffer and update buckets. The compiler generates appropriate calls to the library (`getNextBucket`, `setupFrontier`, and `updateBuckets`). The `setupFrontier` API (Figure 4-10(a), Line 24) performs a prefix sum on the `outEdges` buffer to compute the next frontier. We use a lazy priority queue (declared in Figure 4-10(a), Line 2) for storing active vertices based on their priorities. The lazy bucketing is based on Julienne's bucket data structures that only materialize a few buckets, and keep vertices outside of the current range in an overflow bucket [32]. We improve its performance by redesigning the lazy priority queue interface. Julienne's original interface invokes a lambda function call to compute the priority. The new priority-based extension computes the priorities using a priority vector and Δ value for priority coarsening, eliminating extra function calls for computing the coarsened priorities.

Lazy bucket with constant sum reduction. We also incorporated a specialized histogram-based reduction optimization (first proposed in Julienne [32]) to reduce priority updates with a constant value each time. This optimization can be combined with the lazy bucket update strategy to improve performance. For k-core, because the priorities for each vertex always reduce by one at each update, we can optimize it further by keeping track of only the number of updates with a histogram. This way, we avoid contention on vertices that have a large number of neighbors on the frontier.

To generate code for the histogram optimization, the compiler first analyzes the

UDF to determine whether the change to the priority of the vertex is a fixed value and if it is a sum reduction (Figure 5-16 (top), Line 3). The compiler ensures that there is only one priority update operator in the UDF. It then extracts the fixed value (`-1`), the minimum priority (`k`), and vertex identifier (`dst`). In the transformed function (Figure 5-16 (bottom)), an if statement and max operator are generated to check and maintain the priority of the vertex. The `applyUpdatePriority` operator obtains the counts of updates to each vertex using a histogram approach and supplies the vertex and count as arguments to the transformed function (Figure 5-16 (bottom), Line 1). The compiler copies all of the expressions used in the priority update operator and the expressions that they depend on in the transformed function.

5.2.5 Eager Bucket Update Optimization

The eager bucket update approach immediately applies every update to the bucket when the priorities of the vertices are changed. The compiler uses program analysis to determine feasibility of the transformation, transforms UDFs and edge traversal code, and uses optimized runtime libraries to generate efficient code for the eager bucket update approach.

The compiler analyzes the while loop (Figure 4-2, Lines 17–21) to look for the pattern of an iterative priority update with a termination criterion. The analysis checks that there is no other use of the generated vertexset (`bucket`) except for the `applyUpdatePriority` operator, ensuring correctness.

Once the analysis identifies the while loop and edge traversal operator, the compiler replaces the while loop with an ordered processing operator. The ordered processing operator uses an OpenMP parallel region (Figure 4-10(c), Lines 12–32) to set up thread-local data structures, such as `local_bins`. We built an optimized runtime library for the ordered processing operator based on the Δ-stepping implementation in GAPBS [14]. A global vertex frontier (Figure 4-10(c), Line 11) keeps track of vertices of the next priority (the next bucket). In each iteration of the while loop, the `#pragma omp for` (Figure 4-10(c), Lines 15–16) distributes work among the threads. After priorities and buckets are updated, each local thread proposes its next bucket priority, and the smallest priority across threads will be selected (omitted code on Figure 4-10(c), Line 28). Once the next bucket priority is determined, each thread copies vertices in its next local bucket to the global frontier (Figure 4-10(c), Line 30)

Finally, the compiler transforms the UDFs by appending the local buckets to the argument list and inserting synchronization instructions. These transformations allow priority update operators to directly update thread-local buckets (Figure 4-10(c), Lines 23–26).

5.2.6 Eager Bucket Fusion Optimization

In this section, we describe the design and implementation of the new bucket fusion optimization for the eager bucket update approach first introduced in Section 3.4.3 and shown in Algorithm 3-8. The optimization is very effective (achieving more than three times speedup over state-of-the-art implementations) on the road networks for ordered graph algorithms, such as Single Source Shortest Paths (SSSP) with delta stepping as shown in Section 6.3.5. This optimization has since been integrated into the popular GAP benchmark suite [14].

The bucket fusion optimization adds another while loop after the end of the for-loop on Line 27 of Figure 4-10(c), and before finding the next bucket across threads on Line 28. This inner while loop processes the current bucket in the local priority queue (`local_bins`) if it is not empty and its size is less than a threshold. In the inner while loop, vertices are processed using the same transformed UDFs as before to process the local bucket. This simple optimization can significantly reduce the number of rounds, thus reducing barrier synchronizations at the end of each round. Another benefit of this approach is also that the local buckets likely to be in the cache and the same NUMA node. Thus, this optimization improves the load balance and locality by enforcing that the threads to process the thread-local bucket of the next priority first when it is available.

The optimization can potentially result in significant load imbalance issues if the thread-local bucket of the next priority is too large, creating stragglers. To address this issue, we introduced a threshold on the size of the thread-local bucket to be processed as shown in Algorithm 3-8 Line 18. The size threshold improves load balancing, as only small local buckets are processed by the current thread. Large buckets are still distributed across different threads to reduce the load on the current thread. We found the threshold of 1000 to be fairly robust, and it can be used for all kinds of different algorithms, graphs, and multi-core CPUs.

5.2.7 Autotuning GraphIt Schedules

Finding the right set of schedules can be challenging for non-experts. GraphIt can have up to 10^5 valid schedules with each run taking more than 30 seconds for our set of applications and input graphs. Exhaustive searches would require weeks of time. As a result, we use OpenTuner [8] to build an autotuner on top of GraphIt that leverages stochastic search techniques (e.g., AUC bandit, greedy mutation, differential mutation, and hill climbing).

Search space. We limit the tuning to a single edgeset `apply` operation identified by the user. The user does not need to supply any template for the schedules. The autotuner will try different configurations, such as the direction of the traversal (`configApplyDirection`), the parallelization scheme (`configApplyParallelization`), the data layout for the dense vertexset (`configApplyDenseVertexSet`), the partitioning strategy of the graph (`configApplyNumSSG`), the NUMA execution policy (`configApplyNUMA`), and the priority update strategies(`configApplyPriorityUpdate`).

Not all generated schedules are valid because schedules have dependencies among them. For example, `configApplyNumSSG`, which takes a direction parameter, is only valid if the direction specified is also set by `configApplyDirection`. Instead of reporting an invalid schedule as error, GraphIt's autotuner ignores invalid schedules to smooth the search space. For example, the `configApplyNumSSG` configuration is ignored if the specified direction is invalid.

5.3 Chapter Summary

In this chapter, we first described the graph iteration space scheduling representation used by the compiler. We then demonstrated how GraphIt leverages various program analyses, transformations, and code generation algorithms to generate efficient code. Finally, we provided more details on the implementation of the optimized bucketing data structures and the autotuner built on top of GraphIt. In Chapter 6 we will evaluate the performance of the generated implementations and demonstrate that the generated code can match or exceed the performance of state-of-the-art graph processing frameworks on CPUs across different algorithms and graphs.

Chapter 6

Evaluation

In this chapter, we demonstrate the performance of GraphIt on both unordered and ordered graph algorithms. We compare the performance of GraphIt with state-of-the-art graph processing frameworks and analyze the tradeoff of various optimizations (schedules). We demonstrate the performance effects of the novel optimizations introduced in this thesis, including the program structure optimization, cache optimization, and bucket fusion optimization.

6.1 Experimental Setup

We use a dual socket system with Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores and 48 hyper-threads. The system has 128GB of DDR3-1600 memory and 30 MB last level cache on each socket, and runs with Transparent Huge Pages (THP) enabled.

Datasets. Table 6.1 lists our input datasets and their sizes. LiveJournal, Twitter, and Friendster are three social network graphs. Friendster is special because its number of edges does not fit into a 32-bit signed integer. We use WebGraph from the 2012 common crawl. The Netflix rating dataset and its synthesized expansion (Netflix2x) are used to evaluate Collaborative Filtering. For k-core and SetCover, we symmetrize the input graphs. For Δ-stepping based SSSP, wBFS, PPSP using Δ-stepping, and A^* search, we use the original directed versions of the graphs with integer edge weights.

Table 6.1: Graphs used for experiments. The number of edges are directed edges. Graphs are symmetrized for k-core and SetCover.

Type	Dataset	Num. Verts	Num. Edges	Symmetric Num.Edges
Social	*Orkut* (OK) [109]	3 M	234 M	234 M
Graphs	*LiveJournal* (LJ) [29]	5 M	69 M	85 M
	Twitter (TW) [55]	41 M	1469 M	2405 M
	Friendster (FT) [109]	125 M	3612 M	3612 M
Web Graph	*WebGraph* (WB) [69]	101 M	2043 M	3880 M
Road	*Massachusetts* (MA) [1]	0.45 M	1.2 M	1.2 M
Graphs	*Germany* (GE) [1]	12 M	32 M	32 M
	RoadUSA (RD) [30]	24 M	58 M	58 M
User-Item Bipartite Graph	*Netflix* (NX) [17]	0.5 M	198 M	
	Netflix2x (NX2) [58]	1 M	792 M	

The RoadUSA (RD), Germany(GE) and Massachusetts (MA) road graphs are used for the A* search algorithm, as they have the longitude and latitude data for each vertex. GE and MA are constructed from data downloaded from OpenStreetMap [1].

We used different weights for the same graphs in different parts of the evaluation. The specific weight distributions used for experiments are described in the caption of Table 6.2 for the unordered graph algorithms and in Table 6.6 for the ordered graph algorithms.

6.2 Unordered Graph Algorithms

In this section, we compare GraphIt's performance to state-of-the-art frameworks and DSLs on graphs of various sizes and structures. We also analyze performance tradeoffs among different GraphIt schedules.

6.2.1 Algorithms

We try to use the same algorithms across different frameworks to study the impact of performance optimizations. Our evaluation is performend on the seven algorithms listed below. All the GraphIt implementations are available in the repository.[1]

PageRank (PR). We use an iterative sparse matrix dense vector (SpMV) multiplications algorithm with synchronous updates and double buffering. For this section,

[1]The GraphIt compiler is available under the MIT license at `http://graphit-lang.org/`

we have each framework execute 20 iterations and record the average execution time for each round.

Breadth-First Search (BFS). We use a direction optimized algorithm that computes the parent of each node. The direction optimization can be configured through the scheduling language.

Connected Components (CC). We use a synchronous label propagation based implementation. This implementation is less efficient than the sampling-based afforested algorithms [96] proposed recently.

Single Source Shortest Paths (SSSP). We use a frontier-based Bellman-Ford algorithm that relies on label propagation. In Section 6.3, we use a more efficient delta-stepping based implementation.

Collaborative Filtering (CF). We use a gradient-descent based implementation that minimizes the errors across all the nodes similar to GraphMat's implementation [95].

Betweenness Centrality (BC). We use a direction-optimized Brandes algorithm [22] similar to Ligra's implementation [91]. The algorithm computes the betweenness centrality values of each node from a single starting point.

PageRankDelta (PRDelta). We use a direction-optimized algorithm that only propagates changes in the rank values at each round, similar to Ligra's implementation [91]. The algorithm iterates until the changes in the rank values are lower than the specified threshold.

6.2.2 Existing Frameworks

We compare GraphIt's performance to six state-of-the-art in-memory graph processing systems: Ligra [91], GraphMat [95], Green-Marl [46], Galois [74], Gemini [118], and Grazelle [39]. Ligra has fast implementations of BFS and SSSP [91]. Among prior work, GraphMat has the fastest shared-memory implementation of collaborative filtering [95]. Green-Marl is one of the fastest DSLs for the algorithms we evaluate [46]. Galois (v2.2.1) has an efficient asynchronous engine that works well on road graphs

[74]. Gemini is a distributed graph processing system with notable shared-machine performance [118]. Compared to existing frameworks, Grazelle has the fastest PR and CC using edge list vectorization, inner loop parallelism, and NUMA optimizations [39].

We tried to use the same algorithms across the different graph processing frameworks. For Galois, we used the asynchronous algorithm for BFS and the label propagation algorithm for SSSP. Ligra and Grazelle use the same level-synchronous algorithms as GraphIt. Only Ligra implements PRDelta, Ligra and GraphMat implement CF, and Grazelle does not implement SSSP. Green-Marl and GraphMat crashed when we attempt to process the Friendster graph.

6.2.3 Comparisons with State-of-the-Art Frameworks

Our experiments reveal that GraphIt outperforms the next fastest of the shared-memory frameworks on 24 out of 32 experiments by up to 4.8×, and is never more than 43% slower than the fastest framework on the other experiments. For each framework and DSL, we present a heat map of slowdowns compared to the fastest of all seven frameworks and DSLs in Figure 6-1. GraphIt introduces the new cache segmenting optimization described in Section 3.2.3. The DSL achieves competitive or better performance compared to other frameworks also by generating efficient implementations of known combinations of optimizations, and finding previously unexplored combinations by searching through a much larger space of optimizations. GraphIt also reduces the lines of code compared to the next fastest framework by up to one order of magnitude.

Table 6.2 shows the execution time of GraphIt and other systems. The best performing schedules for GraphIt are shown in Table 6.3. Table 6.4 shows the line counts of four graph algorithms for each framework. GraphIt often uses significantly fewer lines of code compared to the other frameworks. Unlike GraphIt, other frameworks with direction optimizations require programmers to provide many low-level implementation details as discussed in Section 4.1.

GraphIt outperforms the next fastest of the six state-of-the-art shared-memory frameworks on 24 out of 32 experiments by up to 4.8×, and is never more than 43%

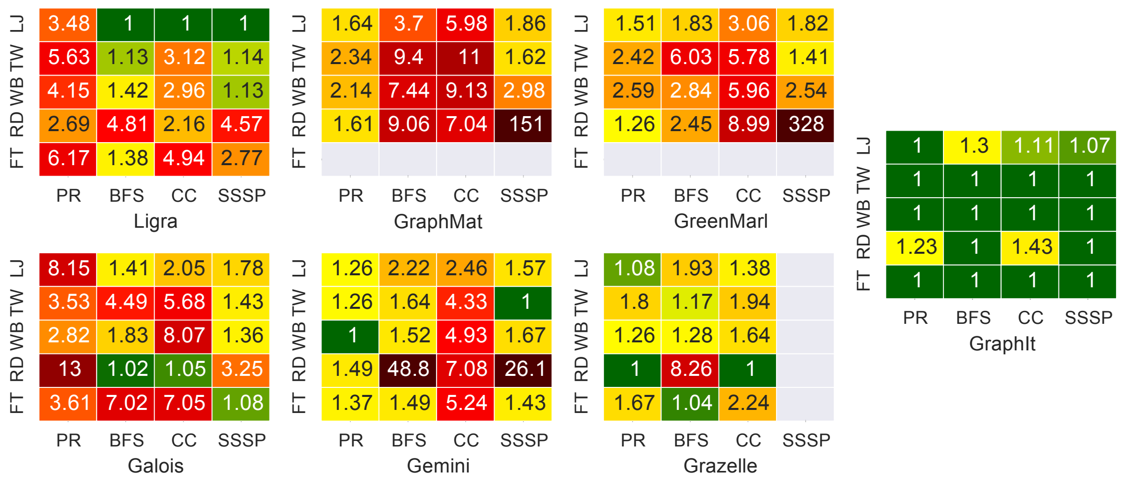


Figure 6-1: A heat map of slowdowns of various frameworks compared to the fastest of all frameworks for PageRank (PR), Breadth-First Search (BFS), Connected Components (CC) using label propagation, and Single Source Shortest Paths (SSSP) using Bellman-Ford, on five graphs with varying sizes and structures (LiveJournal (LJ), Twitter (TW), WebGraph (WB), USAroad (RD), and Friendster (FT)). Lower numbers (green) are better, with one being the fastest for the specific algorithm running on the specific graph. Gray indicates that either an algorithm or a graph is not supported by the framework. We try to use the same algorithms across different frameworks. For Galois, we used the asynchronous algorithm for BFS, and the Ligra algorithm for SSSP.

slower than the fastest framework on the other experiments.

PageRank (PR). GraphIt has the fastest PR on 4 out of the 5 graphs and is up to 54% faster than the next fastest framework because it enables both cache and NUMA optimizations when necessary as described in Section 3.2.3. Table 6.5 shows that on the Twitter graph, GraphIt has the lowest LLC misses, QPI traffic, and cycles stalled compared to Gemini and Grazelle, which are the second and third fastest. GraphIt also reduces the line count by up to an order of magnitude compared to Grazelle and Gemini as shown in Table 6.4. Grazelle uses the Vector-Sparse edge list to improve vectorization, which works well on graphs with low-degree vertices [39], outperforming GraphIt by 23% on USAroad. GraphIt does not yet have this optimization. Frameworks other than Gemini and Grazelle do not optimize for cache or NUMA, resulting in much worse running times.

Breadth-First Search (BFS). GraphIt has the fastest BFS on 4 out of the 5 graphs

Table 6.2: Running time (seconds) of GraphIt and state-of-the-art frameworks. The fastest results are bolded. The missing numbers correspond to a framework not supporting an algorithm and/or not successfully running on an input graph. Galois' Betweenness Centrality (BC) uses an asynchronous algorithm, while other frameworks use a synchronous one. We ran PageRank (PR) for 20 iterations, PageRankDelta (PRDelta) for 10 iterations, and Collaborative Filtering (CF) for 10 iterations. Breadth-First Search (BFS), Single Source Shortest Paths (SSSP), and Betweenness Centrality (BC) times are averaged over 10 starting points. The weights are all set to 1 for this table.

Algorithm	PR					BFS					CC					CF	
Graph	LJ	TW	WB	RD	FT	LJ	TW	WB	RD	FT	LJ	TW	WB	RD	FT	NX	NX2
GraphIt	**0.34**	**8.71**	**16.34**	0.91	**32.58**	0.04	**0.30**	**0.645**	**0.22**	**0.49**	0.07	**0.89**	**1.960**	17.10	**2.63**	**1.29**	**4.59**
Ligra	1.19	49.00	68.10	1.99	201.00	**0.027**	0.34	0.92	1.04	0.68	**0.06**	2.78	5.81	25.90	13.00	5.35	25.50
GraphMat	0.56	20.40	35.00	1.19		0.10	2.80	4.80	1.96		0.37	9.80	17.90	84.50		5.01	21.60
Green-Marl	0.52	21.04	42.48	0.93		0.05	1.80	1.83	0.53		0.19	5.14	11.68	107.93			
Galois	2.79	30.75	46.27	9.61	117.47	0.04	1.34	1.18	0.22	3.44	0.13	5.06	15.82	12.66	18.54		
Gemini	0.43	10.98	16.44	1.10	44.60	0.06	0.49	0.98	10.55	0.73	0.15	3.85	9.66	85.00	13.77		
Grazelle	0.37	15.70	20.65	**0.740**	54.36	0.05	0.35	0.83	1.79	0.51	0.08	1.73	3.21	**12.20**	5.88		

Algorithm	SSSP					PRDelta					BC				
Graph	LJ	TW	WB	RD	FT	LJ	TW	WB	RD	FT	LJ	TW	WB	RD	FT
GraphIt	0.06	**1.35**	**1.68**	**0.290**	**4.30**	**0.18**	**4.72**	**7.14**	**0.50**	**12.58**	0.10	1.55	2.50	**0.650**	**3.75**
Ligra	**0.05**	1.55	1.90	1.30	11.93	0.24	9.19	19.30	0.69	40.80	0.09	1.93	3.62	2.53	6.16
GraphMat	0.10	2.20	5.00	43.00											
Green-Marl	0.09	1.92	4.27	93.50							**0.08**	3.60	6.40	29.05	
Galois	0.09	1.94	2.29	0.93	4.64						0.24	3.40	4.29	0.81	9.90
Gemini	0.08	1.36	2.80	7.42	6.15						0.15	**1.36**	**2.30**	31.06	3.85

(up to 28% faster than the next fastest) because of its ability to generate code with different direction and bitvector optimizations. On LiveJournal, Twitter, WebGraph, and Friendster, GraphIt adopts Ligra's direction optimization. On USAroad, GraphIt always uses SparsePush and omits the check for when to switch traversal direction, reducing runtime overhead. In the pull direction traversals, GraphIt uses bitvectors to represent the frontiers when boolean array representations do not fit in the last level cache, whereas Ligra always uses boolean arrays and Grazelle always uses bitvectors. GraphIt outperforms Galois' BFS, even though Galois is highly-optimized for road graphs. GraphMat and Green-Marl do not have the direction optimization so it is much slower. Ligra is slightly faster than GraphIt on the smaller LiveJournal graph due to better memory utilization, but is slower on larger graphs.

Connected Components with Label Propagation (CC). GraphIt has the fastest CC on Twitter, WebGraph, and Friendster because of the direction, bitvector, and cache optimizations. Table 6.5 shows GraphIt's reduced LLC miss rate and cycles

Table 6.3: Schedules that GraphIt uses for all applications on different graphs. The schedules assume that the edgeset `apply` operator is labeled with s1. The keyword 'Program' and the continuation symbol '->' are omitted. 'ca' is the abbreviation for 'configApply'. Note that configApplyNumSSG uses an integer parameter (X) which is dependent on the graph size and the cache size of a system. BC has two edgeset `apply` operators, denoted with s1 and s2.

Apps	USAroad	LiveJournal	Twitter	WebGraph	Friendster

PR	caDirection(“s1”, ”DensePull")
	caParallelization(“s1”, ”dynamic-vertex-parallel”)
		caDirection(“s1”, ”DensePull")
		caParallelization(“s1”, “dynamic-vertex-parallel”)
			caDirection(“s1”, "DensePull")
			caParallelization(“s1”, “edge-aware-dynamic-vertex-parallel”)
			caNumSSG(“s1”, "fixed-vertex-count", X)
			caNUMA(”s1”, ”dynamic-static-parallel”)		

BFS	caDirection(“s1”, "SparsePush")
	caParallelization(“s1”, “static-vertex-parallel”)
		caDirection(“s1”, "DensePull-SparsePush")
		caParallelization(“s1”, “dynamic-vertex-parallel")
			caDirection(“s1”, ”DensePull-SparsePush")
			caDenseVertexSet(“s1”, “src-vertexset”, “bitvector”, “DensePull”)
			caParallelization(“s1”, "dynamic-vertex-parallel")		

CC	caDirection(“s1”, "DensePush-SparsePush")
	caParallelization(“s1”, “static-vertex-parallel”)
		caDirection(“s1”, "DensePull-SparsePush")
		caParallelization(“s1”, "dynamic-vertex-parallel")
			caDirection(“s1”, "DensePull-SparsePush")
			caDenseVertexSet(“s1”, “src-vertexset”, “bitvector”, “DensePull”)
			caParallelization(“s1”, "dynamic-vertex-parallel")
			caNumSSG(“s1”, “fixed-vertex-count", X, “DensePull”)		

SSSP	caDirection(“s1”, "SparsePush")
	caParallelization(“s1”, “dynamic-vertex-parallel")
		caDirection(“s1”, "DensePush-SparsePush")
		caParallelization(“s1”, "dynamic-vertex-parallel")
			caDirection(“s1”, "DensePush-SparsePush")
			caParallelization(“s1”, "dynamic-vertex-parallel")		

PRDelta	caDirection(“s1”, "SparsePush")
	caParallelization(“s1”, “dynamic-vertex-parallel”)
	fuseFields("OutDegree", "Delta")
		caDirection(“s1”, "DensePull-SparsePush")
		caParallelization(“s1”, "dynamic-vertex-parallel")
		fuseFields("OutDegree", "Delta")
			caDirection(“s1”, "DensePull-SparsePush")
			caDenseVertexSet(“s1”, “src-vertexset”, “bitvector”, “DensePull")
			caParallelization(“s1”, "dynamic-vertex-parallel")
			caNumSSG(“s1”, "fixed-vertex-count", X, “DensePull”)
			caNUMA(“s1”, “static-parallel”, “DensePull")
			fuseFields("OutDegree", "Delta")		

BC	caDirection(“s1”, "SparsePush")
	caParallelization(“s1”, “static-vertex-parallel”)
	caDirection(“s2”, "SparsePush")
	caParallelization(“s2”, “static-vertex-parallel”)
		caDirection(“s1”, "DensePull-SparsePush")
		caParallelization(“s1”, “dynamic-vertex-parallel”)
		caDirection(“s2”, "DensePull-SparsePush")
		caParallelization(“s2”, “dynamic-vertex-parallel")
			caDirection(“s1”, ”DensePull-SparsePush")
			caDenseVertexSet(“s1”, “src-vertexset”, “bitvector”, “DensePull”)
			caParallelization(“s1”, “dynamic-vertex-parallel")
			caDirection(“s2”, ”DensePull-SparsePush")
			caDenseVertexSet(“s2”, “src-vertexset”, “bitvector”, “DensePull”)
			caParallelization(“s2”, "dynamic-vertex-parallel")		

CF	For Netflix and Netflix2x graphs
	caDirection(“s1”, "DensePull")
	caParallelization(“s1”, “edge-aware-dynamic-vertex-parallel")
	caNumSSG(“s1”, fixed-vertex-count, X)				

stalled. Interestingly, Gemini has the lowest QPI traffic, but is much slower than GraphIt. With NUMA optimizations, vertices in one socket fail to see the newly propagated labels from vertices in another socket, resulting in slower convergence. Unlike other NUMA-aware graph processing frameworks, GraphIt can easily enable or disable NUMA optimizations depending on the algorithm. We choose the label propagation algorithm option on Galois and use the FRONTIERS_WITHOUT_ASYNC option on Grazelle in order to compare the same algorithm across frameworks. Galois' CC is

Table 6.4: Line counts of PR, BFS, CC, and SSSP for GraphIt, Ligra, GraphMat, Green-Marl, Galois, Gemini, and Grazelle. Only Green-Marl has fewer lines of code than GraphIt. GraphIt has an order of magnitude fewer lines of code than Grazelle (the second fastest framework on the majority of the algorithms we measured). For Galois, we only included the code for the specific algorithm that we used. Green-Marl has a built-in BFS.

	GraphIt	Ligra	GraphMat	Green-Marl	Galois	Gemini	Grazelle
PR	34	74	140	**20**	114	127	388
BFS	22	30	137	**1**	58	110	471
CC	**22**	44	90	25	94	109	659
SSSP	**25**	60	124	30	88	104	

Table 6.5: Last-level cache (LLC) miss rate, QPI traffic, cycles with pending memory loads and cache misses, and parallel running time (seconds) of PR, CC, and PRDelta running on Twitter, and CF running on Netflix on the various frameworks, including GraphIt(GT), Ligra(LG), Gemini (GE), Grazelle(Gr)

Algorithm	PR				CC				PRDelta		CF	
Metrics	GT	LG	GE	GR	GT	LG	GE	GR	GT	LG	GT	LG
LLC miss rate (%)	**24.59**	60.97	45.09	56.68	**10.27**	48.92	43.46	56.24	**32.96**	71.16	**2.82**	37.86
QPI traffic (GB/s)	**7.26**	34.83	8.00	20.50	19.81	27.63	**6.20**	18.96	**8.50**	33.46	**5.68**	19.64
Cycle stalls (trillions)	**2.40**	17.00	3.50	4.70	**0.20**	0.96	1.20	0.30	**1.25**	5.00	**0.09**	0.22
Runtime (s)	**8.71**	49.00	10.98	15.70	**0.89**	2.78	3.85	1.73	**4.72**	9.19	**1.29**	5.35

35% faster than GraphIt on USAroad because it uses a special asynchronous engine instead of a frontier-based model. We also ran Galois's union-find CC implementation but found it to be slower than GraphIt on all graphs except USAroad. Grazelle's CC using the Vector-Sparse format, implemented with hundreds of lines of assembly code as shown in Table 6.4, is 43% faster than GraphIt on USAroad. The best performing schedule that we found on USAroad without any asynchronous mechanism is DensePush-SparsePush.

Collaborative Filtering (CF). For CF, GraphIt is faster than Ligra and GraphMat (by 4–4.8×) because the edge-aware-dynamic-vertex-parallel schedule achieves good load balance on Netflix. Cache optimization further improves GraphIt's performance and is especially beneficial on Netflix2x.

Single-Source Shortest Paths with Bellman-Ford (SSSP). GraphIt has the fastest SSSP on 4 out of the 5 graphs because of its ability to enable or disable the direction optimization and the bitvector representation of the frontier. We run Galois with the Bellman-Ford algorithm so that the algorithms are the same across systems. We also tried Galois's asynchronous SSSP but found it to be faster than GraphIt only on WebGraph. Green-Marl's SSSP on USAroad is 328 times slower than GraphIt because it uses the DensePush configuration. On every round, it must iterate through all vertices to check if they are active. This is expensive on USAroad because for over 6000 rounds, the active vertices count is less than 0.4% of all the vertices.

PageRank Delta (PRDelta). GraphIt outperforms Ligra on all graphs by 2–4× due to better locality from using bitvectors as frontiers, fusing the Delta and OutDegree arrays as shown in Figure 4-1, and applying both the cache and NUMA optimizations in the pull direction. Table 6.5 shows GraphIt's reduced LLC miss rate, QPI traffic, and cycles stalled.

Betweenness Centrality (BC). GraphIt achieves the fastest BC performance on the USAroad and Friendster graphs and has comparable performance on the other graphs. GraphIt is a bit slower than Gemini on Twitter and WebGraph because it does not support bitvector as a layout option for vertex data vectors layouts. We plan to add this in the future.

6.2.4 Performance of Different Schedules

Figure 6-2 demonstrates the impact of traversal direction, data structures used for keeping track of active vertices, and cache optimizations. For a given algorithm, no single schedule works well on all input graphs. For BFS, DensePullSparsePush with cache optimizations reduces the number of memory accesses on LiveJournal, Twitter, WebGraph, and Friendster, achieving up to 30× speedup. However, using only SparsePush can reduce the runtime overhead on USAroad as described in Section 6.2.3. For CC, the bitvector and cache optimizations improve locality of memory accesses for Twitter, WebGraph, and Friendster, but hurt the performance of LiveJournal and USAroad due to lower work-efficiency. For PRDelta, SparsePush sometimes outperforms DensePullSparsePush, but when the cache optimization is applied to the pull direction, hybrid traversal is preferred.

Figure 6-3 reveals that the parallelization scheme can have a major effect on scalability, and again there is no single scheme that works the best for all algorithms and inputs. For CF, the amount of work per vertex is proportional to the number of edges incident to that vertex. Consequently, the edge-aware-dynamic-vertex-parallel scheme is 2.4× faster than the dynamic-vertex-parallel approach because of better load balance. For PRDelta, the number of active vertices quickly decreases, and many of the edges do not need to be traversed. As a result, the edge-aware-dynamic-vertex-parallel scheme has a smaller impact on performance. The dynamic-vertex-parallel

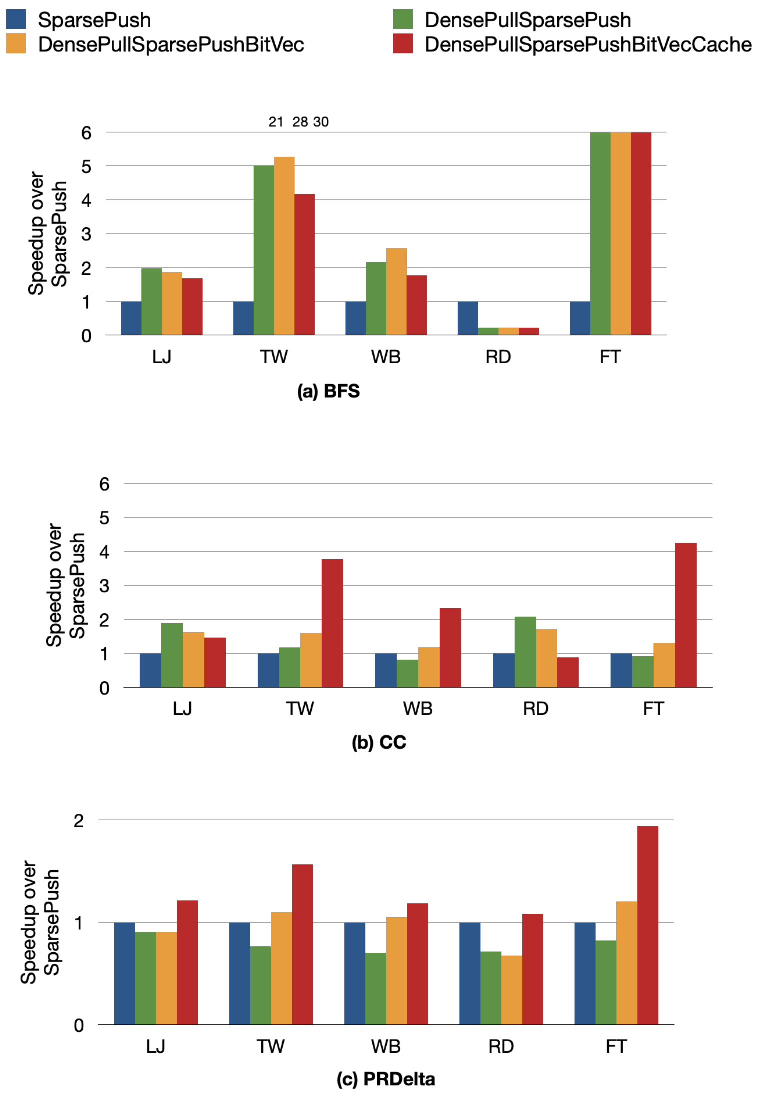


Figure 6-2: Performance of different schedules for BFS, CC, and PRDelta. SparsePush and DensePullSparsePush refer to the traversal directions. BitVec refers to the dense frontier data structure. Cache refers to the cache optimization. The descriptions of these schedules can be found in Section 3.2. The full scheduling commands are shown in Table 6.3.

approach is a good candidate for BFS because not all edges incident to a vertex are traversed. Using the edge-aware-dynamic-vertex-parallel scheme for BFS results in damaging the overall load balance. We omit the edge-parallel approach because it is consistently worse than edge-aware-dynamic-vertex-parallel due to the extra synchronization overhead.

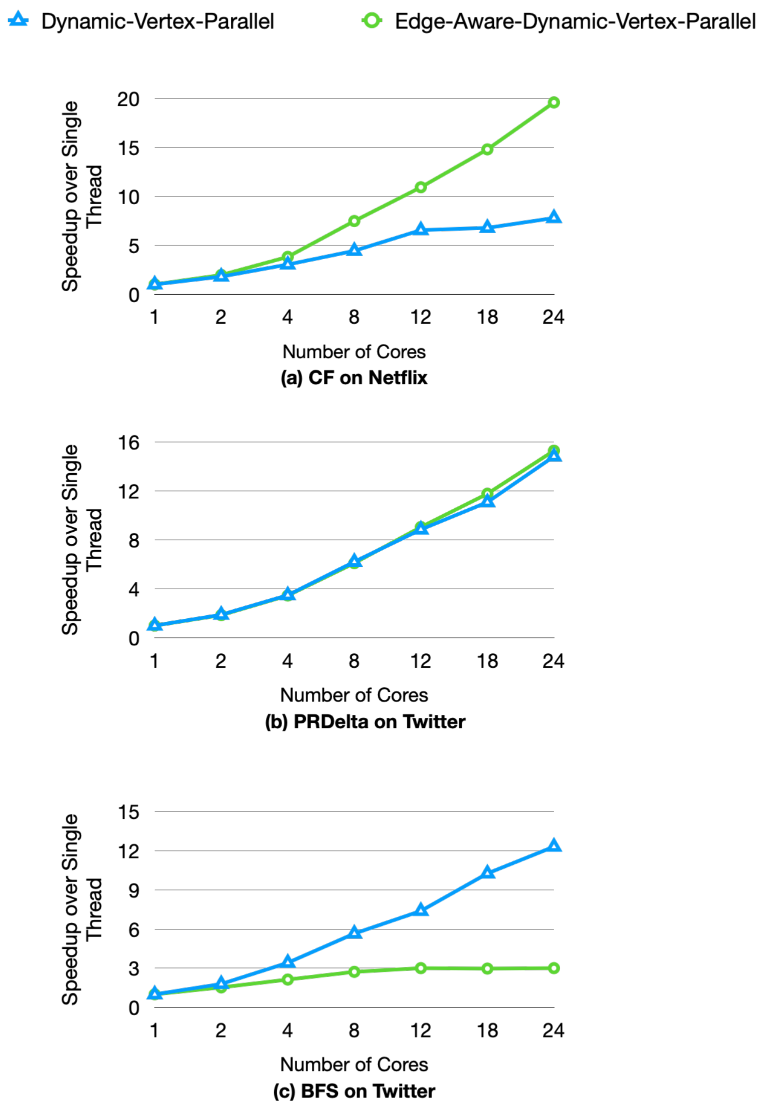


Figure 6-3: Scalability of CF, PRDelta, and BFS with different schedules. Hyper-threading is disabled.

6.2.5 Fusion of Multiple Graph Kernels

Figure 6-4 demonstrates the performance improvement of novel kernel fusion optimization (first introduced in Section 3.2.5) with PageRank and Eigenvector Centrality. They have similar memory access patterns. GraphIt significantly improves the spatial locality of the memory accesses by fusing together the two kernels and the vectors they access (vertex data layout optimization). Figure 6-4 shows significant reduction in cycles stalled on L1 data cache and L2 cache misses, leading to the speedups.

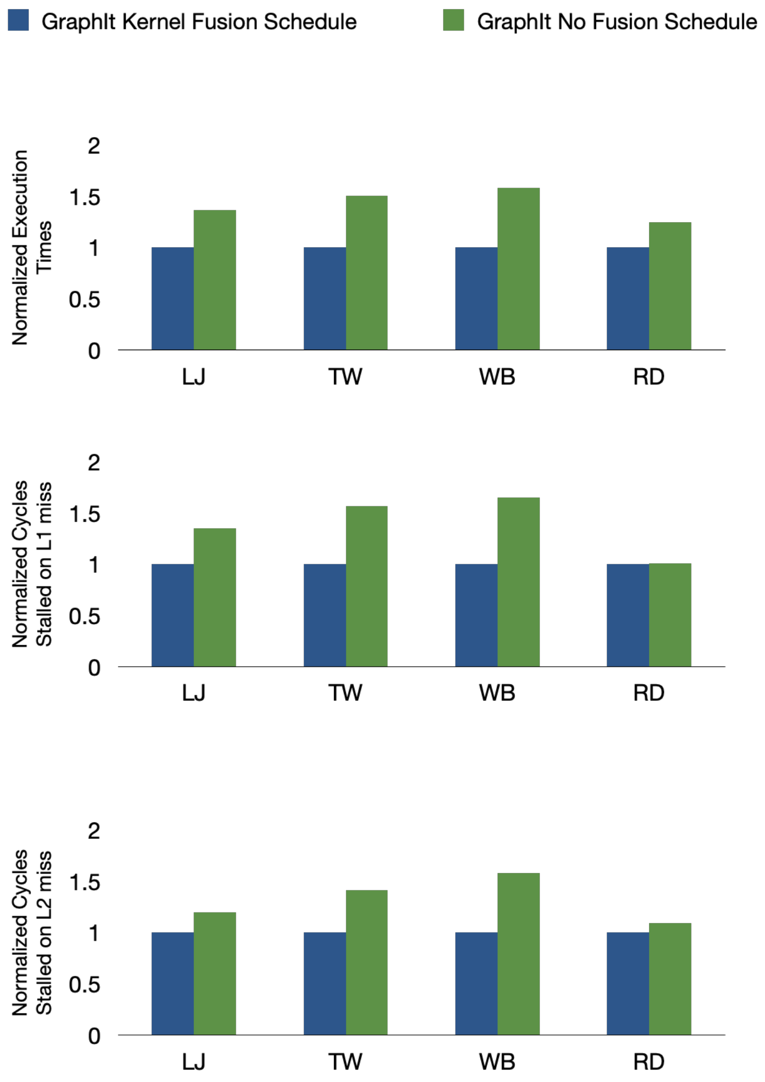


Figure 6-4: Normalized Execution Time, and L1 and L2 Cache Stall Cycles with Fusion of PageRank and Eigenvector Centrality

6.3 Ordered Graph Algorithms

We compare the performance of the new priority-based extension in GraphIt to state-of-the-art frameworks and analyze performance tradeoffs among different GraphIt schedules.

6.3.1 Applications

We evaluate the extension to GraphIt on SSSP with Δ-stepping, weighted breadth-first search (wBFS), point-to-point shortest path (PPSP), A^* search, k-core decomposition

(k-core), and approximate set cover (SetCover).

SSSP and Weighted Breadth-First Search (wBFS). SSSP with Δ-stepping solves the single-source shortest path problem as shown in Figure 2-3. In Δ-stepping, vertices are partitioned into buckets with interval Δ according to their current shortest distance. In each iteration, the smallest non-empty bucket i which contains all vertices with distance in $[i\Delta, (i+1)\Delta)$ is processed. wBFS is a special case of Δ-stepping for graphs with positive integer edge weights, with delta fixed to 1 [32]. We benchmarked wBFS on only the social networks and web graphs with weights in the range $[1, \log n)$, following the convention in previous work [32].

Point-to-point Shortest Path (PPSP). Point-to-point shortest path (PPSP) takes a graph $G(V, E, w(E))$, a source vertex $s \in V$, and a destination vertex $d \in V$ as inputs and computes the shortest path between s and d. In our PPSP application, we used the Δ-stepping algorithm with priority coarsening. It terminates the program early when it enters iteration i where $i\Delta$ is greater than or equal to the shortest distance between s and d it has already found.

A* Search. The A* search algorithm finds the shortest path between two points. The difference between A* search and Δ-stepping is that, instead of using the current shortest distance to a vertex as priority, A* search uses the *estimated distance* of the shortest path that goes from the source to the target vertex that passes through the current vertex as the priority. Our A* search implementation is based on a related work [3] and requires the longitude and latitude of the vertices.

***k*-core.** The k-core of an undirected graph $G(V, E)$ refers to a maximal connected sub-graph of G where all vertices in the sub-graph have induced-degree at least k. The k-core problem takes an undirected graph $G(V, E)$ as input and for each $u \in V$ computes the *maximum* k-core that u is contained in (this value is referred to as the coreness of the vertex) using a peeling procedure [64].

Approximate Set Cover. The Set Cover problem takes as input a universe $\mathcal{U}$ containing a set of ground elements, a collection of sets $\mathcal{F}$ s.t. $\cup_{f \in \mathcal{F}} f = \mathcal{U}$. The problem is to find the cheapest collection of sets $\mathcal{A} \subseteq \mathcal{F}$ that covers $\mathcal{U}$, i.e. $\cup_{a \in \mathcal{A}} a = \mathcal{U}$. In this thesis, we implement the unweighted version of the problem, where $c : \mathcal{F} \rightarrow 1$, but the algorithm used easily generalizes to the weighted case [32]. The algorithm at a high-level works by bucketing the sets based on their cost per element, i.e., the ratio of the number of uncovered elements they cover to their cost. At each step, a nearly-independent subset of sets from the highest bucket (sets with the best cost per element) are chosen, removed, and the remaining sets are reinserted into a bucket corresponding to their new cost per element. We refer to the following papers by Blelloch et al. [19, 20] for algorithmic details and a survey of related work.

6.3.2 Existing Frameworks

Galois v4 [74] uses approximate priority ordering with an ordered list abstraction for SSSP. We implemented PPSP and A* search using the ordered list. To the best of our knowledge and from communications with the developers, strict priority-based ordering is not currently supported for Galois. Galois does not provide implementations of wBFS, k-core and SetCover, which require strict priority ordering. In addition, GAPBS [14] is a suite of C++ implementations of graph algorithms and uses eager bucket update for SSSP, but does not provide implementations of k-core and SetCover. We used Julienne [32] from early 2019. The developers of Julienne have since incorporated the optimized bucketing interface proposed in this thesis in the latest version. GraphIt [116] and Ligra [91] are two of the fastest unordered graph frameworks. We used the best configurations (e.g., priority coarsening factor Δ and the number of cores) for the comparison frameworks. Schedules and parameters used are in the artifact.

6.3.3 Comparisons with Other Frameworks

Table 6.6 shows the execution times of GraphIt with the new priority-based extension and other frameworks. GraphIt outperforms the next fastest of Julienne, Galois, GAPBS, GraphIt, and Ligra by up to 3× and is no more than 6% slower than the fastest. GraphIt is up to 16.8× faster than Julienne, 7.8× faster than Galois, and 3.5× faster than hand-optimized GAPBS. Compared to unordered frameworks, GraphIt without the priority-based extension (unordered) and Ligra, GraphIt with

Table 6.6: Running time (seconds) of GraphIt with the priority-based extension and state-of-the-art frameworks. GraphIt, GAPBS, Galois, and Julienne use ordered algorithms. GraphIt with no extension (unordered) and Ligra use unordered Bellman-Ford for SSSP, PPSP, wBFS, and A* search, and unordered k-core. The fastest results are in bold. Graphs marked with † have weight distribution of $[1, \log n)$. Road networks come with original weights. Other graphs have weight distribution between $[1, 1000)$. $-$ represents an algorithm not implemented in a framework and **x** represents a run that did not finish due to timeout or out-of-memory error. ORD is for the ordered extension in GraphIt. UORD is used for the unordered algorithms for SSSP, PPSP, k-core, wBFS, and A* search in GraphIt and Ligra

Algorithm	SSSP							PPSP						
Graph	LJ	OK	TW	FT	WB	GE	RD	LJ	OK	TW	FT	WB	GE	RD
GraphIt (ORD)	**0.093**	**0.106**	3.09	**5.637**	**2.902**	**0.207**	**0.224**	0.04	**0.06**	**2.60**	**4.06**	**2.47**	**0.05**	**0.04**
GAPBS	0.10	0.11	3.55	6.09	3.30	0.59	0.77	**0.04**	0.06	2.71	4.31	2.63	0.12	0.11
Galois	0.12	0.23	**2.93**	8.00	3.01	0.24	0.28	0.08	0.17	2.63	7.09	2.61	0.06	0.05
Julienne	0.17	0.33	4.52	x	4.11	3.10	3.69	0.10	0.16	4.90	x	4.11	1.84	0.69
GraphIt (UORD)	0.22	0.48	6.38	38.46	8.52	90.52	122.37	0.22	0.48	6.38	38.46	8.52	90.52	122.37
Ligra (UORD)	0.301	0.60	7.78	x	x	94.16	129.20	0.301	0.60	7.78	x	x	94.16	129.20

Algorithm	k-core							Approximate Set Cover						
Graph	LJ	OK	TW	FT	WB	GE	RD	LJ	OK	TW	FT	WB	GE	RD
GraphIt (ORD)	**0.745**	**1.634**	**10.294**	**14.423**	**12.876**	**0.173**	**0.305**	**0.494**	**0.56**	**5.30**	**11.50**	**7.57**	**0.55**	**0.86**
GAPBS														
Galois														
Julienne	0.75	1.62	10.50	14.60	13.10	0.18	0.33	0.70	0.87	6.89	13.20	10.70	0.66	1.03
GraphIt (UORD)	6.13	8.15	228.11	325.29	x	0.42	1.76							
Ligra (UORD)	5.99	8.09	225.10	324.00	x	0.71	1.76							

Algorithm	wBFS					A* search		
Graph	LJ†	OK†	TW†	FT†	WB†	MA	GE	RD
GraphIt (ORD)	**0.07**	**0.10**	**1.82**	**7.56**	**2.13**	**0.01**	**0.06**	**0.075**
GAPBS	0.07	0.11	1.90	7.88	2.23	0.03	0.14	0.22
Galois	0.13	0.25	2.76	5.75	2.67	0.08	0.07	0.08
Julienne	0.15	0.15	2.32	x	2.81	0.18	1.55	4.88
GraphIt (UORD)	0.12	0.20	2.52	21.77	3.66	0.46	90.52	122.37
Ligra (UORD)	0.16	0.26	3.05	x	x	0.83	94.16	129.20

the extension achieves speedups between 1.67× to more than 600× due to improved algorithm efficiency. The times for SSSP and wBFS are averaged over 10 starting vertices. The times for PPSP and A* search are averaged over 10 source-destination pairs. We chose the start and end points to have a balanced selection of different distances. We also show a heatmap of performance comparisons among the ordered extension of GraphIt, Julienne, and Galois in Figure 6-5.

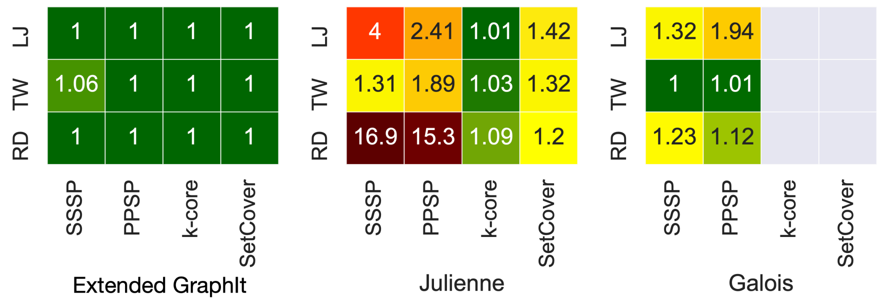


Figure 6-5: A heatmap for the performances of ordered extension of GraphIt, Julienne, and Galois. We benchmarked with SSSP (delta stepping), PPSP (delta stepping), k-core, and Approximate SetCover. The faster numbers are in green and the numbers are slowdowns compared to the fastest of the three frameworks.

GraphIt with the priority extension has the fastest SSSP performance on six out of the seven input graphs. Julienne uses significantly more instructions than GraphIt (up to 16.4× more instructions than GraphIt). On every iteration, Julienne computes an out-degree sum for the vertices on the frontier to use the direction optimization, which adds significant runtime overhead. GraphIt avoids this overhead by disabling the direction optimization with the scheduling language. Julienne also uses the lazy bucket update that generates extra instructions to buffer the bucket updates whereas GraphIt reduces instructions by using the eager bucket update. GraphIt is faster than GAPBS because of the bucket fusion optimization that allows GraphIt to process more vertices in each round and use fewer rounds (details are shown in Table 6.8). The optimization is especially effective for road networks, where the synchronization overhead is a significant performance bottleneck. Galois achieves good performances on SSSP because it does not have as much overhead from global synchronization needed to enforce strict priority. However, it is slower than GraphIt on most graphs because approximate priority ordering sacrifices some work-efficiency.

GraphIt with the priority extension is the fastest on most of the graphs for PPSP, wBFS, and A* search, which use a variant of the Δ-stepping algorithm with priority coarsening. Both GraphIt and GAPBS use eager bucket update for these algorithms. GraphIt outperforms GAPBS because of bucket fusion. Galois is often slower than GraphIt due to lower work-efficiency with the approximate priority ordering. Julienne

Table 6.7: Line counts of single-source shortest paths with Δ-stepping (SSSP), point-to-point shortest path with Δ-stepping (PPSP), A* search, k-core, and SetCover for GraphIt, GAPBS, Galois, and Julienne. The missing numbers correspond to a framework not providing an algorithm.

	GraphIt with extension	GAPBS	Galois	Julienne
SSSP	**28**	77	90	65
PPSP	**24**	80	99	103
A*	**74**	105	139	84
KCore	**24**	–	–	35
SetCover	**70**	–	–	72

uses the lazy bucket update and is slower than GraphIt due to the runtime overheads of the lazy approach.

PPSP and A* search are faster than SSSP because they only run until the distance to the destination vertex is finalized. A* search is sometimes slower than PPSP because of additional random memory accesses and computation needed to estimate distances to the destination.

For k-core and SetCover, the extended GraphIt runs faster than Julienne because the optimized lazy bucketing interface uses the priority vector to compute the priorities of each vertex. Julienne uses a UDF to compute the priority every time, resulting in function call overheads and redundant computations. Galois does not provide ordered algorithms for k-core and SetCover, which require strict priority and synchronizations after processing each priority.

Delta Selection for Priority Coarsening. The best Δ value for each algorithm depends on the size and the structure of the graph. The best Δ values for social networks (ranging from 1 to 100) are much smaller than deltas for road networks with large diameters (ranging from 2^{13} to 2^{17}). Social networks need only a small Δ value because they have ample parallelism with large frontiers and work-efficiency is more important. Road networks need larger Δ values for more parallelism. We also tuned the Δ values for the comparison frameworks to provide the best performance.

Line Count Comparisons. Table 6.7 shows the line counts of the five graph algorithms implemented in four frameworks. GAPBS, Galois, and Julienne all require the

Table 6.8: Running times and number of rounds reductions with the bucket fusion optimization on single-source shortest paths (SSSP) using Δ-stepping.

Datasets	with Fusion	without Fusion
TW	**3.09s** [1025 rounds]	3.55s [1489 rounds]
FT	**5.64s** [5604 rounds]	6.09s [7281 rounds]
WB	**2.90s** [772 rounds]	3.30s [2248 rounds]
RD	**0.22s** [1069 rounds]	0.77s [48407 rounds]

Table 6.9: Performance Impact of Eager and Lazy Bucket Updates. Lazy update for k-core uses constant sum reduction optimization.

	k-core		SSSP with Δ-stepping	
Datasets	Eager Update	Lazy Update	Eager Update	Lazy Update
LJ	0.84	**0.75**	**0.093**	0.24
TW	44.43	**10.29**	**3.09**	6.66
FT	46.59	**14.42**	**5.64**	10.34
WB	35.58	**12.88**	**2.90**	7.82
RD	0.55	**0.31**	**0.22**	9.48

programmer to take care of implementation details such as atomic synchronization and deduplication. GraphIt uses the compiler to automatically generate these instructions. For A* search and SetCover, GraphIt needs to use long `extern` functions that significantly increases the line counts.

6.3.4 Scalability Analysis

We analyze the scalability of different frameworks in Figure 6-6 for SSSP on social and road networks. The social networks (TW and FT) have very small diameters and large numbers of vertices. As a result, they have a lot of parallelism in each bucket, and all three frameworks scale reasonably well (Figure 6-6(a) and (b)). Compared to GAPBS, GraphIt uses bucket fusion to significantly reduce synchronization overheads and improves parallelism on the RoadUSA network (Figure 6-6(c)). GAPBS suffers from NUMA accesses when going beyond a single socket (12 cores). Julienne's overheads from lazy bucket updates makes it hard to scale on the RoadUSA graph.

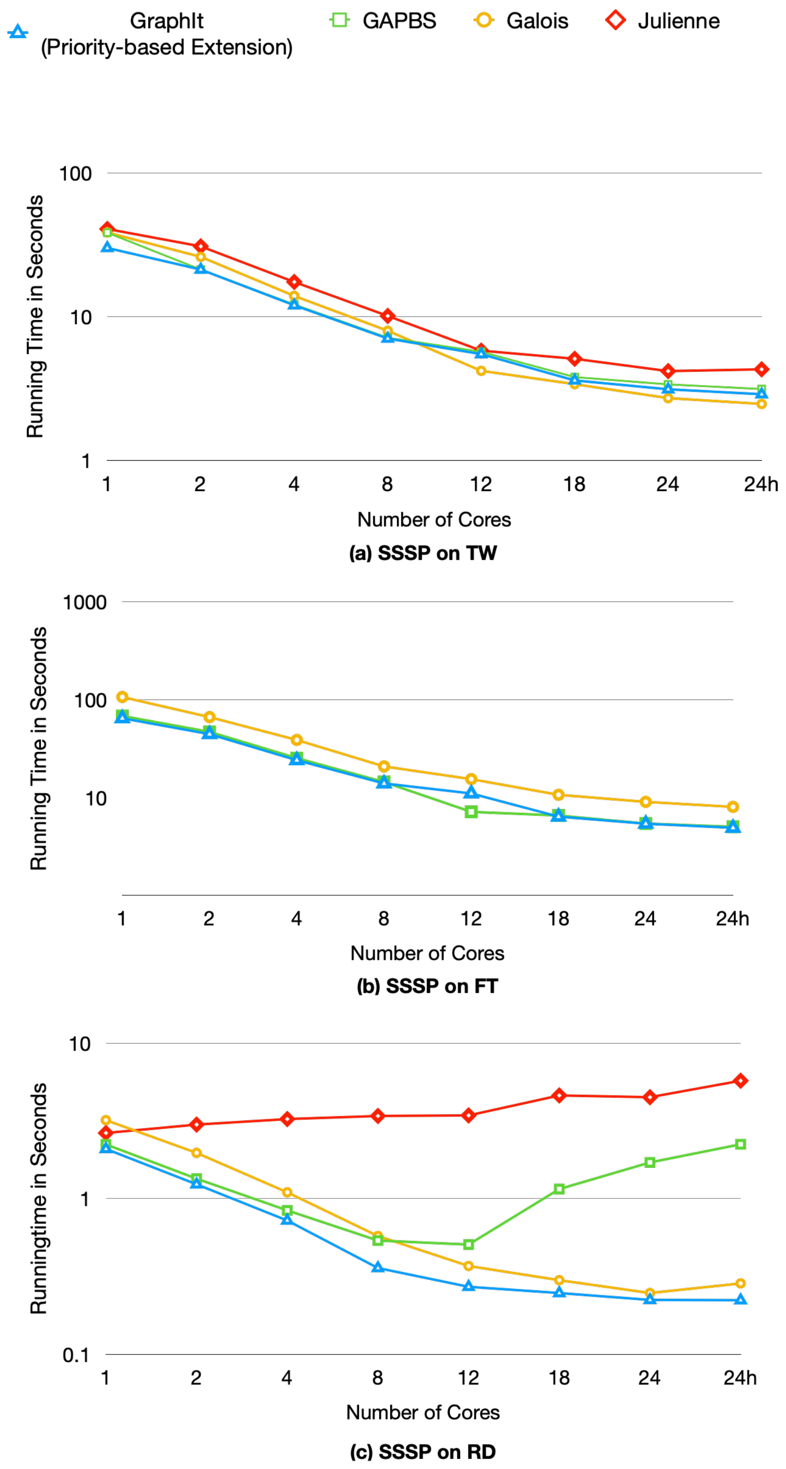
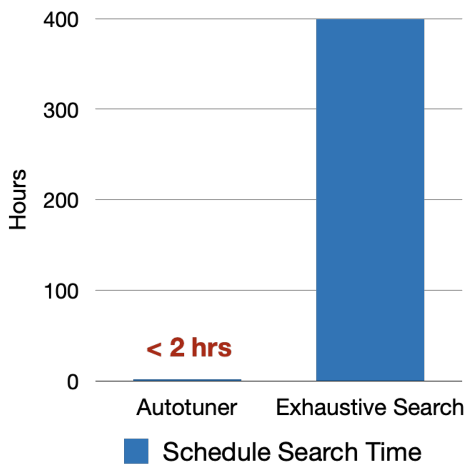


Figure 6-6: Scalability of different frameworks on single-source shortest paths with Δ-stepping (SSSP).

Figure 6-7: Time to find high-performance schedules with autotuninng and exhaustive search for large social networks.



6.3.5 Performance of Different Schedules

Table 6.8 shows that SSSP with the novel bucket fusion optimization (first introduced in Section 3.4.3) achieves up to 3.4× speedup over the version without bucket fusion on road networks, where there are a large number of rounds processing each bucket. Table 6.8 shows that the optimization improves running time by significantly reducing the number of rounds needed to complete the algorithm.

Table 6.9 shows the performance impact of eager versus lazy bucket updates on k-core and SSSP. k-core does a large number of redundant updates on the priority of each vertex. The priority of each vertex is updated the same number of times as its out-degree. In this case, using the lazy bucket approach drastically reduces the number of bucket insertions. Additionally, with a lazy bucket approach, we can also buffer the priority updates and later reduce them with a histogram approach (lazy with constant sum reduction optimization). This histogram-based reduction avoids overhead from atomic operations. For SSSP there are few redundant updates and the lazy bucket approach introduces significant runtime overhead over the eager bucket approach.

Table 6.10: Autotuned Performances for PageRank (PR), Breadth-First Search(BFS), Connected Components with Label Propagation (CC), SSSP (Bellman-Ford), PageRankDelta (PRDelta), and Collaborative Filtering (CF), compared with Handtuned Performances on the 24-core CPUs. The algorithms are autotuned for a maximum of 5000 seconds. Less than one means that the autotuned version is faster than the handtuned version.

PR					
Graph	LJ	TW	WB	RD	FT
GraphIt (handtuned)	0.34s	8.71s	16.39s	0.91s	32.57s
GraphIt (autotuned)	0.34s	8.5s	15.97s	0.82s	29.19s
autotuned / handtuned	1.0	0.98	0.97	0.9	0.9
BFS					
Graph	LJ	TW	WB	RD	FT
GraphIt (handtuned)	0.03s	0.28s	0.65s	0.22s	0.49s
GraphIt (autotuned)	0.03s	0.31s	0.63s	0.22s	0.51s
autotuned / handtuned	1.0	1.10	0.98	1.0	1.04
CC					
Graph	LJ	TW	WB	RD	FT
GraphIt (handtuned)	0.07s	0.89s	1.96s	17.10s	2.63s
GraphIt (autotuned)	0.06s	0.91s	1.99s	17.10s	2.57s
autotuned / handtuned	0.85	1.02	1.02	1.0	0.98
SSSP					
Graph	LJ	TW	WB	RD	FT
GraphIt (handtuned)	0.06s	1.35s	1.68s	0.29s	4.30s
GraphIt (autotuned)	0.06s	1.32s	1.61s	0.29s	4.08s
autotuned / handtuned	1.0	0.98	0.96	1.0	0.95
PRDelta					
Graph	LJ	TW	WB	RD	FT
GraphIt (handtuned)	0.17s	4.72s	7.14s	0.46s	10.15s
GraphIt (autotuned)	0.17s	4.90s	7.70s	0.40s	11.20s
autotuned / handtuned	1.0	1.04	1.08	0.87	1.10
CF					
Graph	NX	NX2			
GraphIt (handtuned)	1.26s	4.58s			
GraphIt (autotuned)	1.29s	4.61s			
autotuned / handtuned	1.02	1.01			

6.4 Autotuning

The autotuner finds high-performance schedules much faster than an exhaustive search baseline as shown in Figure 6-7. The autotuner is able to determine a high-performance configuration usually within 10 - 15 trails. The time used to tune the integer parameters is also included. We are able to speedup the process by setting appropriate time limits for each run and by using binary-serialized graph formats that are much smaller than text-based formats to reduce the graph loading time.

Table 6.10 shows performance of autotuned implementations versus handtuned implementations for various algorithms running on both social and road networks. The autotuner for GraphIt is able to automatically find schedules that performed within 10% of the handtuned schedules used for Table 6.2 and Table 6.6. The autotuned schedules are sometimes slower because the autotuner did not find the optimal integer parameter for segmentation. The autotuned versions are faster in some cases by discovering better schedules, such as a better direction than the one we selected by hand.

Details of the search space are described in Section 5.2.7. For most graphs, the autotuner can find a high-performance schedule within 300s after trying 30-40 schedules (including tuning integer parameters) in a large space of about 10^6 schedules. The autotuning process finished within 5000 seconds for the largest graphs. Users can specify a time limit to reduce autotuning time.

6.5 Chapter Summary

In this chapter, we demonstrated that GraphIt achieves high performance across different types of graphs and algorithms on modern multi-core CPUs, whereas existing graph processing frameworks achieve good performance only on a subset of graphs and algorithms. This is because GraphIt supports a much larger space of performance optimizations with a new compiler approach. Furthermore, GraphIt also improves the performance of graph algorithms over state-of-the-art frameworks by up to $4.8\times$ by discovering previously unexplored combinations of optimizations and utilizing newly proposed novel cache and bucket fusion optimizations. In Chapter 8, we provide a more comprehensive review of the related optimizations supported by different frameworks.

Chapter 7

Limitations and Future Work

In this chapter, we discuss the limitations of the current GraphIt DSL and compiler and propose a few future research directions. We focus on the algorithms that are currently not supported in GraphIt due to a lack of relevant programming operators in Section 7.1. We outline a few optimizations that can potentially be integrated into GraphIt in Section 7.2. In Section 7.3, we describe potential approaches to support more hardware platforms, such as GPUs. Section 7.4 maps out ways that can further improve the extensibility of the compiler.

7.1 Support for More Algorithms

In this section, we outline a few categories of graph algorithms that GraphIt can be extended to support, including algorithms that modify the graph, sampling-based algorithms, hypergraph algorithms, and graph neural networks (GNNs).

Algorithms that Modify the Graph. There are many algorithms that require modifications to the graph, such as Kruskal's algorithm [54] for finding the minimum-spanning-tree and streaming graph algorithms [33, 34]. Currently GraphIt does not have operators that make it easy to implement graph-modification algorithms. Potential operators include removing and adding edges and vertices in the graph.

Sampling-based Algorithms. Sampling-based algorithms have been shown to perform well on certain applications, such as connected components [96] and approximate triangle counting [89]. Currently, GraphIt does not support sampling-based algorithms. It is possible to implement operators that generate sampled subsets of edge and vertex sets.

Hypergraph Algorithms. Hypergraphs are made up of hyperedges, which can contain multiple vertices. Hypergraphs can better represent some networked data because they preserve some additional information in the original data [90]. It is interesting to explore how to design hyperedge processing algorithmic operators in GraphIt and their corresponding scheduling APIs.

Graph Neural Networks. Graph Neural Networks (GNNs) [106] are particularly useful in a number of graph-based applications. GraphIt currently does not have an easy way to support these applications. To support GNNs, we must add operators to support sampling edges adjacent to each vertex. Additionally, we will also need to add support for matrix operations that are sometimes required to process the aggregated input. This requires GraphIt to efficiently support a linear algebra library within the DSL. Otherwise, GraphIt needs to support efficient data transfer between the DSL and other external linear algebra packages such as NumPy.

Other Algorithms. It is also interesting to explore how to support other algorithms, such as community detection, solvers, Delaunay Triangulation [21], and support for sparse and dense linear algebra.

7.2 Support for More Optimizations

In this section, we describe a few promising directions to further expand the space of optimizations (schedules) for GraphIt, including vectorization and new graph formats.

Vectorization. Vectorization can significantly improve the performance of certain graph applications, such as PageRank running on the road networks [39]. To support the vectorization optimization, the compiler needs to transform both the UDFs and the underlying graph to a more vectorization-friendly format.

Graph Formats. Different graph formats can potentially improve the performance of certain applications. We support the cache-optimized segmented CSR format described in Sectioin 3.2.3. However, more sparse matrix formats [25] exist that can be used to express the adjacency matrix of the graph, which can potentially improve the performance of certain graph applications.

7.3 Support for More Hardware Platforms

No single hardware platform performs best for all graph applications. Some applications perform better on CPUs and others perform better on GPUs or domain-specific accelerators (DSAs). Shared-memory CPUs have out-of-order execution, which helps hide the long latency of irregular memory accesses that miss in the last-level cache. CPUs also have larger memories than GPUs and other accelerators, which enables processing of larger graphs. By contrast, GPUs have up to an order of magnitude more compute power and memory bandwidth than CPUs and can better exploit the data parallelism of some graph programs when the graph fits in the GPU memory.

Many DSAs for sparse computations have emerged recently [41, 49, 59, 26, 72, 4, 110, 5]. They provide hardware features such as efficient speculative execution that can drastically improve graph performance. However, the program must be transformed specifically for each DSA to take advantage of the new hardware features. Yet, it is infeasible to build a new compiler for each DSA. Thus, building a portable compiler infrastructure is crucial to exploiting the diverse hardware of different DSAs.

Distributed backends, such as distributed CPUs and GPUs, are also important potential extensions for GraphIt. We can leverage distributed systems to process large graphs that do not fit in the memory of a single CPU or GPU.

It is important to redesign the compiler to make it easy to support a new hardware backend. We are investigating new designs for the scheduling language and internal intermediate representations.

7.4 Flexible and Extensible Compiler Infrastructure

In this section, we identify a few directions that the compiler infrastructure can be further improved, such as extending the algorithm and scheduling language with more operators and optimizations.

Support for New Optimizations (Schedules). It is useful to figure out how to add a new optimization or schedule into the compiler without having to rewrite a significant portion of the compiler. Currently, developers can extend the scheduling space by plugging in their optimized runtime libraries with minimum modifications to the compiler. However, this approach only works for optimizations that do not require additional transformations on the program.

Support for New Algorithm Operators. Currently, we have designed the DSL to enable easy integration of additional algorithmic operators through the runtime libraries. However, the scheduling language cannot tune these operators easily because their implementations rely on the fixed runtime libraries. In the future, it will be interesting to figure out a modularized approach to add a new algorithmic operator and the corresponding schedules that can tune the new operator.

Embedding the DSL in Python or C++. Many graph applications are only a part of a larger data analytics and machine learning pipeline. Thus, GraphIt programs must interact with existing libraries and other parts of the pipeline. Currently, GraphIt is a standalone programming language that generates functions that can be used from Python or C++. Embedding GraphIt directly in Python or C++ is likely to provide better integration with the existing programs, libraries, and other parts of the data analytics pipeline.

Automatic Schedule Generation beyond Autotuning. The current autotuner in GraphIt is still relatively slow and requires the input data. A more advanced autotuner should be able to sample the input graph, encode the program, and use machine learning techniques to quickly recommend a set of schedules statically without running many different configurations with the input.

Chapter 8

Related Work

Graph processing is a well-studied field. In this chapter, we give a detailed overview of related work in the space. We first discuss the differences between GraphIt and existing graph processing frameworks and DSLs in Section 8.1. In Section 8.2, we highlight some performance optimizations that are closely related to the ones we proposed in Chapter 3. Finally, we list some DSLs in other domains that inspired our work on GraphIt in Section 8.3.

8.1 Graph Processing Frameworks and DSLs

A large body of related work exists in graph processing. In this section, we describe the relationships between GraphIt and the previous work. We discuss shared-memory, out-of-core, and distributed graph processing frameworks. We also survey new graph frameworks that support ordered graph algorithms.

8.1.1 Shared-Memory Unordered Graph Processing Frameworks

Many high-performance graph frameworks and DSLs, including GraphIt, optimize their performance for shared-memory systems. Many of these frameworks support only a limited set of combinations of optimization techniques as shown in Table 8.1 (these optimizations are described in Chapter 3). *GraphIt significantly expands the*

Table 8.1: Optimizations adopted by various frameworks (explained in Section 3.1): WSVP (work-stealing vertex-parallel), WSEVP (work-stealing edge-aware vertex-parallel), SPVP (static-partitioned vertex-parallel with no work-stealing), EP (edge-parallel), BA (dense boolean array), BV (dense bitvector), AoS (Array of Structs), SoA (Struct of Arrays), SPS (SparsePush), DPS (DensePush), SP (SparsePull), DP (DensePull), SPS-DP (hybrid direction with SPS and DP depending on frontier size), DPS-SPS (hybrid with DPS and SPS), DPS-DP (hybrid with DPS and DP).

Frameworks	Traversal Directions	Dense Frontier Data Layout	Parallel Opt.	Vertex Data Layout	Cache Opt.	NUMA Opt.	Opt. Combinations Count	Integer Params Count
GraphIt	SPS, DPS, SP, DP, SPS-DP, DPS-SPS	BA, BV	WSVP, WSEVP, SPVP, EP	AoS, SoA	Partitioned, No Partition	Partitioned, Interleaved	**100+**	3
Ligra	SPS-DP, DPS-SPS	BA	WSVP, EP	SoA	None	Interleaved	4	1
Green-Marl	DPS, DP	BA	WSVP	SoA	None	Interleaved	2	0
GraphMat	DPS, DP	BV	WSVP	AoS	None	Interleaved	2	0
Galois	SPS, DP, SPS-DP	BA	WSVP	AoS	None	Interleaved	3	1
Polymer	SPS-DP, DPS-SPS	BA	WSVP	SoA	None	Partitioned	2	0
Gemini	SPS, DP, SPS-DP	BA,BV	WSVP	SoA	None	Partitioned	6	1
GraphGrind	SPS-DP, DPS-SPS	BA	WSVP	SoA	None	Partitioned, Interleaved	4	1
Grazelle	DPS, DP, DPS-DP	BV	EP	SoA	None	Partitioned	3	1

space of optimizations by composing numerous effective optimizations, supporting two orders of magnitude more optimization combinations than existing frameworks. GraphIt achieves high performance by enabling programmers to easily find the best combination of optimizations for their specific algorithm and input graph. GraphIt also finds previously unexplored combinations of optimizations to significantly outperform the state-of-the-art frameworks on many algorithms.

Many shared-memory graph systems, such as Ligra [91], Gunrock [103], GraphGrind [94], Polymer [112], Gemini [118] and Grazelle [39], adopt the frontier-based model. Galois [74] also has an implementation of the model and a scheduler that makes it particularly efficient for road graphs. The frontier-based model [91] operates efficiently on subsets of vertices (frontiers) and their outgoing edges using the direction optimization [11]. Flat data-parallel operators are used to apply functions to the frontier vertices and their neighbors with parallel direction optimizations. Existing frameworks only support up to three of the many possible directions, with little support for different parallelization schemes and frontier and vertex data layout optimizations. GraphIt significantly expands the space of optimizations by enabling combinations of data layout optimization, different direction choices, and various parallelization schemes (Table 8.1). GraphIt also makes programming easier by freeing the programmer from specifying low-level implementation details, such as updated vertex tracking and atomic synchronizations.

Many frameworks and techniques have been introduced to improve locality with NUMA and cache optimizations. GraphGrind, Grazelle, Gemini and Polymer all support NUMA optimizations. CSR Segmenting [115] and cache blocking [75, 13] have been introduced to improve the cache performance of graph applications through graph partitioning. However, both techniques have not been integrated into a general programming model or combined with direction optimizations. GraphIt supports NUMA optimizations and integrates a simplified variant of CSR segmenting to compose cache optimizations with other optimizations.

Other shared-memory systems [95, 108] adopt the vertex-centric model to exploit data parallelism across vertices. Programmers specify the logic that each (active) vertex executes iteratively. Frameworks [52, 62] use sparse matrix-vector multiplication with semirings to express graph algorithms. However, both programming models cannot easily integrate direction optimization, which requires different synchronization strategies for each vertex in the push and pull directions.

Green-Marl [46], Socialite [57], Abelian [37], and EmptyHeaded [2] are DSLs for shared-memory graph processing. Green-Marl provides a BFS primitive; thus, programs that can be expressed with BFS invocations are relatively concise. However, for other graph programs, the programmer must write the loops over vertices and edges explicitly, making it hard to integrate direction optimization due to the lower level nature of the language. Socialite and EmptyHeaded provide relational query languages to express graph algorithms. The underlying data representation is in the form of tables, and due to extensive research in join optimizations, these systems perform especially well for graph algorithms that can be expressed efficiently using joins (e.g., subgraph finding). However, because these languages do not allow for explicit representation of active vertex sets, their performance on graph traversal algorithms is worse than the frontier-based frameworks [86, 2]. These DSLs also do not support the composition of optimizations or enable extensive performance tuning capabilities.

A number of graph processing frameworks have been developed for GPUs (see [88] for a survey). We did not focus on GPUs in this paper as the current GPU memory capacities do not allow us to process very large graphs in-memory.

8.1.2 Out-of-Core Graph Processing Frameworks

A significant amount of work has dealt with graphs that cannot fit in memory (e.g., [56, 117, 83, 119, 63, 51, 99, 101, 100, 108]), whereas GraphIt focuses on in-memory graph processing. Some of the optimizations in out-of-core systems also focus on improving locality of accesses, parallelism, and work-efficiency, but the tradeoff space for these techniques is very different when optimizing for the disk/DRAM boundary, instead of the DRAM/cache boundary. The higher disk access latency, lower memory bandwidth, and larger granularity of access lead to vastly different techniques [51]. When the graphs do fit in memory, out-of-core systems, such as X-Stream [83], are much slower than shared-memory frameworks [115, 39].

8.1.3 Distributed Graph Processing Frameworks

Graph analytics has also been studied extensively in distributed memory systems (e.g., [61, 38, 79, 82, 24, 85, 118, 66, 108, 27]). The tradeoff space is also different for distributed graph processing systems due to the larger network communication overhead and greater need for load balance. Techniques used by GraphIt, such as direction optimization and locality enhancing graph partitioning can also be applied in the distributed domain [118]. These systems, when ran on a single machine, cannot generally outperform shared-memory frameworks [86].

8.1.4 Ordered Graph Frameworks

A significant amount of work has been conducted on unordered graph processing frameworks (e.g., [91, 27, 118, 39, 115, 56, 42, 79, 85, 102, 38, 95, 104, 68, 107, 97, 78, 116, 92, 71, 73, 33], among many others). These frameworks do not have data structures and operators to support efficient implementations of ordered algorithms, and cannot support a wide selection of ordered graph algorithms. A few unordered frameworks [104, 68, 95] have the users define functions that filter out vertices to support Δ-stepping for SSSP. This approach is inefficient and does not generalize to other ordered algorithms. Wonderland [113] uses abstraction-guided priority-based scheduling to reduce the total number of iterations for some graph algorithms. However, it requires preprocessing and does not implement a strict ordering of the ordered graph algorithms. PnP [107] proposes direction-based optimizations for point-to-point queries, which is orthogonal to the optimizations in this paper, and can be combined together to potentially achieve even better performance. GraphIt [116] decouples the algorithm from optimizations for unordered graph algorithms. This thesis introduces new priority-based operators to the algorithm language, proposes new optimizations for the ordered algorithms in the scheduling language, and extends the compiler to generate efficient code.

8.2 Other Performance Optimizations

We have described a number of performance optimizations in Chapter 3. Here we list a few related performance optimizations.

Frequency-based Clustering. Graph reordering has become an important class of performance optimizations for graph applications. Recent work [10] proposed a new reordering technique, hub clustering, which has lower overhead than frequency-based clustering while maintaining similar performance improvements. The same work also performed a comprehensive study on various graph reordering techniques. Frequency-based clustering is also similar to the Hilbert curve orderings for graph and sparse matrix data [111, 40, 67], which sort the edges to achieve locality in both the source and destination vertices of nearby edges. While our technique can only guarantee locality in one of these (either sources or destinations), we found that it performs slightly better than Hilbert curve order on a single thread and is easier to scale to a multicore (the Hilbert ordering may require multiple threads to update the same vertex, which needs atomic writes or private vectors, whereas our method allows purely “pull-based” updates where only one thread is writing to each output vector location). Threads in a Hilbert order also compete for the shared LLC to fit each thread’s private working sets, unlike our method in which all threads read from the LLC shared data.

Bucketing. Bucketing is a common way to exploit parallelism and maintain ordering in ordered graph algorithms. It is expressive enough to implement many parallel ordered graph algorithms [32, 14]. Existing frameworks support either the lazy bucket update or the eager bucket update approach. GAPBS [14] is a suite of hand-optimized C++ programs that includes SSSP using the eager bucket update approach. Julienne [32] is a high-level programming framework that uses the lazy bucket update approach, which is efficient for applications that have a lot of redundant updates, such as k-Core and SetCover. However, it is not as efficient for applications that have fewer redundant updates and less work per bucket, such as SSSP and A* search. GraphIt with the priority-based extension unifies both the eager and lazy bucket update approaches with a new programming model and compiler extensions to achieve consistent high performance.

Speculative Execution. Speculative execution can also exploit parallelism in ordered graph algorithms [44, 45]. This approach can potentially generate more parallelism as vertices with different priorities are executed in parallel as long as the dependencies are preserved. This is particularly important for many discrete simulation applications that lack parallelism. However, speculative execution in software incurs significant performance overheads as a commit queue has to be maintained, conflicts need to be detected, and values are buffered for potential rollback on conflicts. Hardware solutions have been proposed to reduce the overheads of speculative execution [49, 93, 48, 50, 3], but it is costly to build customized hardware. Furthermore, some ordered graph algorithms, such as approximate set cover and k-core, cannot be easily expressed with speculative execution.

Approximate Priority Ordering. Some work has disregarded a strict priority ordering and use an approximate priority ordering [74, 27, 7, 6]. This approach uses several “relaxed" priority queues in parallel to maintain local priority ordering. However, it does not synchronize globally among the different priority queues. To the best of our knowledge and from communications with the developers, Galois [74, 27] does not currently support strict priority ordering and only supports an approximate ordering. Galois [74] provides an ordered list abstraction, which does not explicitly synchronize after each priority. Therefore, it is difficult to implement algorithms that require explicit synchronization, such as k-core. Galois also require users to handle atomic synchronizations for correctness. This approach cannot implement certain ordered algorithms that require strict priority ordering, such as work-efficient k-core decomposition and SetCover. Moreover, D-galois [28] implements k-core for only a specific k, whereas GraphIt's k-core finds *all* cores.

Synchronization Relaxation. Several frameworks relax synchronizations in graph algorithms for better performance while preserving correctness [43, 98, 16]. Compared to existing synchronization relaxation work, bucket fusion in our new priority-based extension is more restricted regarding synchronization relaxation. The synchronization between rounds can be removed only when the vertices processed in the next round have the same priority as vertices processed in the current round. This way, we ensure no priority inversion happens.

8.3 DSLs in other domains

The design to separate the algorithm specification from the performance optimizations in GraphIt is largely inspired by Halide. The main difference is that GraphIt is focused on the sparse graph computations, whereas Halide is primarily designed for the dense image processing computations. Many DSLs incorporated a scheduling language, such as Halide [80], CHILL [23], Tiramisu [9], Taichi [47], and HMPP [81].

These languages, with the exception of Taichi, mainly focus on loop nest optimization in applications that manipulate dense arrays. Unlike these scheduling languages, the GraphIt scheduling language is designed for sparse graph applications. It is the first scheduling language designed to address the challenges of graph applications, graph data structures, and graph optimizations. It allows the programmer to perform data layout transformations and allows full separation between the algorithm and the schedule, which is not possible in HMPP. Full separation is an important feature that results in higher portability across different hardware architectures as shown in [80]. Unlike CHiLL, which was primarily designed for the application of affine transformations on loop nests, the GraphIt scheduling language supports a large set of non-affine transformations, which are the main type of optimizations in the context of graph applications.

Physical Simulation DSLs. GraphIt is heavily influenced by DSLs for physical simulations, including Simit [53] and Liszt [31]. The data model of GraphIt is inspired by the data model in Simit. The dependence analysis in GraphIt is similar to the one employed in Liszt. However, Simit and Liszt do not support efficient filtering on vertices and edges and do not have a scheduling language.

Chapter 9

Conclusion

We have described GraphIt, a novel DSL for graph processing that generates fast implementations for algorithms with different performance characteristics running on graphs with varying sizes and structures. GraphIt separates algorithm specifications from performance optimizations. The algorithm language simplifies expressing both ordered and unordered graph algorithms. We formulate graph optimizations as tradeoffs among locality, parallelism, and work-efficiency. The scheduling language enables programmers to easily search through the complicated tradeoff space. We also expanded the optimization space with several new performance optimizations, including frequency-based clustering, CSR segmenting, and bucket fusion. We introduce the *graph iteration space* to model, compose, and ensure the validity of the edge traversal optimizations. The separation of algorithm and schedule, and the correctness guarantee of edge traversal optimizations enabled us to build an autotuner on top of GraphIt. Our experiments demonstrate that GraphIt is up to 4.8$\times$ faster than state-of-the-art graph frameworks while reducing the lines of code by up to one order of magnitude. GraphIt is available as an open-source project.[1]

[1] The GraphIt compiler is available under the MIT license at `http://graphit-lang.org/` and `https://github.com/GraphIt-DSL/graphit`

Bibliography

[1] OpenStreetMap. https://www.openstreetmap.org/, 2019.

[2] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. In *International Conference on Management of Data*, SIGMOD '16, pages 431–446, 2016.

[3] Maleen Abeydeera and Daniel Sanchez. Chronos: Efficient speculative parallelism for accelerators. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[4] A. Addisie, H. Kassa, O. Matthews, and V. Bertacco. Heterogeneous memory subsystem for natural graph analytics. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 134–145, Sep. 2018.

[5] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2015.

[6] Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Zheng Li, and Giorgi Nadiradze. Distributionally linearizable data structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 133–142, 2018.

[7] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 11–20, 2015.

[8] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, 2014.

[9] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205, 2019.

[10] V. Balaji and B. Lucia. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 203–214, 2018.

[11] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, 2012.

[12] Scott Beamer, Krste Asanovic, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 56–65, 2015.

[13] Scott Beamer, Krste Asanovic, and David Patterson. Reducing pagerank communication via propagation blocking. In *IEEE International Parallel and Distributed Processing Symposium*, pages 820–831, May 2017.

[14] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[15] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[16] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 235–248, 2017.

[17] James Bennett, Stan Lanning, and Netflix Netflix. The netflix prize. In *In KDD Cup and Workshop in conjunction with KDD*, 2007.

[18] Maciej Besta, Michal 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*, HPDC '17, pages 93–104, 2017.

[19] Guy E. Blelloch, Richard Peng, and Kanat Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2011.

[20] Guy E. Blelloch, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Parallel and I/O efficient set covering algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012.

[21] A. Bowyer. Computing Dirichlet tessellations*. *The Computer Journal*, 24(2):162–166, 01 1981.

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

[23] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, 2008.

[24] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 1:1–1:15, 2015.

[25] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[26] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang. GraphH: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):640–653, April 2019.

[27] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 752–768, 2018.

[28] Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Phoenix: A substrate for resilient distributed graph analytics. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 615–630, 2019.

[29] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[30] Camil Demetrescu, Andrew Goldberg, and David Johnson. 9th dimacs implementation challenge - shortest paths. http://www.dis.uniroma1.it/challenge9/, 2019.

[31] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, 2011.

[32] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 293–304, 2017.

[33] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 918–934, New York, NY, USA, 2019. Association for Computing Machinery.

[34] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5, 2012.

[35] 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 '18, pages 1775–1784, 2018.

[36] Zhisong Fu, Zhengwei Wu, Houyu Li, Yize Li, Min Wu, Xiaojie Chen, Xiaomeng Ye, Benquan Yu, and Xi Hu. Geabase: A high-performance distributed graph database for industry-scale applications. In *International Conference on Advanced Cloud and Big Data (CBD)*, pages 170–175, 2017.

[37] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Andrew Lenharth, and Keshav Pingali. Abelian: A compiler for graph analytics on distributed, heterogeneous platforms. In *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, pages 249–264, 2018.

[38] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, 2012.

[39] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. Making pull-based graph processing performant. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 246–260, 2018.

[40] Gundolf Haase, Manfred Liebmann, and Gernot Plank. A hilbert-order multiplication scheme for unstructured sparse matrices. *Int. J. Parallel Emerg. Distrib. Syst.*, 22(4):213–220, January 2007.

[41] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[42] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 56:1–56:13, 2016.

[43] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. Kla: A new algorithmic paradigm for parallel graph computations. In *International Conference on Parallel Architectures and Compilation (PACT)*, pages 27–38, 2014.

[44] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 3–12, 2011.

[45] Muhammad Amber Hassaan, Donald D. Nguyen, and Keshav K. Pingali. Kinetic dependence graphs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 457–471, 2015.

[46] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: A dsl for easy and efficient graph analysis. *SIGARCH Comput. Archit. News*, 40(1):349–362, March 2012.

[47] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph.*, 38(6), November 2019.

[48] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez. Data-centric execution of speculative parallel programs. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[49] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. A scalable architecture for ordered parallelism. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 228–241, Dec 2015.

[50] M. C. Jeffrey, V. A. Ying, S. Subramanian, H. R. Lee, J. Emer, and D. Sanchez. Harmonizing speculative and non-speculative execution in architectures for ordered parallelism. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 217–230, Oct 2018.

[51] Sang Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Grafboost: Accelerated flash storage for external graph analytics. In *International Symposium on Computer Architecture*, 2018.

[52] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2), 2011.

[53] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. Simit: A language for physical simulation. *ACM Trans. Graph.*, 35(2):20:1–20:21, March 2016.

[54] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[55] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *International Conference on World Wide Web (WWW)*, pages 591–600, 2010.

[56] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a pc. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[57] Monica S. Lam, Stephen Guo, and Jiwon Seo. Socialite: Datalog extensions for efficient social network analysis. In *ICDE*, pages 278–289, 2013.

[58] Boduo Li, Sandeep Tata, and Yannis Sismanis. Sparkler: Supporting large-scale matrix factorization. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 625–636, 2013.

[59] Gushu Li, Guohao Dai, Shuangchen Li, Yu Wang, and Yuan Xie. GraphIA: An in-situ accelerator for large-scale graph processing. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, pages 79–84, 2018.

[60] Zhiyuan Li, Pen-Chung Yew, and Chuag-Qi Zhu. Data dependence analysis on multi-dimensional array references. In *Proceedings of the 3rd international conference on Supercomputing*, pages 215–224, 1989.

[61] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

[62] Adam Lugowski, David Alber, Aydın Buluç, John Gilbert, Steve Reinhardt, Yun Teng, and Andrew Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *SDM*, 2012.

[63] 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*, EuroSys '17, pages 527–543, 2017.

[64] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, July 1983.

[65] Dror E Maydan, John L Hennessy, and Monica S Lam. Efficient and exact data dependence analysis. In *ACM SIGPLAN Notices*, volume 26, pages 1–14, 1991.

[66] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, October 2015.

[67] Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! but at what COST? In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[68] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. A pattern based algorithmic autotuner for graph processing on gpus. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 201–213, 2019.

[69] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. Web data commons - hyperlink graphs. http://webdatacommons.org/hyperlinkgraph.

[70] Ulrich Meyer and Peter Sanders. δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[71] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, 2018.

[72] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Phi: Architectural support for synchronization- and bandwidth-efficient commutative scatter updates. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1009–1022, 2019.

[73] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Phi: Architectural support for synchronization- and bandwidth-efficient commutative scatter updates. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1009–1022, 2019.

[74] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 456–471, 2013.

[75] Rajesh Nishtala, Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):297–311, May 2007.

[76] David A Padua and Michael J Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

[77] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[78] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 2016.

[79] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. Managing large graphs on multi-cores with graph awareness. In *USENIX Conference on Annual Technical Conference (ATC)*, 2012.

[80] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, December 2017.

[81] Dolbeau Romain, Stephane Bihan, and Francois Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on general purpose processing on graphics processing units*, GPGPU 2007, 2007.

[82] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424, 2015.

[83] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 472–488, 2013.

[84] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, December 2017.

[85] Sherif Sakr, Faisal Moeen Orakzai, Ibrahim Abdelaziz, and Zuhair Khayyat. *Large-Scale Graph Processing Using Apache Giraph*. Springer Publishing Company, Incorporated, 1st edition, 2017.

[86] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 979–990, 2014.

[87] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. Graphjet: Real-time content recommendations at twitter. *Proc. VLDB Endow.*, 9(13):1281–1292, September 2016.

[88] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on gpus: A survey. *ACM Comput. Surv.*, 50(6):81:1–81:35, January 2018.

[89] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 149–160, April 2015.

[90] Julian Shun. Practical parallel hypergraph algorithms. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 232–249, New York, NY, USA, 2020. Association for Computing Machinery.

[91] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 135–146, 2013.

[92] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K. John. Start late or finish early: A distributed graph processing system with redundancy reduction. *PVLDB*, 12(2):154–168, 2018.

[93] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez. Fractal: An execution model for fine-grain nested speculative parallelism. In *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 587–599, 2017.

[94] 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, pages 16:1–16:10, 2017.

[95] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015.

[96] M. Sutton, T. Ben-Nun, and A. Barak. Optimizing parallel graph connectivity computation via subgraph sampling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 12–21, 2018.

[97] Keval Vora, Rajiv Gupta, and Guoqing (Harry) Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 237–251, 2017.

[98] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. Aspire: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *ACM International Conference on Object Oriented Programming Systems Languages, and Applications (OOPSLA)*, page 861–878, 2014.

[99] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference*, pages 507–522, 2016.

[100] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *International Conference on Architectural*

Support for Programming Languages and Operating Systems, ASPLOS '17, pages 389–404, 2017.

[101] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. Graphq: Graph query processing with abstraction refinement—scalable and programmable analytics over very large graphs on a single PC. In *USENIX Annual Technical Conference*, pages 387–401, 2015.

[102] Lei Wang, Liangji Zhuang, Junhang Chen, Huimin Cui, Fang Lv, Ying Liu, and Xiaobing Feng. Lazygraph: Lazy data coherency for replicas in distributed graph-parallel computation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 276–289, 2018.

[103] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and et al. Gunrock: Gpu graph analytics. *ACM Trans. Parallel Comput.*, 4(1), August 2017.

[104] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. Gunrock: GPU graph analytics. *ACM Trans. Parallel Comput.*, 4(1):3:1–3:49, August 2017.

[105] Michael E Wolf and Monica S Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, 1991.

[106] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *CoRR*, abs/1901.00596, 2019.

[107] Chengshuo Xu, Keval Vora, and Rajiv Gupta. Pnp: Pruning and prediction for point-to-point iterative graph analytics. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 587–600, 2019.

[108] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1-2):1–195, 2017.

[109] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowl. Inf. Syst.*, 42(1):181–213, January 2015.

[110] Pengcheng Yao, Long Zheng, Xiaofei Liao, Hai Jin, and Bingsheng He. An efficient graph accelerator with parallel data conflict management. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 8:1–8:12, 2018.

[111] Albert-Jan N. Yzelman and Rob H. Bisseling. A cache-oblivious sparse matrix,vector multiplication scheme based on the hilbert curve. In *Progress in Industrial Mathematics at ECMI 2010*, volume 17 of *Mathematics in Industry*, pages 627–633. Springer Berlin Heidelberg, 2012.

[112] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 183–193, 2015.

[113] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 608–621, 2018.

[114] 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.

[115] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *IEEE International Conference on Big Data (Big Data)*, pages 293–302, 2017.

[116] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA):121:1–121:30, October 2018.

[117] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. Flashgraph: Processing billion-node graphs on an array of commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, 2015.

[118] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 301–316, 2016.

[119] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 375–386, 2015.