Optimizing Ordered Graph Algorithms with GraphIt

Yunming Zhang
MIT CSAIL
USA
yunming@mit.edu

Ajay Brahmakshatriya
MIT CSAIL
USA
ajaybr@mit.edu

Xinyi Chen
MIT CSAIL
USA
xinyic@mit.edu

Laxman Dhulipala
Carnegie Mellon University
USA
ldhulipa@cs.cmu.edu

Shoaib Kamil
Adobe Research
USA
kamil@adobe.com

Saman Amarasinghe
MIT CSAIL
USA
saman@csail.mit.edu

Julian Shun
MIT CSAIL
USA
jshun@mit.edu

Abstract

Many graph problems can be solved using ordered parallel graph algorithms that achieve significant speedup over their unordered counterparts by reducing redundant work. This paper introduces a new priority-based extension to GraphIt, a domain-specific language for writing graph applications, to simplify writing high-performance parallel ordered graph algorithms. The extension enables vertices to be processed in a dynamic order while hiding low-level implementation details from the user. We extend the compiler with new program analyses, transformations, and code generation to produce fast implementations of ordered parallel graph algorithms. We also introduce *bucket fusion*, a new performance optimization that fuses together different rounds of ordered algorithms to reduce synchronization overhead, resulting in 1.2×–3× speedup over the fastest existing ordered algorithm implementations on road networks with large diameters. With the extension, GraphIt achieves up to 3× speedup on six ordered graph algorithms over state-of-the-art frameworks and hand-optimized implementations (Julienne, Galois, and GAPBS) that support ordered algorithms.

CCS Concepts • **Mathematics of computing → Graph algorithms**; • **Software and its engineering → Parallel programming languages**; **Domain specific languages**.

Keywords Compiler Optimizations, Graph Processing

ACM Reference Format:
Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. 2020. Optimizing Ordered Graph Algorithms with GraphIt. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368826.3377909

CGO '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7047-9/20/02.
https://doi.org/10.1145/3368826.3377909

1 Introduction

Many important graph problems can be implemented using either *ordered* or *unordered* parallel algorithms. *Ordered* algorithms process active vertices following a dynamic priority-based ordering, potentially reducing redundant work. By contrast, *unordered* algorithms process active vertices in an arbitrary order, improving parallelism while potentially performing a significant amount of redundant work. In practice, optimized ordered graph algorithms are up to two orders of magnitude faster than the unordered versions [7, 16, 22, 23], as shown in Figure 1. For example, computing single-source shortest paths (SSSP) on graphs with non-negative edge weights can be implemented either using the Bellman-Ford algorithm [8] (an unordered algorithm) or the Δ-stepping algorithm [32] (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 path distances to vertices that are closer to the source vertex first, before processing vertices farther away.

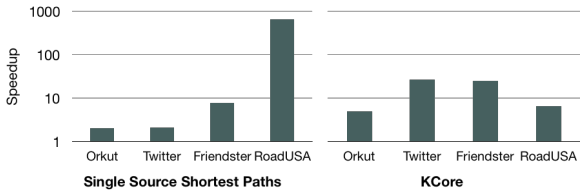

Figure 1. Speedup of ordered algorithms for single-source shortest path and *k*-core over the corresponding unordered algorithms implemented in our framework on a 24-core machine.

Writing high-performance ordered graph algorithms is challenging for users who are not experts in performance optimization. Existing frameworks that support ordered graph algorithms [7, 16, 35] require users to be familiar with C/C++ data structures, atomic synchronizations, bitvector manipulations, and other performance optimizations. For example,

[1] In this paper, we define Δ-stepping as an ordered algorithm, in contrast to previous work [22] which defines Δ-stepping as an unordered algorithm.

```
1 constexpr uintE TOP_BIT = ((uintE)INT_E_MAX) + 1;
2 constexpr uintE VAL_MASK = INT_E_MAX;
3 struct Visit_F {
4   array_imap<uintE> dists;
5   Visit_F(array_imap<uintE>& _dists) : dists(_dists) { }
6   ....
7   inline Maybe<uintE> updateAtomic(uintE& s, uintE& d, intE& w) {
8     uintE oval = dists.s[d];
9     uintE dist = oval | TOP_BIT;
10    uintE n_dist = (dists.s[s] | TOP_BIT) + w;
11    if (n_dist < dist) {
12      if (!(oval & TOP_BIT) && CAS(&(dists[d]), oval, n_dist)) {
13        return Maybe<uintE>(oval);}
14      writeMin(&(dists[d]), n_dist);}
15    return Maybe<uintE>();}
16  inline bool cond(const uintE& d) const { return true; }};
```

Figure 2. Part of Julienne's Δ-stepping edge update function, corresponding to Lines 7–10 of Fig. 3 in GraphIt's Δ-stepping.

```
1  element Vertex end
2  element Edge end
3  const edges : edgeset{Edge}(Vertex,Vertex, int)=load(argv[1]);
4  const dist : vector{Vertex}(int) = INT_MAX;
5  const pq: priority_queue{Vertex}(int);
6
7  func updateEdge(src : Vertex, dst : Vertex, weight : int)
8      var new_dist : int = dist[src] + weight;
9      pq.updatePriorityMin(dst, dist[dst], new_dist);
10 end
11
12 func main()
13     var start_vertex : int = atoi(argv[2]);
14     dist[start_vertex] = 0;
15     pq = new priority_queue
16         {Vertex}(int)(true, "lower_first", dist, start_vertex);
17     while (pq.finished() == false)
18         var bucket : vertexset{Vertex} = pq.dequeueReadySet();
19         #s1# edges.from(bucket).applyUpdatePriority(updateEdge);
20         delete bucket;
21     end
22 end
```

Figure 3. GraphIt algorithm for Δ-stepping for SSSP. Priority-based data structures and operators are highlighted in red.

Figure 2 shows a snippet of a user-defined function for Δ-stepping in Julienne [16], a state-of-the-art framework for ordered graph algorithms. The code involves atomics and low-level C/C++ operations.

We propose a new priority-based extension to GraphIt that simplifies writing parallel ordered graph algorithms. GraphIt separates algorithm specifications from performance optimization strategies. The user specifies the high-level algorithm with the algorithm language and uses a separate scheduling language to configure different performance optimizations. The algorithm language extension introduces a set of priority-based data structures and operators to maintain execution ordering while hiding low-level details such as synchronization, deduplication, and data structures to maintain ordering of execution. Figure 3 shows the implementation of Δ-stepping using the priority-based extension which dequeues vertices with the lowest priority and updates their neighbors' distances in each round of the while loop. The while loop terminates when all the vertices' distances are finalized. The algorithm uses an abstract priority queue data structure, `pq` (Line 5), and the operators `updatePriorityMin` (Line 9) and `dequeueReadySet` (Line 18) to maintain priorities.

The priority-based extension uses a *bucketing* data structure [7, 16] to maintain the execution ordering. 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* [7] approach or a *lazy bucket update* [16] 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. Existing frameworks supporting ordered parallel graph algorithms only support one of the two bucketing strategies described above. However, using a suboptimal bucketing strategy can result in more than *10× slowdown*, as we show later. Eager and lazy bucketing implementations use different data structures and parallelization schemes, making it difficult to combine both approaches within a single framework.

With the priority-based extension, programmers can switch between lazy and eager bucket update strategies and combine bucketing optimizations with other optimizations using the scheduling language. The compiler leverages program analyses and transformations to generate efficient code for different optimizations. The separation of algorithm and schedule also enables us to build an autotuner for GraphIt that can automatically find high-performance combinations of optimizations for a given ordered algorithm and graph.

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. 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 priority are updated to the current 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 over existing work.

We implement the priority-based model as a language and compiler extension to GraphIt [52][2], a domain-specific language for writing high-performance graph algorithms. With the extension, GraphIt achieves up to 3× speedup on six ordered graph algorithms (Δ-stepping based single-source shortest paths (SSSP), Δ-stepping based point-to-point shortest path (PPSP), weighted BFS (wBFS), A* search, *k*-core decomposition, and approximate set cover (SetCover)) over the fastest state-of-the-art frameworks that support ordered algorithms (Julienne [16] and Galois [35]) and hand-optimized

[2]https://github.com/GraphIt-DSL/graphit

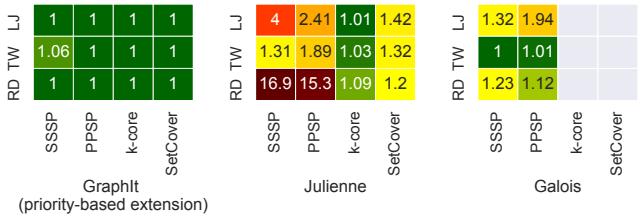

Figure 4. A heatmap of slowdowns of three frameworks compared to the fastest of all frameworks for SSSP, PPSP, k-core, and SetCover. Lower numbers (green) are better, with a value of 1 being the fastest. Gray means that an algorithm is not supported. TW and LJ are Twitter, and LiveJournal graphs with random weights between 1 and 1000. RD is the RoadUSA graph with original weights.

implementations (GAPBS [7]). Figure 4 shows that GraphIt is up to 16.9× and 1.94× faster than Julienne and Galois on the four selected algorithms and supports more algorithms than Galois. Using GraphIt also reduces the lines of code compared to existing frameworks and libraries by up to 4×.

The contributions of this paper are as follows.

- An analysis of the performance tradeoffs between eager and lazy bucket update optimizations (Sections 3 and 6).
- A novel performance optimization for the eager bucket update approach, *bucket fusion* (Sections 3 and 6).
- A new priority-based programming model in GraphIt that simplifies the programming of ordered graph algorithms and makes it easy to switch between and combine different optimizations (Section 4).
- Compiler extensions that leverage program analyses, program transformations, and code generation to produce efficient implementations with different combinations of optimization strategies (Section 5).
- A comprehensive evaluation of GraphIt that shows that it is up to 3× faster than state-of-the-art graph frameworks on six ordered graph algorithms (Section 6). GraphIt also significantly reduces the lines of code compared to existing frameworks and libraries.

2 Preliminaries

We first define *ordered graph processing* used throughout this paper. Each vertex has a priority p_v. Initially, the users can explicitly initialize 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.

```
1  Dist = {∞, . . . , ∞}                                   ▷ Length |V| array
2  procedure SSSP with Δ-stepping(Graph G, Δ, startV)
3      B = new LazyBucket(Dist, Δ, startV);
4      Dist[startV] = 0
5      while ¬empty B do
6          minBucket = B.getMinBucket()
7          buffer = new BucketUpdateBuffer();
8          parallel for src : minBucket do
9              for e : G.getOutEdge[src] do
10                 Dist[e.dst] = min(Dist[e.dst], Dist[src] + e.weight)
11                 buffer.syncAppend(e.dst, ⌊Dist[e.dst]/Δ⌋)
12         buffer = buffer.reduceBucketUpdates();
13         B.bulkUpdateBuckets(buffer);
```

Figure 5. Δ-stepping for single-source shortest paths (SSSP) with the lazy bucket update approach.

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 PPSP, but not in k-core and SetCover.

3 Performance Optimizations for Ordered Graph Algorithms

We use Δ-stepping for single-source shortest paths (SSSP) as a running example to illustrate the performance tradeoffs between the lazy and eager bucket update approaches, and to introduce our new bucket fusion optimization.

3.1 Lazy Bucket Update

We first consider using the lazy bucket update approach for the Δ-stepping algorithm, with pseudocode shown in Figure 5. The algorithm constructs a bucketing data structure in Line 3, which groups the vertices into *buckets* according to their priority. 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 5–13). 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 10). 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 11). 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 have one final bucket update (Line 12). Finally, the buckets are updated in bulk with `bulkUpdateBuckets` (Line 13).

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

```
1  Dist = {∞, . . . , ∞}                                          ▷ Length |V| array
2  procedure SSSP with Δ-stepping(Graph G, Δ, startV)
3      B = new ThreadLocalBuckets(Dist, Δ, startV);
4      for threadID : threads do
5          B.append(new LocalBucket());
6      Dist[startV] = 0
7      while ¬empty B do
8          minBucket = B.getGlobalMinBucket()
9          parallel for threadID : threads do
10             for src : minBucket.getVertices(threadID) do
11                 for e : G.getOutEdge[src] do
12                     Dist[e.dst] = min(Dist[e.dst], Dist[src] + e.weight)
13                     B[threadID].updateBucket(e.dst, ⌊Dist[e.dst]/Δ⌋)
```

Figure 6. Δ-stepping for SSSP with the eager bucket update approach.

as histogram-based reduction on 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 12) 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).

3.2 Eager Bucket Update

Another approach for implementing Δ-stepping is to use an eager bucket update approach (shown in Figure 6) 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 9). Each thread works on a disjoint subset of vertices in the current bucket (Line 10). Using thread-local buckets avoids atomic synchronization overheads on bucket updates (Lines 3 and 12–13). 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 8). If a thread does not have a local bucket of the next smallest priority, then it will skip 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 5, Line 12). However, it potentially needs to perform multiple bucket updates per vertex in each round.

3.3 Eager Bucket Fusion Optimization

A major challenge in bucketing is that a large number of buckets need to be processed, resulting in thousands or even tens of thousands of processing rounds. Since 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

```
1  Dist = {∞, . . . , ∞}                                          ▷ Length |V| array
2  procedure SSSP with Δ-stepping(Graph G, Δ, startV)
3      B = new ThreadLocalBuckets(Dist, Δ, startV);
4      for threadID : threads do
5          B.append(new LocalBucket());
6      Dist[startV] = 0
7      while ¬empty B do
8          minBucket = B.getMinBucket()
9          parallel for threadID : threads do
10             for src : minBucket.getVertices(threadID) do
11                 for e : G.getOutEdge[src] do
12                     Dist[e.dst] = min(Dist[e.dst], Dist[src] + e.weight)
13                     B[threadID].updateBucket(e.dst, ⌊Dist[e.dst]/Δ⌋)
14             while B[threadID].currentLocalBucket() is not empty  do
15                 currentLocalBucket = B[threadID].currentLocalBucket()
16                 if currentLocalBucket.size() < threshold then
17                     for src : currentLocalBucket do
18                         for e : G.getOutEdge[src] do
19                             Dist[e.dst] = min(Dist[e.dst], Dist[src] + e.weight)
20                             B[threadID].updateBucket(e.dst, ⌊Dist[e.dst]/Δ⌋)
21                 else break
```

Figure 7. Δ-stepping for single-source shortest paths with the eager bucket update approach and the bucket fusion optimization.

in a single round. As a result, 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 7. The same optimization can be applied in other applications, such as wBFS, A* search and point-to-point shortest path. This algorithm extends the eager bucket update algorithm (Figure 6) by adding a while loop inside each local thread (Figure 7, Line 14). 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 7, Line 16). 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 7, Lines 17–20) is the same as the original processing logic (Figure 7, 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).

Table 1. Algorithm language extensions in GraphIt.

Edgeset Apply Operator	Return Type	Description
`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 allow_priority coarsening, string priority_direction, vector priority_vector, Vertex optional_start_vertex)`	priority_queue	The constructor for the priority queue. It specifies whether priority coarsening is allowed or not, higher or lower priority gets executed first, the vector that is used to compute the priority values, and an optional start vertex. A `lower_first` priority direction specifies that lower priority values are executed first, whereas a `higher_first` indicates higher priority values are executed first.
`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 will not go below the threshold.

4 Programming Model

The new priority-based extension follows the design of GraphIt and separates the algorithm specification from the performance optimizations, similar to Halide [38] and Tiramisu [6]. The user writes a high-level algorithm using the algorithm language and specifies optimization strategies using the scheduling language. The extension introduces a set of priority-based data structures and operators to GraphIt to maintain execution order in the algorithm language and adds support for bucketing optimizations in the scheduling language.

4.1 Algorithm Language

The algorithm language exposes opportunities for eager bucket update, eager update with bucket fusion, lazy bucket update, and other optimizations. The high-level operators hide low-level implementation details such as atomic synchronization, deduplication, bucket updates, and priority coarsening. The algorithm language shares the vertex and edge sets, and operators that apply user-defined functions on the sets with the GraphIt algorithm language.

The new priority-based extension proposes high-level priority queue-based abstractions to switch between thread-local and global buckets. The extension to GraphIt also introduces priority update operators to hide the bucket update mechanisms, and provides a new edgeset apply operator, `applyUpdatePriority`. The priority-based data structures and operators are shown in Table 1.

Figure 3 shows an example of Δ-stepping for SSSP. GraphIt works on vertex and edge sets. 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 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.

Programmers use the constructor of the priority queue (Lines 15–16) to specify algorithmic information, such as the priority ordering, support for priority coarsening, and the direction that priorities change (documented in Table 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 (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` (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.

4.2 Scheduling Language

The scheduling language allows users to specify different optimization strategies without changing the algorithm. We

Table 2. Scheduling functions for `applyUpdatePriority` operators. Default options are in bold.

Apply Scheduling Functions	Descriptions
`configApplyPriorityUpdate(label,config);`	Config options: eager_with_fusion, eager_no_fusion, lazy_constant_sum, and **lazy**.
`configApplyPriorityUpdateDelta(label,config);`	Configures the Δ parameter for coarsening the priority range.
`configBucketFusionThreshold(label, config);`	Configures the threshold for the bucket fusion optimization.
`configNumBuckets(label,config);`	Configures the number of buckets that are materialized for the lazy bucket update approach.

```
17   ...
18     while (pq.finished() == false)
19         var bucket : vertexsubset = pq.dequeueReadySet();
20         #s1# edges.from(bucket).applyUpdatePriority(updateEdge);
21         delete bucket;
22     end
   ...
25 schedule:
26 program->configApplyPriorityUpdate("s1", "lazy")
27 ->configApplyPriorityUpdateDelta("s1", "4")
28 ->configApplyDirection("s1", "SparsePush")
29 ->configApplyParallelization("s1","dynamic-vertex-parallel");
```

Figure 8. GraphIt scheduling specification for Δ-stepping.

extend the scheduling language of GraphIt with new commands to enable switching between eager and lazy bucket update strategies. Users can also tune other parameters, such as the coarsening factor for priority coarsening. The scheduling API extensions are shown in Table 2.

Figure 8 shows 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. `configApplyUpdateDelta` is used to set the delta for priority coarsening.

Users can change the schedules to generate code with different combinations of optimizations as shown in Figure 9. Figure 9(a) shows 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` means sparse frontier and push direction). Figure 9(b) shows the code generated when we combine a different traversal direction (`DensePull`) with the lazy bucketing strategy. Figure 9(c) shows code generated with the eager bucket update strategy. Code generation is explained in Section 5.

5 Compiler Implementation

We demonstrate how the compiler generates code for different bucketing optimizations. 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.

5.1 Lazy Bucket Update Schedules

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 if there are write-write conflicts and insert atomics instructions as necessary (Figure 9(a) Line 20). Additionally, the compiler needs to insert variables to track whether a vertex's priority has been updated or not (`tracking_var` in Figure 9(a), Line 18). This variable is used in the generated code to determine which vertices should be added to the buffer `outEdges` (Figure 9(a), Line 21). Deduplication is enabled with a compare-and-swap (CAS) on deduplication flags (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. Changing the schedules with different traversal directions or frontier layout affects the code generation for edge traversal and user-defined functions (Figure 9(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 9(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 9(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 [16]. 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.

Lazy with constant sum reduction. We also incorporated a specialized histogram-based reduction optimization (first proposed in Julienne [16]) 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, since the priorities for each vertex always reduce

```
1  int * dist = new int[num_verts];
2  LazyPriorityQueue* pq;
3  int delta = 4;
4  WGraph* G = loadGraph(argv[1]);
5
6  //simplified snippets of the generated main function
7  …
8  dist[start_vertex] = 0;
9  pq = new LazyPriorityQueue(true, "lower", dist, delta);
10 while (pq.finished()){
11   VertexSubset *  frontier = getNextBucket(pq);
12   uint* outEdges = setupOutputBuffer(g, frontier);
13   uint* offsets = setupOutputBufferOffsets(g, frontier);
14   parallel_for (uint s : frontier.vert_array) {
15     int j = 0;
16     uint offset = offsets[i];
17     for(WNode d : G.getOutNgh(s)){
18       bool tracking_var = false;
19       int new_dist = dist[s.v] + d.weight;
20       tracking_var = atomicWriteMin(&dist[d.v], new_dist);
21       If (tracking_var && CAS(dedup_flags[d.v],0,1)){
22         outEdges[offset + j] = d.v;
23       } else { outEdges[offset + j] = UINT_MAX; }
24       j++;
25   }}
26   VertexSubset* nextFrontier = setupFrontier(outEdges);
27   updateBuckets(nextFrontier, pq, delta);
28   …
29 }
30 …
```

(a) *Lazy Bucket Update with Parallel SparsePush Traversal*

```
program->configApplyPriorityUpdate("s1", "lazy")
->configApplyPriorityUpdateDelta("s1", 4)
->configApplyDirection("s1", SparsePush")
->configApplyParallelization("s1", "dynamic-vertex-parallel");
```

```
1  int * dist = new int[num_verts];
2  LazyPriorityQueue* pq;
3  int delta = 4;
4  WGraph* G = loadGraph(argv[1]);
5
6  //simplified snippets of the generated main function
7  …
8  dist[start_vertex] = 0;
9  pq = new LazyPriorityQueue(true, "lower", dist, delta);
10 while (pq.finished()){
11   VertexSubset *  frontier = getNextBucket(pq);
12   bool * next = newA(bool, g.num_nodes());
13   parallel_for (uint i = 0; i < numNodes; i++) next[i] = 0;
14   parallel_for (uint d=0; d < numNodes; d++) {
15     for(WNode s : G.getInNgh(d)){
16       if (frontier->bool_map_[s.v] ) {
17         bool tracking_var = false;
18         int new_dist = dist[s.v] + s.weight;
19         If (new_dist < dist[d]){
20           dist[d] = new_dist;
21           tracking_var = true;}
22         if ( tracking_var ) {next[d] = 1;}
23 }}}
24   VertexSubset* nextFrontier = setupFrontier(next);
25   updateBuckets(nextFrontier, pq, delta);
26   …
27 }
28 …
29
```

(b) *Lazy Bucket Update with Parallel DensePull Traversal*

```
program->configApplyPriorityUpdate("s1", "lazy")
->configApplyPriorityUpdateDelta("s1", 4)
->configApplyDirection("s1", DensePull")
->configApplyParallelization("s1", "dynamic-vertex-parallel");
```

```
1  int * dist = new int[num_verts];
2  EagerPriorityQueue* pq;
3  int delta = 4;
4  WGraph* G = loadGraph(argv[1]);
5
6  //simplified snippets of the generated main function
7  …
8  dist[start_vertex] = 0;
9  frontier[0] = start_vertex;
10 pq = new EagerPriorityQueue(true, "lower", dist, delta);
11 uint* frontier =  new uint[G.num_edges()];
12 #pragma omp parallel
13 {  vector<vector<uint> > local_bins(0);
14    while (pq.finished()) {
15      #pragma omp for nowait schedule(dynamic, 64)
16      for (size_t i = 0; i < frontier.size(); i++) {
17        uint s = frontier[i];
18        for (WNode d : G.getOutNgh(s)) {
19          int new_dist = dist[s] + d.weight;
20          bool changed = atomicWriteMin(&dist[d.v],new_dist);
21          if (changed == false) {break;}}
22          if (changed) {
23            size_t dest_bin = new_dist/delta;
24            if (dest_bin >= local_bins.size()) {
25              local_bins.resize(dest_bin+1);}
26            local_bins[dest_bin].push_back(d.v);
27        }}}// end of for frontier for loop
28    … //omitted:find next bucket
29    #pragma omp barrier
30    … //omitted:copy local buckets to global bucket
31    #pragma omp barrier } // end of parallel region
32 …
```

(c) *Eager Bucket Update with Parallel SparsePush Traversal*

```
program->configApplyPriorityUpdate("s1", "eager")
->configApplyPriorityUpdateDelta("s1", 4)
->configApplyDirection("s1", SparsePush")
->configApplyParallelization("s1", "dynamic-vertex-parallel");
```

Figure 9. Simplified generated C++ code for Δ-stepping for SSSP with different schedules. Changes in schedules for (b) and (c) compared to (a) are highlighted in blue. Changes in the generated code are highlighted in purple background. `parallel_for` is translated to `cilk_for` or `#pragma omp parallel for`. Struct `WNode` has two fields, `v` and `weight`. `v` stores the vertex ID and `weight` stores the edge weight.

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

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

Figure 10. The original (top) and transformed (bottom) user-defined function for *k*-core using lazy with constant sum reduction.

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 user-defined function to determine whether the change to the priority of the vertex is a fixed value and if it is a sum reduction (Figure 10 (top), Line 3). The compiler ensures that there is only one priority update operator in the user-defined function. It then extracts the fixed value (`-1`), the minimum priority (`k`), and vertex identifier (`dst`). In the transformed function (Figure 10 (bottom)), an if statement and max operator are generated to check and maintain the priority of the vertex. The `applyUpdatePriority` operator gets the counts of updates to each vertex using a histogram approach and supplies the vertex and count as arguments to the transformed function (Figure 10 (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 Eager Bucket Update Schedules

The compiler uses program analysis to determine feasibility of the transformation, transforms user-defined functions 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 3, 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 9(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 [7]. A global vertex frontier (Figure 9(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 9(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 9(c), Line 28). Once the next bucket priority is decided, each thread

Table 3. 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 Graphs	*Orkut* (OK) [49]	3 M	234 M	234 M
	LiveJournal (LJ) [14]	5 M	69 M	85M
	Twitter (TW) [27]	41 M	1469 M	2405 M
	Friendster (FT) [49]	125 M	3612 M	3612 M
Web Graph	*WebGraph* (WB) [31]	101 M	2043 M	3880 M
Road Graphs	*Massachusetts* (MA) [1]	0.45 M	1.2 M	1.2 M
	Germany (GE) [1]	12 M	32 M	32 M
	RoadUSA (RD) [15]	24 M	58 M	58 M

will copy vertices in its next local bucket to the global frontier (Figure 9(c), Line 30)

Finally, the compiler transforms the user-defined functions by appending the local buckets to the argument list and inserting appropriate synchronization instructions. These transformations allow priority update operators to directly update thread-local buckets (Figure 9(c), Lines 23–26).

Bucket Fusion. The bucket fusion optimization adds another while loop after end of the for-loop on Line 27 of Figure 9(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 user-defined functions as before. The size threshold improves load balancing, as large buckets are distributed across different threads.

5.3 Autotuning

We built an autotuner on top of the extension to automatically find high-performance schedules for a given algorithm and graph. The autotuner is built using OpenTuner [5] and stochastically searches through a large number of optimization strategies generated with the scheduling language. It uses an ensemble of search methods, such as the area under curve bandit meta technique, to find good combinations of optimizations within a reasonable amount of time.

6 Evaluation

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. 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 128 GB of DDR3-1600 memory and 30 MB last level cache on each socket and runs with Transparent Huge Pages (THP) enabled and Ubuntu 18.04.

Data Sets. Table 3 shows our input graphs and their sizes. 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 graphs with integer edge weights. 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]. Weight distributions used for experiments are described in the caption of Table 4.

Existing Frameworks. Galois v4 [35] 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 currently not supported for Galois. Galois does not provide implementations of wBFS, k-core and SetCover, which require strict priority ordering. GAPBS [7] is a suite of C++ implementations of graph algorithms and uses eager bucket update for SSSP. GAPBS does not provide implementations of k-core and SetCover. We used Julienne [16] from early 2019. The developers of Julienne have since incorporated the optimized bucketing interface proposed in this paper in the latest version. GraphIt [52] and Ligra [40] 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.

6.1 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 5. 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 [16]. 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 [16].

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.

Table 4. 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.

Algorithm	SSSP							PPSP							wBFS				
Graph	LJ	OK	TW	FT	WB	GE	RD	LJ	OK	TW	FT	WB	GE	RD	LJ†	OK†	TW†	FT†	WB†
GraphIt with extension (ordered)	**0.093**	**0.106**	3.094	**5.637**	**2.902**	**0.207**	**0.224**	0.043	**0.061**	**2.597**	**4.063**	**2.473**	**0.049**	**0.045**	**0.072**	**0.104**	**1.822**	**7.563**	**2.129**
GAPBS	0.1	0.107	3.547	6.094	3.304	0.59	0.765	**0.042**	0.063	2.707	4.312	2.628	0.116	0.112	0.072	0.107	1.903	7.879	2.228
Galois	0.123	0.234	**2.93**	7.996	3.005	0.244	0.276	0.084	0.165	2.625	7.092	2.606	0.059	0.051	–	–	–	–	–
Julienne	0.169	0.334	4.522	x	4.11	3.104	3.685	0.104	0.16	4.904	x	4.107	1.836	0.687	0.148	0.145	2.32	x	2.813
GraphIt (unordered)	0.221	0.479	6.376	38.458	8.521	90.524	122.374	0.221	0.479	6.376	38.458	8.521	90.524	122.374	0.12	0.198	2.519	21.77	3.659
Ligra (unordered)	0.301	0.604	7.778	x	x	94.162	129.2	0.301	0.604	7.778	x	x	94.162	129.2	0.164	0.257	3.054	x	x

Algorithm	k-core							Approximate Set Cover							A* search		
Graph	LJ	OK	TW	FT	WB	GE	RD	LJ	OK	TW	FT	WB	GE	RD	MA	GE	RD
GraphIt with extension (ordered)	**0.745**	**1.634**	**10.294**	**14.423**	**12.876**	**0.173**	**0.305**	**0.494**	**0.564**	**5.299**	**11.499**	**7.57**	**0.545**	**0.859**	**0.010**	**0.060**	**0.075**
GAPBS	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0.03	0.142	0.221
Galois	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0.078	0.066	0.083
Julienne	0.752	1.62	10.5	14.6	13.1	0.184	0.327	0.703	0.868	6.89	13.2	10.7	0.66	1.03	0.181	1.551	4.876
GraphIt (unordered)	6.131	8.152	x	325.286	x	0.421	1.757	–	–	–	–	–	–	–	0.456	90.524	122.374
Ligra (unordered)	5.99	8.09	225.102	324	x	0.708	1.76	–	–	–	–	–	–	–	0.832	94.162	129.2

Our A* search implementation is based on a related work [2] and needs the longitude and latitude of the vertices.

k-core. A 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 [29].

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. $\cup_{f \in \mathcal{F}} f = \mathcal{U}$, and a cost function $c : \mathcal{F} \rightarrow \mathbb{R}_+$. 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 paper, we implement the unweighted version of the problem, where $c : \mathcal{F} \rightarrow 1$, but the algorithm used easily generalizes to the weighted case [16]. 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. [10, 11] for algorithmic details and a survey of related work.

6.2 Comparisons with other Frameworks

Table 4 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 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 start and end points to have a balanced selection of different distances.

GraphIt with the priority extension has the fastest SSSP performance on six out of the seven input graphs. Julienne uses significantly more instruction than GraphIt (up to 16.4× 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 lazy bucket update that generates extra instructions to buffer the bucket updates whereas GraphIt saves instructions by using 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). The optimization is especially effective for road networks, where 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 uses 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 as they only run until the distance to the destination vertex is finalized.

Table 5. Line counts of SSSP, 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

A* search is sometimes slower than PPSP because of additional random memory accesses and computation needed for estimating 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 user-defined function 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.

Autotuning. The autotuner for GraphIt is able to automatically find schedules that performed within 5% of the hand-tuned schedules used for Table 4. 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.

Line Count Comparisons. Table 5 shows the line counts of the five graph algorithms implemented in four frameworks. GAPBS, Galois, and Julienne all require the 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 Scalability Analysis

We analyze the scalability of different frameworks in Figure 11 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 11(a) and (b)). Compared to GAPBS, GraphIt uses bucket fusion to significantly reduce synchronization overheads and improves parallelism on the RoadUSA network (Figure 11(c)). GAPBS suffers from NUMA accesses when going beyond a single socket (12 cores). Julienne's overheads from lazy bucket updates make it hard to scale on the RoadUSA graph.

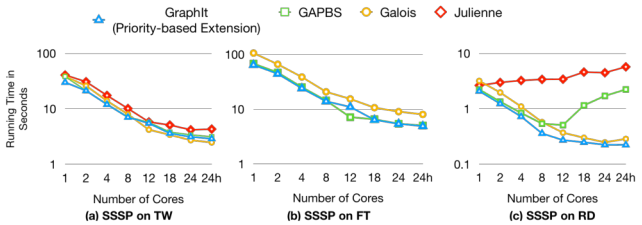

Figure 11. Scalability of different frameworks on SSSP.

Table 6. Running times and number of rounds reductions with the bucket fusion optimization on 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 7. 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

6.4 Performance of Different Schedules

Table 6 shows that SSSP with bucket fusion 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 shows that the optimization improves running time by significantly reducing the number of rounds needed to complete the algorithm.

Table 7 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. Every vertex's priority will be updated the same number of times as its out-degree. In this case, using the lazy bucket update approach drastically reduces the number of bucket insertions. Additionally, with a lazy 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 not many redundant updates and the lazy approach introduces significant runtime overhead over the eager approach.

7 Related Work

Parallel Graph Processing Frameworks. There has been a significant amount of work on unordered graph processing frameworks (e.g., [12, 17–20, 28, 30, 33, 34, 36, 37, 39–41, 43, 44, 46–48, 51–53], 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 [30, 43, 47] have the users define functions that filter out vertices to support Δ-stepping for SSSP. This approach is not very efficient and does not generalize to other ordered algorithms. Wonderland [50] 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 [48] 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 [52] decouples the algorithm from optimizations for unordered graph algorithms. This paper 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.

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 [7, 16]. Existing frameworks support either lazy bucket update or eager bucket update approach. GAPBS [7] is a suite of hand-optimized C++ programs that includes SSSP using the eager bucket update approach. Julienne [16] 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 [22, 23]. 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 [2, 24–26, 42], 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 disregard a strict priority ordering and use an approximate priority ordering [3, 4, 12, 35]. 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 [12, 35] does not currently support strict priority ordering and only supports an approximate ordering. Galois [35] provides an ordered list abstraction, which does not explicitly synchronize after each priority. As a result, it is hard 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. D-galois [13] implements k-core for only a specific k, whereas GraphIt's k-core finds *all* cores.

Synchronization Relaxation. There has been a number of frameworks that relax synchronizations in graph algorithms for better performance while preserving correctness [9, 21, 45]. Compared to existing synchronization relaxation work, bucket fusion in our new priority-based extension is more restricted on synchronization relaxation. The synchronization between rounds can be removed only when the vertices processed in the next round has the same priority as vertices processed in the current round. This way, we ensure no priority inversion happens.

8 Conclusion

We introduce a new priority-based extension to GraphIt that simplifies the programming of parallel ordered graph algorithms and generates high-performance implementations. We propose a novel bucket fusion optimization that significantly improves the performance of many ordered graph algorithms on road networks. GraphIt with the extension achieves up to 3× speedup on six ordered algorithms over state-of-the-art frameworks (Julienne, Galois, and GAPBS) while significantly reducing the number of lines of code.

A Artifact Evaluation Information

- **Algorithms:** SSSP with Δ-stepping, PPSP, wBFS, A* search, k-core, and Approximate Set Cover
- **Compilation:** C++ compiler with C++14 support, Cilk Plus and OpenMP
- **Binary:** Compiled C++ code
- **Data set:** Social, Web, and Road graphs
- **Run-time environment:** Ubuntu 11.04
- **Hardware:** 2-socket Intel Xeon E5-2695 v3 CPUs with Transparent Huge Pages enabled

- **Publicly available?** Yes
- **Code licenses (if publicly available)?** MIT License

The detailed instructions to evaluate the artifact are available at https://github.com/GraphIt-DSL/graphit/blob/master/graphit_eval/priority_graph_cgo2020_eval/readme.md.

The evaluation in the link first demonstrates how SSSP with Δ-stepping with different schedules are compiled to C++ programs (Figure 9). Then we provide instructions on how to run different algorithms on small graphs serially. Finally, there is an optional part that replicates the parallel performance on a more powerful 2-socket machines for LiveJournal, Twitter, and RoadUSA graphs (Table 4).

Acknowledgments

We thank Maleen Abeydeera for help with A* search and Mark Jeffrey for helpful comments. This research was supported by DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, MIT Research Support Committee Award, DARPA SDH Award #HR0011-18-3-0007, Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA, Toyota Research Institute, DoE Exascale award #DE-SC0008923, DARPA D3M Award #FA8750-17-2-0126.

References

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

[2] Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: Efficient Speculative Parallelism for Accelerators. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[3] Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Zheng Li, and Giorgi Nadiradze. 2018. Distributionally Linearizable Data Structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 133–142.

[4] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: A Scalable Relaxed Priority Queue. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 11–20.

[5] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*.

[6] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 193–205.

[7] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). http://arxiv.org/abs/1508.03619

[8] Richard Bellman. 1958. On a Routing Problem. *Quart. Appl. Math.* 16, 1 (1958), 87–90.

[9] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 235–248.

[10] Guy E. Blelloch, Richard Peng, and Kanat Tangwongsan. 2011. Linear-work Greedy Parallel Approximate Set Cover and Variants. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[11] Guy E. Blelloch, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Parallel and I/O Efficient Set Covering Algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[12] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 752–768.

[13] Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2019. Phoenix: A Substrate for Resilient Distributed Graph Analytics. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 615–630.

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

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

[16] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2017. Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 293–304.

[17] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency Graph Streaming Using Compressed Purely-functional Trees. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 918–934.

[18] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. 17–30.

[19] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. 2018. Making Pull-based Graph Processing Performant. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 246–260.

[20] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A High-performance and Energy-efficient Accelerator for Graph Analytics. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 56:1–56:13.

[21] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. 2014. KLA: A New Algorithmic Paradigm for Parallel Graph Computations. In *International Conference on Parallel Architectures and Compilation (PACT)*. 27–38.

[22] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. 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)*. 3–12.

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

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

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

[26] M. C. Jeffrey, V. A. Ying, S. Subramanian, H. R. Lee, J. Emer, and D. Sanchez. 2018. Harmonizing Speculative and Non-Speculative Execution in Architectures for Ordered Parallelism. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 217–230.

[27] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *International Conference on World Wide Web (WWW)*. 591–600.

[28] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 31–46.

[29] David W. Matula and Leland L. Beck. 1983. Smallest-last Ordering and Clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (July 1983), 417–427.

[30] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A Pattern Based Algorithmic Autotuner for Graph Processing on GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 201–213.

[31] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. [n.d.]. Web Data Commons - Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph.

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

[33] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14.

[34] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1009–1022.

[35] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *ACM Symposium on Operating Systems Principles (SOSP)*. 456–471.

[36] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 1–19.

[37] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. 2012. Managing Large Graphs on Multi-cores with Graph Awareness. In *USENIX Conference on Annual Technical Conference (ATC)*.

[38] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling Algorithms from Schedules for High-performance Image Processing. *Commun. ACM* 61, 1 (Dec. 2017), 106–115.

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

[40] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 135–146.

[41] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K. John. 2018. Start Late or Finish Early: A Distributed Graph Processing System with Redundancy Reduction. *PVLDB* 12, 2 (2018), 154–168.

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

[43] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225.

[44] Keval Vora, Rajiv Gupta, and Guoqing (Harry) Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 237–251.

[45] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. 2014. 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)*. 861–878.

[46] Lei Wang, Liangji Zhuang, Junhang Chen, Huimin Cui, Fang Lv, Ying Liu, and Xiaobing Feng. 2018. Lazygraph: Lazy Data Coherency for Replicas in Distributed Graph-parallel Computation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 276–289.

[47] 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. 2017. Gunrock: GPU Graph Analytics. *ACM Trans. Parallel Comput.* 4, 1, Article 3 (Aug. 2017), 3:1–3:49 pages.

[48] Chengshuo Xu, Keval Vora, and Rajiv Gupta. 2019. PnP: Pruning and Prediction for Point-To-Point Iterative Graph Analytics. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 587–600.

[49] Jaewon Yang and Jure Leskovec. 2015. Defining and Evaluating Network Communities Based on Ground-truth. *Knowl. Inf. Syst.* 42, 1 (Jan. 2015), 181–213.

[50] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 608–621.

[51] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making Caches Work for Graph Analytics. In *IEEE International Conference on Big Data (Big Data)*. 293–302.

[52] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-performance Graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (Oct. 2018), 121:1–121:30 pages.

[53] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 301–316.