Making Pull-Based Graph Processing Performant

Samuel Grossman
Stanford University
samuelgr@stanford.edu

Heiner Litz
University of California, Santa Cruz
hlitz@ucsc.edu

Christos Kozyrakis
Stanford University
kozyraki@stanford.edu

Abstract

Graph processing engines following either the push-based or pull-based pattern conceptually consist of a two-level nested loop structure. Parallelizing and vectorizing these loops is critical for high overall performance and memory bandwidth utilization. Outer loop parallelization is simple for both engine types but suffers from high load imbalance. This work focuses on inner loop parallelization for pull engines, which when performed naively leads to a significant increase in conflicting memory writes that must be synchronized.

Our first contribution is a *scheduler-aware* interface for parallel loops that allows us to optimize for the common case in which each thread executes several consecutive iterations. This eliminates most write traffic and avoids all synchronization, leading to speedups of up to 50×.

Our second contribution is the *Vector-Sparse* format, which addresses the obstacles to vectorization that stem from the commonly-used Compressed-Sparse data structure. Our new format eliminates unaligned memory accesses and bounds checks within vector operations, two common problems when processing low-degree vertices. Vectorization with Vector-Sparse leads to speedups of up to 2.5×.

Our contributions are embodied in *Grazelle*, a hybrid graph processing framework. On a server equipped with four Intel Xeon E7-4850 v3 processors, Grazelle respectively outperforms Ligra, Polymer, GraphMat, and X-Stream by up to 15.2×, 4.6×, 4.7×, and 66.8×.

ACM Reference Format:
Samuel Grossman, Heiner Litz, and Christos Kozyrakis. 2018. Making Pull-Based Graph Processing Performant. In *PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3178487.3178506

PPoPP '18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178506

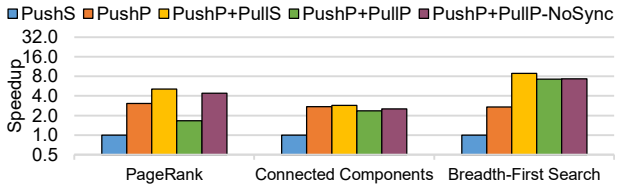


Figure 1. Efficiency of Ligra's inner loop parallelization on the `twitter-2010` graph, running on a four-socket 112-core server. Vertical axis is logarithmic. Ligra parallelizes its loops using Intel Cilk Plus, which uses work-stealing [28].

1 Introduction

Graph problems, which operate on data represented as a set of objects (vertices) and connections (edges), are increasingly important for a wide set of applications including machine learning, social networking, business intelligence, and bioinformatics [22, 26]. They are conventionally viewed as difficult to solve efficiently, especially at scale, as a result of several defining properties: irregularity in the graph datasets that leads to unpredictable memory accesses, many small-sized random accesses, relatively poor data locality, and a typically low amount of work per memory access [45, 65].

Graph processing engines are usually implemented following one of two general patterns: *push* [23, 51, 55, 58, 61, 65, 67] or *pull* [58, 63, 67]. Both patterns conceptually consist of a two-level nested loop. A push engine (Listing 1) propagates out-bound updates grouped by source, so the outer loop iterates over source vertices and the inner loop over the out-edges associated with each source vertex. Conversely, a pull engine (Listing 2) aggregates in-bound updates grouped by destination. Its outer loop iterates over destination vertices, and its inner loop over in-edges. The read-heavy nature of a pull engine's operation leads to higher edge processing throughput than a push engine. However, because the outer loop of a push engine iterates over source vertices, a push engine is better able to leverage the *frontier*, which facilitates skipping over inactive source vertices. This trade-off has given rise to *hybrid* graph frameworks, which contain both engines and dynamically switch between them [2, 3, 58].

To leverage the increasing core counts in processor chips, existing graph processing engines [37, 49, 55, 58, 61, 65, 67] parallelize their respective outer loops. This has the effect of dividing up the source (push) or destination (pull) vertices across multiple threads. Unfortunately, for many real world graphs such as `twitter-2010`, parallelizing only the outer

loop is insufficient as it can lead to significant load imbalance. In such graphs, vertex degrees differ by several orders of magnitude; even with dynamic load balancing and work-stealing [49, 55], threads that process vertices with millions of edges will be slow. This can be seen in Figure 1, which compares different loop parallelization configurations for Ligra [58]. The push engine exhibits a 3× speedup when both loops are parallelized (PushP) over outer loop parallelization alone (PushS). We expect similar gains from parallelizing the inner loop of a pull engine and best performance from a fully-parallelized hybrid configuration. Surprisingly, the opposite is the case. While the sequential pull engine (PushP+PullS) delivers a speedup of up to 9×, the performance of the fully parallelized hybrid (PushP+PullP) is significantly lower.

Our analysis of the pull engine reveals the following challenges. When the inner loop executes serially, each thread repeatedly updates the same destination vertex in cache or processor registers, writing back to shared memory only once upon inner loop completion. When the inner loop is parallelized, threads must conservatively update destination vertices after each iteration of the inner-loop. Furthermore, since multiple threads may update the same destination vertex, synchronization is required. Figure 1 shows that, even if we ignore the need for synchronization (`PushP+PullP-NoSync`), the effect of the numerous conflicting writes is still significant. Write conflicts result in cache line thrashing and spurious write-backs to memory, both of which harm performance [29]. The source of these problems is that the `parallel_for` constructs in popular programming models like Intel Cilk Plus [28] and OpenMP [50] do not allow programmers to optimize for the common case in which several iterations of the parallel loop execute on the same thread. Programmers must pessimistically assume that each iteration will execute independently and treat the loop body as a stateless function with a single parameter, namely the value of the `for` loop iteration counter [39].

Our first contribution is a *scheduler-aware* interface for `parallel_for` (§3) that allows programmers to exploit the chunking behavior of many schedulers, which results in many consecutive iterations of a parallel loop executing on the same thread. This interface provides the flexibility needed to parallelize the inner loop of a pull engine in such a way as to make use of thread-local storage and eliminate synchronization. The scheduler-aware interface considerably improves the performance of a fully-parallelized pull engine without restricting the behavior of the scheduler itself. We observed a peak speedup of 50× with the `uk-2007` graph over a baseline configuration that is parallelized without using scheduler awareness.

In addition to parallelizing the inner loop of the pull engine, we can also vectorize its operation. While some work does target SIMT (single-instruction multiple-thread) vectorization for GPUs [27, 63], we are unaware of any that does so for the SIMD (single-instruction multiple-data) model used on CPUs. Widely-available AVX instructions [30] allow for a maximum speedup of 4× (four 64-bit elements per 256-bit vector). Since graph processing is memory-bound and its memory bandwidth utilization is limited by the low density of memory operations in the instruction stream [4], vectorization can provide additional benefit by issuing multiple load and store operations simultaneously. However, it is hindered by the two-level *Compressed-Sparse* format (Figure 2) frequently used for edge representation [51, 58, 65, 67]. Tight packing of edges leads to unaligned vector loads and stores, which are slower than properly-aligned versions of the same [29]. Moreover, when a thread loads a vector of edges from the edge array, it has to apply additional checks as the four edges may not belong to the same vertex. These two issues are particularly expensive for low-degree vertices and have led recent literature to conclude that vectorization with Compressed-Sparse is not generally performant [5, 43].

Our second contribution, *Vector-Sparse* (§4), is a modified form of the Compressed-Sparse format that enables efficient vectorization of the inner loop using AVX instructions. Vector-Sparse encodes edges together with outer loop information into aligned and padded vectors, which ensures all memory accesses are properly-aligned. Its bit-level representation additionally leverages predicated execution techniques to avoid bounds checks. Inner loop vectorization with Vector-Sparse leads to a speedup of up to 2.5× compared to a non-vectorized implementation.

Our two contributions are embodied in *Grazelle* (§5), a novel hybrid graph processing framework. We show that on a server equipped with four Intel Xeon E7-4850 v3 processors [31] Grazelle respectively outperforms state-of-the-art frameworks Ligra [58], Polymer [67], GraphMat [61], and X-Stream [55] by up to 15.2×, 4.6×, 4.7×, and 66.8×.

Grazelle is publicly available on GitHub. It can be accessed at https://github.com/stanford-mast/Grazelle-PPoPP18.

2 Background

We focus on single-machine, in-memory graph processing engines because of their demonstrated potential to achieve significantly higher performance than distributed [12, 20, 34, 35, 54, 57, 65, 68, 69] and out-of-core [37, 54, 55, 69, 71] engines. Modern servers with several terabytes of DRAM are sufficient for many real-world problems [38, 41]. In this context, a large body of work covers scheduling across cores [49, 55, 65], graph partitioning across sockets [65, 67], and optimizing synchronization and communication [49, 65, 67], typical concerns for any parallel program [36, 56, 64].

Graph applications execute in two conceptual phases: message exchange and local update. During message exchange, *messages* are transmitted along edges from source vertices and aggregated at each destination vertex. During local update, *property values* associated with each vertex (and, in some cases, each edge) are updated based on the aggregated

```
parallel_for (int vSrc = 0; vSrc < numVertices; ++vSrc) {
    if (!frontier.contains(vSrc)) continue;
    for (int d = 0; d < vertex[vSrc].outdegree; ++d) {
        const int vDst = vertex[vSrc].outneighbor[d];
        if (converged.contains(vDst)) continue;
        atomicCAS(vertex[vDst].value,
            compute(vertex[vSrc].value, vertex[vDst].value)); } }
```

Listing 1. Push-based message exchange implementation.

```
parallel_for (int vDst = 0; vDst < numVertices; ++vDst) {
    if (converged.contains(vDst)) continue;
    for (int s = 0; s < vertex[vDst].indegree; ++s) {
        const int vSrc = vertex[vDst].inneighbor[s];
        if (!frontier.contains(vSrc)) continue;
        vertex[vDst].value =
            compute(vertex[vSrc].value, vertex[vDst].value); } }
```

Listing 2. Pull-based message exchange implementation.

incoming messages. Applications execute iteratively, alternating between the two phases until some convergence condition is reached. If a graph processing engine finishes one phase for all vertices before proceeding to the next, then the execution is *synchronous* [62], otherwise it is *asynchronous* [19]. We focus on synchronous engines because of their simplicity. Furthermore, recent work has shown that there is no clear winner between the two types [19, 66].

A common optimization in graph processing is to track the active subset of vertices, known as the *frontier*. The frontier specifies which vertices are involved in a message exchange iteration. Most often, the frontier signifies which vertices should transmit out-bound messages. Only vertices whose property values are modified during one iteration are added to the frontier for the next. Some applications additionally track which vertices have converged to a final value and should therefore ignore all in-bound messages. In Breadth-First Search, for example, vertices are placed into this set immediately upon visitation [2, 3, 58]. Other applications, such as PageRank, cannot use the frontier [23].

A message exchange phase iterates over edges, which are generally grouped either by source or by destination. These two groupings lead to two implementation patterns, *push* (Listing 1) and *pull* (Listing 2). A simple way to parallelize the outer loop for both patterns is to use a `parallel_for` construct such as those offered by Intel Cilk Plus [28] and OpenMP [50]. `atomicCAS()` is an atomic compare-swap operation, and `compute()` is a commutative and associative application-defined operation [19, 58].

Push engines [23, 51, 58, 61, 65, 67] iterate over source vertices and propagate *outbound messages*, whereas pull engines [58, 63, 67] iterate over destination vertices and aggregate *inbound messages*. We refer to the current vertex of the outer loop as the *top-level vertex*. Many graph frameworks use exclusively a push engine [19, 44, 46, 55, 65]. Beamer et. al. [2, 3] motivated the need for both in the context of Breadth-First Search, an approach that Shun and Blelloch have since generalized [58]. A *hybrid* framework contains one engine of each type and, for each iteration, selects which to use based on the state of the frontier. Such a framework generally selects its pull engine whenever a sufficiently large part of the graph is contained in the frontier.

Graph processing engines commonly use the Compressed-Sparse format [40, 51, 58, 65, 67], shown in Figure 2, to

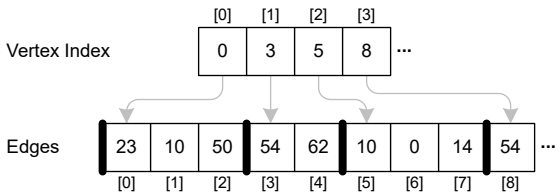


Figure 2. Compressed-Sparse edge data structure.

represent edges. Each instance can either represent in-edges (Compressed-Sparse-Column, or CSC) or out-edges (Compressed-Sparse-Row, or CSR). Thick lines denote boundaries between top-level vertices. The vertex index holds each top-level vertex's starting position in the edge array. One end of each edge can be inferred by position within the index, while the other ends are stored as values in the edge array.

3 Parallelizing with Scheduler Awareness

Our goal is to overcome the two challenges that impede inner loop parallelization of a pull engine: the increased memory write operations and the need for synchronization. The key idea is to structure the pull engine in a manner that is aware of the way schedulers divide and assign parallel work.

Problem: Consider a `parallel_for` loop that executes with multiple threads. Iterations are divided into chunks of potentially varying sizes that are assigned to threads and load-balanced by a scheduler. The application-supplied loop body can be viewed as a `LoopIteration()` function invoked by the threading runtime as shown in Figure 3 (black and red arrows). This function accepts the value of the *iteration index* as a parameter. In a nested `parallel_for` loop, the iteration index includes a value for each loop level. With just an iteration index in hand, the loop body function is necessarily oblivious to the thread locality of surrounding iterations. It must conservatively assume that all iterations of the same loop execute in different threads. Hence, it must write its data into shared memory locations, using synchronization to handle potential conflicts. The function cannot exploit the chunking behavior of many schedulers, in which several consecutive iterations execute on the same thread.

Solution: Figure 3 presents the *scheduler-aware* interface that addresses this problem (black and blue arrows). The interface allows programmers to define how to execute

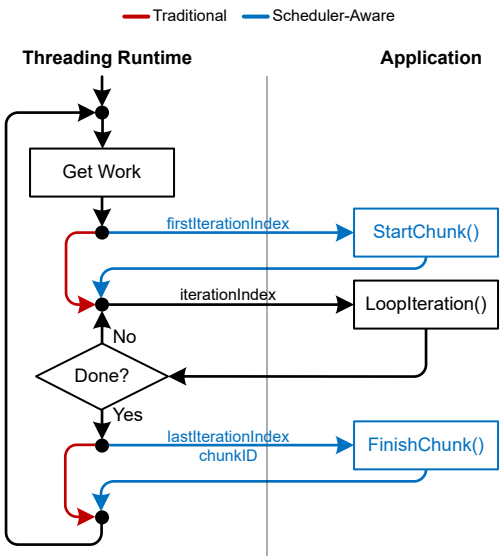


Figure 3. The traditional and scheduler-aware interfaces between the pull engine and the runtime. Text surrounding arrows shows function parameters.

variably-sized chunks of iterations. Hence, in addition to the `LoopIteration()`, the interface includes the application-supplied functions `StartChunk()` and `FinishChunk()` that help initialize and complete the processing of a chunk. First consider the simple case in which a chunk contains some number of iterations from the inner loop. When a thread starts executing the chunk, it uses `StartChunk()` to initialize a variable in thread-local storage (TLS) to store vertex property updates for this chunk (Listing 3). For example, since PageRank uses summation to aggregate, the initial value would be 0. Next, it uses `LoopIteration()` repeatedly to aggregate vertex property updates based on the messages from the edges included in this chunk (Listing 4). Finally, it uses `FinishChunk()` to save updated property value into a global merge buffer indexed by chunk identifier (Listing 5). The merge buffer is preallocated and includes one entry per chunk of iterations created by the scheduler. When all threads have completed processing of all available chunks, a single thread scans the merge buffer and updates the vertex properties in shared memory (Listing 6).

Next, consider the possibility of a chunk containing inner loop iterations from multiple outer loop iterations. To handle this case, `LoopIteration()` checks if the current iteration refers to a different vertex than the last iteration. If so, it stores the aggregated property update for the previous vertex directly into the shared memory location for that vertex and then proceeds to aggregate updates for the new vertex. This store operation to shared memory is safe to execute without synchronization. If the scheduler chunks the iteration space in the nested loop contiguously, there will be at most one

```
TLS.prevDest = vDst;
TLS.prevDestValue = initialValue();
```

Listing 3. Scheduler-aware pull engine, `StartChunk()`.

```
if (TLS.prevDest != vDst) {
    vertices[TLS.prevDest].value = TLS.prevDestValue;
    TLS.prevDest = vDst;
    TLS.prevDestValue = initialValue(); }
TLS.prevDestValue =
    compute(TLS.prevDestValue, vertex[vSrc].value);
```

Listing 4. Scheduler-aware pull engine, `LoopIteration()`.

```
mergeBuffer[chunkID].lastDest = vDst;
mergeBuffer[chunkID].lastValue = TLS.prevDestValue;
```

Listing 5. Scheduler-aware pull engine, `FinishChunk()`.

```
for (int i = 0; i < numChunksExecuted; ++i) {
    const int vDst = mergeBuffer[i].lastDest;
    vertex[vDst] =
        compute(vertex[vDst].value, mergeBuffer[i].lastValue); }
```

Listing 6. Scheduler-aware pull engine, merge operation.

chunk that includes the last few inner-loop iterations for each vertex, so only one thread would execute that store operation. Any other threads targeting the same vertex would write to the merge buffer. This limitation on how the iteration space is chunked is reasonable. It prevents the scheduler from randomizing iterations, which would destroy locality, but does not restrict the number or size of chunks.

Benefits: The scheduler-aware interface greatly improves the efficiency of parallelizing the inner-loop of the pull engine. Memory writes in `LoopIteration()` are almost always to thread-local variables that are captured perfectly by first-level caches and can even be register-allocated. There is no need for synchronization within `LoopIteration()` or `FinishChunk()` as the merge buffer has a seperate slot for each chunk. The final merge (Listing 6) executes sequentially in our implementation because it is extremely fast for the real-world graphs we studied (§6).

The scheduler-aware interface is general enough to be applied to any graph application implemented synchronously using a programming model similar to Gather-Apply-Scatter (GAS) [19] or EDGEMAP/VERTEXMAP [58]. It can be embedded directly into a graph processing framework without substantial impact on the graph application writer. Such impact would be limited to providing an implementation of the `initialValue()` function.

The performance benefit of scheduler awareness depends on the number of write operations performed per inner loop

iteration with the traditional interface, which in turn depends on an application's aggregation operator. For example, PageRank uses summation and thus would perform one write per inner loop iteration, so scheduler awareness is maximally beneficial. Conversely, Connected Components uses minimization and so could skip some writes if the proposed value to write is not less than the value already present for a particular vertex, leading to a reduced benefit. Applications that already perform only a single write per vertex, such as Breadth-First Search, would not benefit at all.

Discussion: The merge buffer has one slot per chunk of iterations. Each slot contains two fields: the last destination vertex in the chunk (`lastDest`) and the partially-aggregated value that was computed for the last destination vertex (`lastValue`). If the threading runtime statically chunks the iteration space, the merge buffer can be allocated at the beginning of the program and reused throughout the execution of the graph application. Note that statically chunking the iteration space does not prohibit the runtime from dynamically assigning and rebalancing chunks across threads. If the runtime changes the assignment of iterations to chunks throughout the runtime, it may need to allocate additional space for merge buffers.

Related Work: Scheduler awareness follows a long line of work that attempts to improve both the data locality and the parallelization of irregular applications. Focusing on the former, we find that the very idea of grouping by top-level vertex, which underlies the operation of both push-based and pull-based engines, stems from Ding and Kennedy's concept of dynamic locality groups [16]. Works that followed have proposed techniques to model the reference locality behavior of programs, reorganized the data layout to enhance reuse, and demonstrated loop transformations [13, 24, 25, 53, 59, 70]. Ideas such as these are manifested in graph processing work as improved techniques for partitioning graphs for cache and NUMA node locality [42, 55, 60, 67].

The second related area of work concerns parallelizing irregular applications. Classic techniques include speculative parallelization [14, 17, 52] and inspector-executor [1, 47], both intended to extract parallelism while minimizing sharing and synchronization overheads. Scheduler awareness shares this goal but differs in approach: it statically transforms parallel loops to avoid sharing altogether rather than attempting to detect it at run-time. However, it is specifically applicable to the type of loop exemplified in Listing 2, which features irregular reads but regular writes.

4 Vectorizing with Vector-Sparse

Our next goal is to overcome the obstacles that Compressed-Sparse poses to vectorization of the inner loop of the pull engine while retaining as much of its compactness as possible. We propose a new format called *Vector-Sparse*. Depending whether top-level vertices are sources or destinations, we say *Vector-Sparse-Source* (VSS) or *Vector-Sparse-Destination* (VSD) respectively. The new format uses a combination of padding and predicated execution to eliminate unaligned accesses and bounds checks when edges are fetched with vector instructions. It also uses a bit-wise encoding that provides fast access to outer-loop information as we traverse the edge array in the inner loop.

Format: Vector-Sparse modifies edge array encoding in Compressed-Sparse. Figure 4 shows its bit-level encoding for 256-bit vectors divided into four 64-bit elements. While the layout is tailored to x86-based processors that support AVX, its underlying ideas are generalizable to other vector architectures and longer vectors (e.g., 512-bit vectors in AVX-512 [33]). Edge weights, though not shown, are supported by appending a weight vector to each edge vector.

Vector-Sparse uses the bottom 48 bits of each 64-bit vector element to encode vertices, which functions exactly as does the edge array in Compressed-Sparse. The upper 16 bits of each element are used for additional information. The valid bit indicates whether the corresponding element contains a valid edge. It is consumed by the AVX operations that support per-element predication [30]. Invalid elements are used as padding so that the number of 64-bit elements per top-level vertex is a multiple of 4 (the vector length). For example, a top-level vertex with a degree of 7 would occupy two 256-bit vectors with 7 valid edge elements and 1 invalid. Its vertex index entry points to the first of the two vectors. Each vector also contains a 48-bit identifier of the top-level vertex, encoded in four fields spread across the four 64-bit elements. This vertex identifier is used when a thread is processing a chunk of edges in order to detect transitions between iterations of the outer loop without bounds checks or accesses to the vertex index.

Vectorization: Listing 7 shows the pseudocode for a vectorized pull engine using VSD as the edge data structure. Frontier checks are omitted for simplicity. Unlike the code shown in Listing 2, the code for shown here uses a single-level loop that iterates over vectors. `extractSources()` and `extractDest()` are macros that extract fields from the format shown in Figure 4. `initialValues()` is a vectorized version of `initialValue()` from Listing 4. `maskedGather()` uses the `vgatherqpd` instruction to perform a gather operation, overwriting each element in `srcVals` predicated on the valid bits loaded to `edges`. `vectorCompute()` is a vectorized version of `compute()` from Listing 2. The vertex index is not used in this loop but remains useful to implement frontier checks. This loop is parallelized across threads per §3.

Related Work: There exists a large body of related work for sparse matrix-vector multiplication (SpMV) [61] that can be applied to graph processing. Both fields leverage the Compressed-Sparse format to efficiently represent sparse matrices [5, 40, 43] and graphs [51, 58, 65, 67].

We are unaware of any prior work that addresses vectorization of graph processing following the SIMD model of

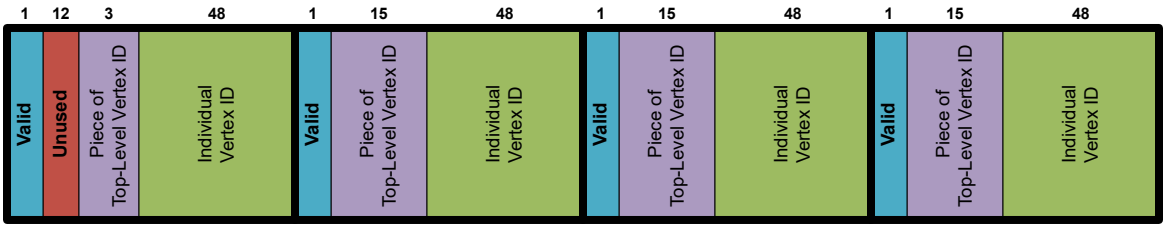


Figure 4. 256-bit edge array element containing up to 4 edges. The number above each field represents its width in bits. Thick lines represent the boundaries between individual edges.

```
for (int i = 0; i < numEdgeVectors; ++i) {
    const v256 edges = edgeVectors[i];
    const v256 vSrc = extractSources(edges);
    const int vDst = extractDest(edges);
    v256 srcVals = initialValues();
    srcVals = maskedGather(vertex[vSrc].value, edges);
    vertex[vDst].value =
        vectorCompute(srcVals, vertex[vDst].value); }
```

Listing 7. Pull engine vectorized with Vector-Sparse.

many CPUs. However, recent studies have shown two ways in which such vectorization can be beneficial. First, Beamer et. al. recently demonstrated that poor memory operation density in the instruction stream is a key performance bottleneck for graph processing [4], a situation improved through vectorization. Second, SpMV is well-understood to be heavily memory-bound [40]; vectorized memory operations take greater advantage of the memory data channel, producing fewer requests per byte transferred and ultimately improving the achievable data transfer rate.

A large body of work in the SpMV community focuses on proposing new ways of representing sparse matrices in memory, with the goal improving compactness and, as a result, memory bandwidth utilization [40]. Several formats glean their efficiency from specific properties of the matrix, such as low variance in the number of non-zero elements [5, 10, 21, 48], which is often a bad assumption in the context of graph processing [19]. Others combine non-zero elements from multiple rows into the same data structure element as an optimization [8, 9, 43]. These types of layouts would preclude effective utilization of the frontier, which depends on the ability to associate large sections of the edge data structure uniquely with specific top-level vertices so that they can be skipped easily.

5 Grazelle

Grazelle is a hybrid graph processing framework that embodies the two contributions described in §3 and §4. Its pull engine is parallelized using the scheduler-aware model, whereas its push engine uses the traditional approach. In both cases, threads are created and managed by direct invocation of `pthreads` functions. Grazelle is implemented in 3 KLOC of C code and 2 KLOC of x86 assembly.

Grazelle's programming model is based on Gather-Apply-Scatter (GAS) [19] and EDGEMAP/VERTEXMAP [58]. It defines two processing phases, *Edge* (message exchange) and *Vertex* (local update), each terminated by a thread barrier. The Edge phase has two implementations, Edge-Push and Edge-Pull, based on the engine pattern (push or pull).

Key data structures include arrays for vertex properties and edge lists represented using Vector-Sparse. As with previous work [58, 67], we use two edge lists, one grouped by source vertex (VSS) and the other grouped by destination (VSD). Vertex property arrays are indexed using the vertex's 48-bit identifier. Grazelle additionally supports global variables as a convenience to the graph application writer. These variables can be used for any purpose; values are produced during one phase and can be consumed upon that phase's completion. The remainder of this section covers implementation details not specifically related to scheduler awareness or vectorization.

Frontier Tracking: Grazelle represents the frontier densely as a bit-mask containing one bit per vertex indexed by vertex identifier. We selected a bit-mask because it is extremely compact and trivial to search: 1 billion vertices would only require 125 MB, and the `tzcnt` instruction [30] enables searching through 64 vertices with just a single instruction. Unlike Grazelle, other engines support dynamically switching between sparse and dense representations for frontiers [58, 67]. We quantify the impact of this implementation issue in §6.3 but otherwise leave it to future work.

Multi-core and NUMA Support: Grazelle pins one software thread to each hardware thread (logical core) available in the system. Each thread is aware of its own *group* (set of threads that share a NUMA node), *local thread ID* within the group, and *global thread ID*. For work assignment and accesses to node-local data structures, threads simply refer to their local IDs. Accesses to globally-shared data are based on a thread's global ID, and thread barriers involve all threads.

The Edge phase is parallelized using a dynamic scheduler that splits the edge vector array into equally-sized chunks and assigns chunks to threads as they become available.

Through experimentation we found that creating $32n$ chunks, where n is the number of threads, achieved near-ideal load balance. Chunk size is dependent on the input graph, but scheduling overheads are not. We selected a dynamic scheduler over a static scheduler because the latter would suffer from variability in work per edge, as the structure of the graph affects vertex locality, which in turn creates differences in the time taken to execute each edge's workload. The Vertex phase is statically scheduled by dividing the vertices into equal-sized chunks, one chunk per thread. The work is sufficiently regular that load balancing is not a problem.

Grazelle optimizes for NUMA by performing light-weight graph partitioning: we divide the edge vector array into equally-sized pieces, place each piece in locally-allocated memory on each NUMA node, and generate a separate vertex index for each NUMA node's piece. Since edges are grouped and sorted in ascending order by top-level vertex, we can easily determine the range of source (Edge-Push) or destination (Edge-Pull) vertices that each NUMA node will encounter during the Edge phase. We use the latter to distribute the vertex property arrays over NUMA nodes in a manner similar to that of Polymer: each array is contiguous in virtual address space, but address translation allows different parts of the backing physical memory to be located on different NUMA nodes [67]. During the Vertex phase, each NUMA node only updates vertices whose property values are locally allocated on it, subject to some flexibility to avoid overlaps and ensure proper alignment. We do not evaluate the effectiveness of our approach and instead refer interested readers to Polymer's evaluation of the impact of NUMA on graph partitioning [67].

6 Evaluation

We used a 4-socket server with Intel Xeon E7-4850 v3 processors (14 physical/28 logical cores and 35 MB LLC) [31] and 1 TB of DRAM, running Ubuntu 14.04 LTS. To clearly separate issues, §6.1 and §6.2 respectively evaluate scheduler awareness and Vector-Sparse mostly using a single socket, while the final comparison with existing work in §6.3 uses all four sockets to also capture NUMA issues.

Input datasets are 6 real-world graphs listed in Table 1. They are taken from a variety of application areas and feature a wide variety of sizes and distributions [18, 22, 26]. `dimacs-usa` and `twitter-2010` are commonly used in related work [37, 49, 55, 58, 67]. `dimacs-usa` is unique in that it is a mesh network, having relatively small and consistent vertex degrees. The others are scale-free graphs with different degree distributions. The in-degree distribution of `uk-2007` is the most skewed of all our input graphs; compared to `twitter-2010`, `uk-2007` contains over 10× more vertices having in-degree of at least 100,000 [6, 7]. Our plots refer to the graphs by abbreviation.

Table 1. Graph datasets used for our evaluation.

Abbr.	Name	Vertices	Edges
C	`cit-Patents` [41]	3.7M	16.5M
D	`dimacs-usa` [15]	23.9M	58.3M
L	`livejournal` [41]	4.8M	69.0M
T	`twitter-2010` [6, 7]	41.7M	1.47B
F	`friendster` [41]	65.6M	1.81B
U	`uk-2007` [6, 7]	105.9M	3.74B

GraphMat and Polymer, two existing frameworks to which we compare Grazelle, crash when we attempt to process the `uk-2007` graph. GraphMat indexes edges using 32-bit signed integers, hence it cannot load correctly a graph with over 3 billion edges. Polymer appears to suffer from an implementation bug. None of Polymer's originally-published results use graphs with more than 3 billion edges [67].

Our evaluation focuses on 3 graph applications: PageRank (PR), Connected Components (CC), and Breadth-First Search (BFS). These applications represent diverse behaviors both in terms of memory accesses and frontier utilization. PageRank does not use the frontier and uses summation as its aggregation operator, so vertex property values are updated every iteration. It therefore can be used to measure peak processing throughput. Connected Components uses the frontier to activate and deactivate source vertices, thus exhibiting the most common type of frontier utilization. Its aggregation operator is minimization, which sometimes allows it to skip memory write operations if the proposed value to be written is not actually less than the value already present for a vertex property. Breadth-First Search is a completely frontier-driven application. In addition to source vertex activation and deactivation, it also marks vertices as converged immediately upon their visitation. Only a single write operation is ever needed per vertex: the first identified candidate to be a vertex's parent becomes its final value.

We omit other applications due to space limitations and because they would not add further information about the effectiveness of scheduler awareness or Vector-Sparse. For example, Collaborative Filtering is very similar to PageRank in that it does not use the frontier, but differs as it uses edge weights and supplies a different mathematical formula for updates to property values [23]. The use of edge weights adds additional transfers but does not change the access pattern, and the use of a different kernel simply means an alternate sequence of arithmetic instructions in the inner loop. Likewise, Single-Source Shortest-Paths uses edge weights and initializes the frontier to contain just a single vertex. It otherwise behaves the same way as Connected Components, all the way down to the use of minimization as its aggregation operator [58]. The only effect of a difference in frontier fullness is biassing the execution towards either push or pull.

6.1 Effectiveness of Scheduler Awareness

Scheduler awareness eliminates write conflicts and reduces the number of write operations a pull engine needs to perform. We expect it to be at its most beneficial when writes and conflicts are common. We therefore begin our analysis with a detailed look at PageRank, followed by some insights as to its impact on Connected Components. Because Breadth-First Search performs one write per vertex and writes do not conflict, it is unaffected by scheduler awareness.

We evaluate the effectiveness of scheduler awareness by comparing two pull engine configurations: one parallelized using a scheduler-aware interface (Listings 3, 4, 5, and 6) and one parallelized using a traditional interface (Listing 2 with the inner `for` changed to `parallel_for` and appropriate atomics added). With the traditional interface, the probability of write conflicts depends on the number of threads used, the degree of vertices processed, and the scheduler granularity (i.e. the number of edges per chunk). Conversely, the scheduler-aware interface must perform a merge operation at the end of the nested loop. The incurred overhead depends on the scheduler granularity (i.e. the number of chunks created).

Figure 5 quantifies the performance impact of scheduler awareness for PageRank using the 6 input graphs with a fixed scheduler granularity of 1,000 edge vectors per chunk, which roughly approximates the default maximum chunk size Cilk Plus would use for any of these graphs [32]. For reference, we also show results for the traditional approach without synchronization, even though it leads to incorrect output. Scheduler awareness is clearly beneficial across the board, irrespective of graph dimensions. The largest benefit is for `uk-2007`: the write conflicts with the traditional interface are sufficiently prevalent that scheduler awareness improves performance by 50×. For such consistently low-degree graphs as `dimacs-usa` the speedup drops as low as 15%; in these cases, scheduler awareness removes all synchronization but does not significantly reduce the actual number of writes performed.

Figure 6 quantifies the sensitivity of PageRank performance to chunk size for three representative graphs. Performance with the traditional interface is often strongly dependent on the chunk size, particularly for scale-free graphs with frequent high-degree vertices. The ideal chunk size is graph-dependent, and simply switching to a large chunk size is undesirable because doing so can lead to load imbalance. Conversely, performance with the scheduler-aware interface is largely insensitive to chunk size.

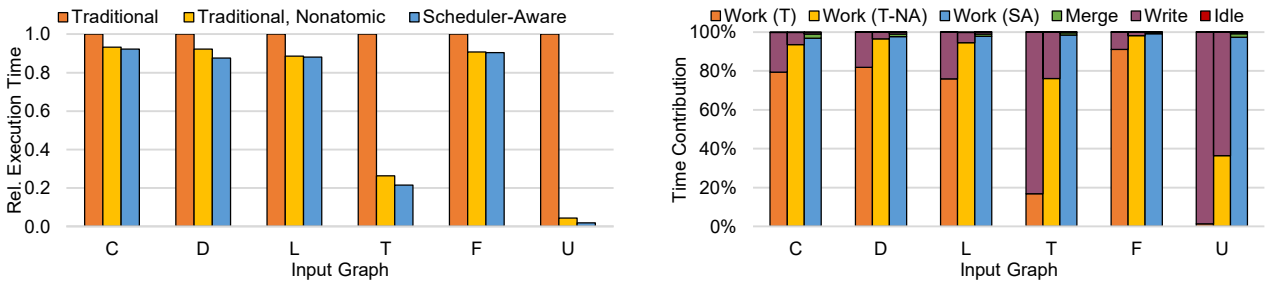
Figure 7 illustrates how scheduler awareness improves multi-core scalability for the same graphs as in Figure 6 by showing performance as we increase the number of active physical cores and NUMA nodes. In each test involving multiple NUMA nodes, the number of active physical cores per node is kept equal. The chunk size is selected for each graph based on its result in Figure 6, with the goal of picking a granularity that produces similar performance between the two interfaces. All values are normalized to the performance result of the traditional interface with a single thread. As expected, scheduler awareness is most effective for graphs with greater numbers of vertices having high in-degree. In fact, without scheduler awareness the performance of PageRank on `uk-2007` barely scales with increasing thread count. Nevertheless, even low-degree graphs can benefit from scheduler awareness, as reflected in the results for `dimacs-usa`.

We turn our attention now to Connected Components, which has lower write intensity than PageRank. To isolate the impact of the reduced write intensity, we present results for two versions: one implemented as described and a modified version with higher write intensity. The latter unconditionally writes values to vertex properties, even if the value to be written is equal to the value already present. Due to space limitations, Figure 8 presents only end-to-end performance results using Grazelle's default scheduling granularity (§5) on a single socket with all 28 logical cores active.

Despite its reduced write intensity, Connected Components clearly benefits from scheduler awareness. `cit-Patents`, for instance, exhibits a speedup of 40%. Scheduler awareness is unsurprisingly more effective with the modified version, resulting in a speedup of up to 2.4×. In the worst case, there is a 3% slowdown for `uk-2007` in Figure 8b. This occurs because Grazelle's default scheduling granularity is coarse for this graph (approximately 1 million vectors per chunk), meaning that the number of write conflicts is quite small with the traditional interface.

6.2 Effectiveness of Vector-Sparse

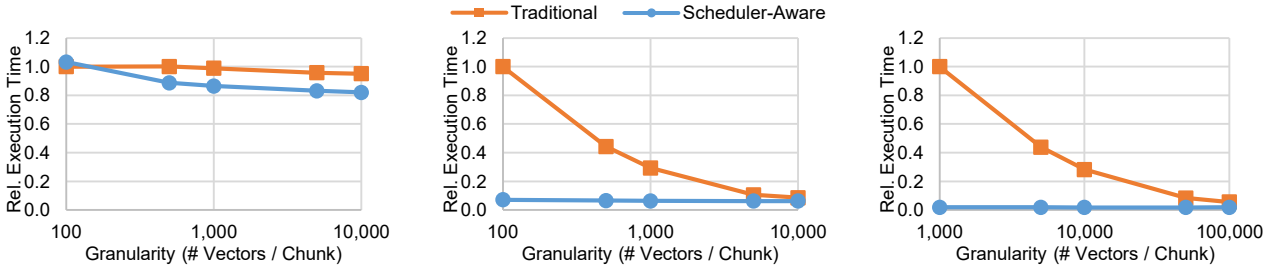
Vector-Sparse adds padding to the very compact Compressed-Sparse layout in order to better support vectorization. In other words, it trades off some compactness for performance. Both the compactness loss and the performance gain depend on the average packing efficiency of the edge vectors. Packing efficiency is the percentage of valid bits set per vector. For a 4-element vector, it ranges from 25% (only one edge is valid) to 100% (all four edges are valid). Figure 9a shows the average edge vector packing efficiency across all 6 of our real-world datasets. Figure 9b shows the same for a total of 30 synthetic graphs generated with the R-MAT generator [11] included in X-Stream [55]. We show results with 4-, 8-, and 16-element vectors (256-, 512-, and 1024-bit vectors) to evaluate the effectiveness of Vector-Sparse with current and future processors. Many real-world graphs, including both `twitter-2010` and `uk-2007`, have an average degree of at least 2^5, which leads to an average packing efficiency of well over 90% for 4-element vectors and close to that even for 8-element vectors. With 4-element vectors, packing efficiency is at least 75% in nearly all cases, suggesting potentially large benefits from vectorization. Unsurprisingly, packing efficiency drops with wider vectors.



(a) Execution time relative to the traditional interface. Lower is better.

(b) Execution time profile for each scheduler interface.

Figure 5. Performance impact of scheduler awareness on PageRank with a scheduling granularity of 1,000 edge vectors per chunk. T = Traditional; T-NA = Traditional, Nonatomic; SA = Scheduler-Aware.

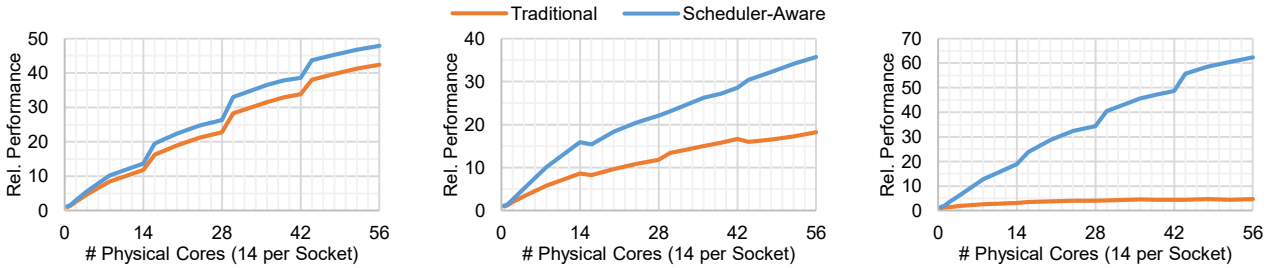


(a) `dimacs-usa`

(b) `twitter-2010`

(c) `uk-2007`

Figure 6. Sensitivity of PageRank performance to chunk size. Horizontal axis is logarithmic. Granularities for `uk-2007` are 10× those of other graphs. Baselines are traditional interface results for the smallest shown granularities. Lower is better.



(a) `dimacs-usa`, granularity 5,000

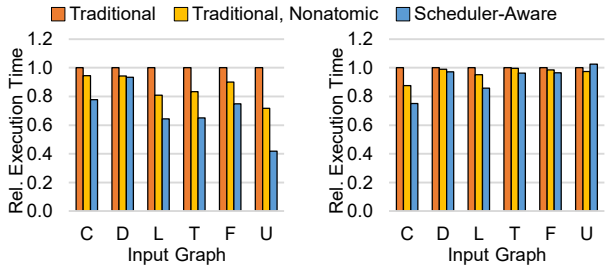
(b) `twitter-2010`, granularity 5,000

(c) `uk-2007`, granularity 50,000

Figure 7. Multi-core scaling of different scheduler interfaces with PageRank. Represented as performance relative to that of the traditional interface run with a single thread. Higher is better.

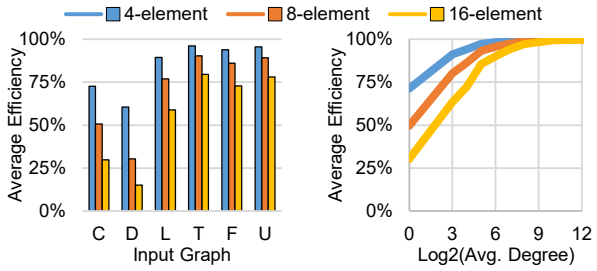
We evaluate performance gains from vectorization with 4-element AVX vectors by comparing vectorized implementations of each phase with non-vectorized implementations of the same. Our non-vectorized implementation of the Vertex phase targets only a single vertex per iteration. In the Edge phase, we disable vectorization by replacing vectorized code, such as the `vgatherqpd` instruction, with versions that process a single edge at a time.

We show results separated by Grazelle phase when running PageRank (Figure 10a) and as end-to-end speedups across the three applications (Figure 10b). Edge-Pull is clearly the most responsive to vectorization, showing speedups of approximately 2× irrespective of input. Edge-Push and Vertex are largely unresponsive to vectorization, the former due to the lack of AVX atomic-update-scatter instructions and the latter because of memory bandwidth saturation. PageRank benefits the most from vectorization because Grazelle exclusively selects Edge-Pull for its execution. Benefits for other applications depends on the extent to which they use Edge-Pull, which in turn depends on the frontier size.



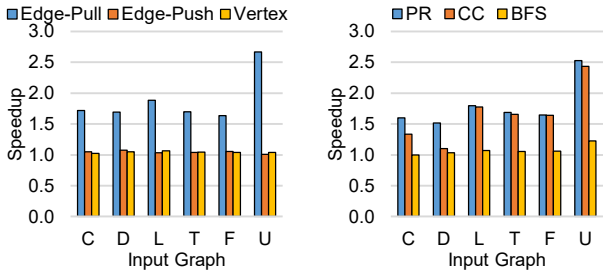
(a) Write-intense version **(b)** Standard version

Figure 8. Performance impact of scheduler awareness on Connected Components with Grazelle's default scheduler granularity. Shown as execution time relative to the traditional interface. Lower is better.



(a) Real-world graphs **(b)** Synthetic graph suite

Figure 9. Packing efficiency for vectors of various lengths.



(a) By PageRank phase **(b)** End-to-end by application

Figure 10. Performance impact of vectorization relative to each corresponding non-vectorized implementation.

6.3 Comparison with Existing Graph Frameworks

We compare Grazelle to Ligra version 1.5, a July 2015 snapshot of Polymer, GraphMat version 1.0, and in-memory X-Stream version 1.0. Ligra's pull engine is the state-of-the-art for a CPU-based implementation, Polymer is a NUMA-aware derivative of Ligra, GraphMat has previously been cited as being the best-performing framework [23, 61]. X-Stream is unique in that it is an edge-centric framework: it creates cache-sized *streaming partitions* from an unordered list of edges and performs in-memory shuffle operations to exchange messages between them [55].

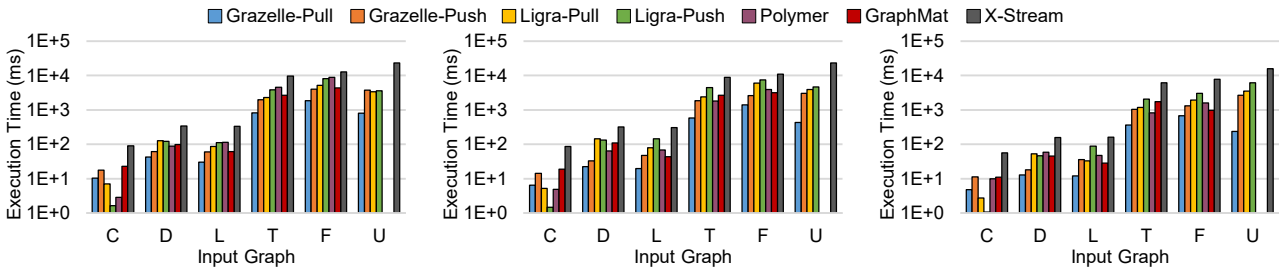
Per-application results are shown in Figures 11, 12, and 13. Lower execution time is better. PageRank results are shown individually for the push-based and pull-based engines of Grazelle and Ligra; Polymer's implementation exclusively uses a push-based engine, and GraphMat does not contain a pull-based engine. Connected Components and Breadth-First Search results are shown for both Ligra and *Ligra-Dense*, a modified version of Ligra that maintains engine switching functionality but uses only a dense frontier representation. One of Ligra's key optimizations is the use of both sparse and dense frontier representations, a feature not implemented in Grazelle, so we include Ligra-Dense results to facilitate a fairer comparison.

With the exception of the small `cit-Patents` graph, PageRank results clearly favor Grazelle's pull-based engine, which outperforms Ligra's pull-based engine, Polymer, GraphMat, and X-Stream by up to 15.2×, 4.6×, 4.7×, and 66.8×[1] respectively, with all four sockets active. Grazelle is the only framework that uses a pull engine with a parallelized and vectorized inner loop, so the results reflect the significant processing throughput gains by using scheduler awareness and Vector-Sparse. Interestingly, Grazelle's push-based engine generally outperforms GraphMat, the biggest difference being 3.6× with all four sockets active. X-Stream's performance is uncompetitive, likely due to the significant overheads of its in-memory shuffle operations.

Connected Components results show Grazelle respectively outperforming Ligra, Ligra-Dense, Polymer, GraphMat, and X-Stream by up to 14.6×, 13.3×, 4.1×, 12.2×, and 223.2× with all four sockets active. These results showcase the higher processing throughput of Grazelle's pull engine even when frontiers shift some iterations to the push engine. GraphMat is built on an engine intended for sparse matrix-vector multiplication and therefore does not handle the frontier as efficiently as the other frameworks, hence its comparatively reduced performance. X-Stream is hampered both by its relatively low processing throughput (Figure 11) and its inferior frontier-handling efficiency: an update targeting a vertex in a particular streaming partition requires loading and processing the entire partition, not just the individual vertex being updated [55].

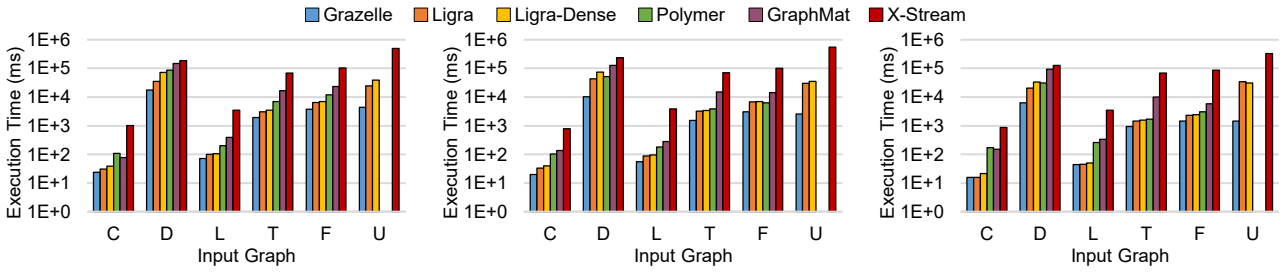
Breadth-First Search results mostly favor Ligra due to its sparse frontier optimization, which is particularly helpful because the frontier is often nearly empty. Grazelle's performance is generally similar to that of Ligra-Dense. This indicates that our proposed optimizations do not display any sort of fundamental incompatibility with applications that extensively use frontier optimizations, even though they do not specifically benefit such applications. Execution times

[1] X-Stream requires that the number of threads be a power of two. It can therefore only scale to 16 out of the available 28 logical cores per socket.



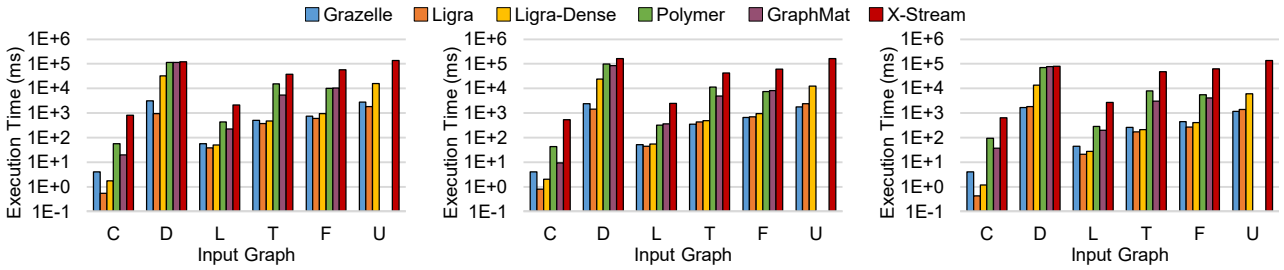
(a) 1 socket **(b)** 2 sockets **(c)** 4 sockets

Figure 11. PageRank per-iteration execution time comparison. Lower is better. All plots use a logarithmic scale.



(a) 1 socket **(b)** 2 sockets **(c)** 4 sockets

Figure 12. Connected Components execution time comparison. Lower is better. All plots use a logarithmic scale.



(a) 1 socket **(b)** 2 sockets **(c)** 4 sockets

Figure 13. Breadth-First Search execution time comparison. Lower is better. All plots use a logarithmic scale.

produced by Polymer, GraphMat, and X-Stream are uncompetitive with Ligra or Grazelle. Polymer's implementation uses exclusively its pull-based engine, and both GraphMat and X-Stream suffer from the same issues as with Connected Components.

7 Conclusion

We presented two optimization strategies that improve the performance of pull-based graph processing engines. The *scheduler-aware* interface for parallel loops allows us to eliminate most write traffic and synchronization operations when parallelizing the inner loop, leading to speedups of up to 50×. The *Vector-Sparse* format enables inner loop vectorization, which improves performance by up to 2.5×. We implemented both in *Grazelle*, a hybrid graph processing framework that respectively outperforms state-of-the-art graph processing frameworks Ligra, Polymer, GraphMat, and X-Stream by up to 15.2×, 4.6×, 4.7×, and 66.8×, even in the presence of frontier optimizations.

Acknowledgements

We thank our anonymous reviewers and our shepherd, Michelle Goodstein, for their feedback and assistance in improving our paper. This work is supported by the National Science Foundation (grant number SHF-1408911), the Stanford Platform Lab, Samsung, and Huawei.

References

[1] Manuel Arenaz, Juan Touriño, and Ramón Doallo. 2004. An Inspector-Executor Algorithm for Irregular Assignment Parallelization. In *ISPA '04*. Springer Berlin Heidelberg, 4–15. https://doi.org/10.1007/978-3-540-30566-8_4

[2] Scott Beamer, Krste Asanović, and David A. Patterson. 2011. *Searching for a parent instead of fighting over children: A fast breadth-first search implementation for Graph500*. Technical Report. EECS Department, University of California, Berkeley.

[3] Scott Beamer, Krste Asanović, and David A. Patterson. 2012. Direction-optimizing Breadth-First Search. In *SC '12*. IEEE Computer Society, 1–10. https://dx.doi.org/10.1109/SC.2012.50

[4] Scott Beamer, Krste Asanović, and David A. Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *IISWC '15*. IEEE, 56–65. https://dx.doi.org/10.1109/IISWC.2015.12

[5] Nathan Bell and Michael Garland. 2009. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *SC '09*. ACM, 18:1–18:11. https://dx.doi.org/10.1145/1654059.1654078

[6] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW '11*. ACM, 587–596.

[7] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW '04*. ACM, 595–601.

[8] Aydın Buluç, Jeremy Fineman, Matteo Frigo, John Gilbert, and Charles Leiserson. 2009. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication using Compressed Sparse Blocks. In *SPAA '09*. ACM, 233–244. https://doi.org/10.1145/1583991.1584053

[9] Aydın Buluç, Samuel Williams, Leonid Oliker, and James Demmel. 2011. Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication. In *IPDPS '11*. IEEE, 721–733. https://doi.org/10.1109/IPDPS.2011.73

[10] Wei Cao, Lu Yao, Zongzhe Li, Yongxian Wang, and Zhenghua Wang. 2010. Implementing Sparse Matrix-Vector Multiplication using CUDA based on a Hybrid Sparse Matrix Format. In *ICCASM '10*. IEEE, V11–161–V11–165. https://dx.doi.org/10.1109/ICCASM.2010.5623237

[11] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *SDM '04*. SIAM. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.215.7520

[12] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *EuroSys '15*. ACM, 1:1–1:15. https://dx.doi.org/10.1145/2741948.2741970

[13] Trishul M. Chilimbi. 2001. Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality. In *PLDI '01*. ACM, 191–202. https://dx.doi.org/10.1145/378795.378840

[14] Francis Dang, Hao Yu, and Lawrence Rauchwerger. 2002. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *IPDPS '02*. IEEE. https://doi.org/10.1109/IPDPS.2002.1015493

[15] Camil Demetrescu. 2010. 9th DIMACS Implementation Challenge. http://www.dis.uniroma1.it/challenge9/download.shtml. (2010).

[16] Chen Ding and Ken Kennedy. 1999. Improving Cache Performance in Dynamic Applications Through Data and Computation Reorganization at Run Time. In *PLDI '99*. ACM, 229–241. https://dx.doi.org/10.1145/301618.301670

[17] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. 2007. Software Behavior Oriented Parallelization. In *PLDI '07*. ACM, 223–234. https://doi.org/10.1145/1250734.1250760

[18] Benedikt Elser and Alberto Montresor. 2013. An Evaluation Study of BigData Frameworks for Graph Processing. In *BigData '13*. IEEE, 60–67. https://dx.doi.org/10.1109/BigData.2013.6691555

[19] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI '12*. USENIX, 17–30. https://www.usenix.org/node/180251

[20] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI '14*. USENIX, 599–613. https://www.usenix.org/node/186216

[21] Roger Grimes, David Kincaid, and David Young. 1979. *ITPACK 2.0: User's Guide*. Technical Report. University of Texas, Austin.

[22] Yong Guo, Marcin Biczak, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L. Willke. 2014. How Well do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis. In *IPDPS '14*. IEEE, 395–404. https://dx.doi.org/10.1109/IPDPS.2014.49

[23] 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 *MICRO '16*. IEEE, 1–13. https://dx.doi.org/10.1109/MICRO.2016.7783759

[24] Hwansoo Han and Chau-Wen Tseng. 2000. A Comparison of Locality Transformations for Irregular Codes. In *LCR '00*. Springer Berlin Heidelberg, 70–84. https://doi.org/10.1007/3-540-40889-4_6

[25] Hwansoo Han and Chau-Wen Tseng. 2006. Exploiting Locality for Irregular Scientific Codes. *IEEE Transactions on Parallel and Distributed Systems* 17, 7 (June 2006), 606–618. https://doi.org/10.1109/TPDS.2006.88

[26] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. 2014. An Experimental Comparison of Pregel-like Graph Processing Systems. *Proc. VLDB Endowment* 7, 12 (August 2014), 1047–1058. https://dx.doi.org/10.14778/2732977.2732980

[27] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *PACT '11*. IEEE, 78–88. https://dx.doi.org/10.1109/PACT.2011.14

[28] Intel. 2014. CilkPlus. https://www.cilkplus.org/. (2014).

[29] Intel. 2015. Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html. (2015).

[30] Intel. 2015. Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. (2015).

[31] Intel. 2015. Intel Xeon Processor E7-4850 v3. http://ark.intel.com/products/84679. (2015).

[32] Intel. 2016. cilk grainsize. https://software.intel.com/en-us/node/684195. (2016).

[33] Intel. 2017. Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf. (2017).

[34] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. 2009. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *ICDM '09*. IEEE, 229–238. https://dx.doi.org/10.1109/ICDM.2009.14

[35] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *EuroSys '13*. ACM, 169–182. https://dx.doi.org/10.1145/2465351.2465369

[36] Arvind Krishnamurthy and Katherine Yelick. 1995. Optimizing Parallel Programs with Explicit Synchronization. In *PLDI '95*. ACM, 196–204. https://dx.doi.org/10.1145/207110.207142

[37] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI '12*. USENIX, 31–46. https://www.usenix.org/node/180252

[38] Laboratory for Web Algorithmics. 2012. Datasets. http://law.di.unimi.it/datasets.php. (2012).

[39] James LaGrone, Ayodunni Aribuki, Cody Addison, and Barbara Chapman. 2011. A Runtime Implementation of OpenMP Tasks. In *IWOMP '11*. Springer Berlin Heidelberg, 165–178. https://doi.org/10.1007/978-3-642-21487-5_13

[40] Daniel Langr and Tvrdík. 2016. Evaluation Critera for Sparse Matrix Storage Formats. *IEEE Transactions on Parallel and Distributed Systems* 27, 2 (February 2016), 428–440. https://dx.doi.org/10.1109/TPDS.2015.2401575

[41] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (2014).

[42] Lingda Li, Robel Geda, Ari B. Hayes, Yanhao Chen, Pranav Chaudhari, Eddy Z. Zhang, and Mario Szegedy. 2017. A Simple Yet Effective Balanced Edge Partition Model for Parallel Computing. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1, 1 (June 2017), 14:1–14:21. https://doi.org/10.1145/3084451

[43] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient Sparse Matrix-Vector Multiplication on x86-Based Many-Core Processors. In *ICS '13*. ACM, 273–282. https://dx.doi.org/10.1145/2464996.2465013

[44] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endowment* 5, 8 (April 2012), 716–727. https://dx.doi.org/10.14778/2212351.2212354

[45] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in Parallel Graph Processing. *Parallel Processing Letters* 17, 1 (March 2007), 5–20. http://www.worldscientific.com/doi/abs/10.1142/S0129626407002843

[46] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *SIGMOD '10*. ACM, 135–146. https://dx.doi.org/10.1145/1807167.1807184

[47] María J. Martín, David E. Singh, Juan Touriño, and Francisco F. Rivera. 2002. Exploiting Locality in the Run-Time Parallelization of Irregular Loops. In *ICPP '02*. IEEE, 27–34. https://doi.org/10.1109/ICPP.2002.1040856

[48] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In *HiPEAC '10*. Springer Berlin Heidelberg, 111–125. https://doi.org/10.1007/978-3-642-11515-8_10

[49] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *SOSP '13*. ACM, 456–471. https://dx.doi.org/10.1145/2517349.2522739

[50] OpenMP ARB. 2016. OpenMP. http://www.openmp.org/. (2016).

[51] 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 ATC '12*. USENIX, 41–52. http://dl.acm.org/citation.cfm?id=2342821.2342825

[52] Lawrence Rauchwerger and David A. Padua. 1999. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Transactions on Parallel and Distributed Systems* 10, 2 (February 1999), 160–180. https://doi.org/10.1109/71.752782

[53] Array Regrouping and Structure Splitting Using Whole-Program Reference Affinity. 2004. Zhong, Yutao and Orlovich, Maksim and Shen, Xipeng and Ding, Chen. In *PLDI '04*. ACM, 255–266. https://dx.doi.org/10.1145/996841.996872

[54] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out Graph Processing from Secondary Storage. In *SOSP '15*. ACM, 410–424. https://dx.doi.org/10.1145/2815400.2815408

[55] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *SOSP '13*. ACM, 472–488. https://dx.doi.org/10.1145/2517349.2522740

[56] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. 1991. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *SPAA '91*. ACM, 237–245. https://dx.doi.org/10.1145/113379.113401

[57] Semih Salihoglu and Jennifer Widom. 2013. GPS: A Graph Processing System. In *SSDBM '13*. ACM, 22:1–22:12. https://dx.doi.org/10.1145/2484838.2484843

[58] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *PPoPP '13*. ACM, 135–146. https://dx.doi.org/10.1145/2442516.2442530

[59] Michelle M. Strout, Larry Carter, and Jeanne Ferrante. 2001. Rescheduling for Locality in Sparse Matrix Computations. In *ICCS '01*. Springer Berlin Heidelberg, 137–146. https://doi.org/10.1007/3-540-45545-0_23

[60] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. Accelerating Graph Analytics by Utilising the Memory Locality of Graph Partitioning. In *ICPP '17*. IEEE, 181–190. https://doi.org/10.1109/ICPP.2017.27

[61] 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 Endowment* 8, 11 (July 2015), 1214–1225. https://dx.doi.org/10.14778/2809974.2809983

[62] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (August 1990), 103–111. https://dx.doi.org/10.1145/79173.79181

[63] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2015. Gunrock: A High-performance Graph Processing Library on the GPU. In *PPoPP '15*. ACM, 265–266. https://dx.doi.org/10.1145/2688500.2688538

[64] Marc H. Willebeek-LeMair and Anthony P. Reeves. 1993. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems* 4, 9 (September 1993), 979–993. https://dx.doi.org/10.1109/71.243526

[65] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GraM: Scaling Graph Computation to the Trillions. In *SoCC '15*. ACM, 408–421. https://dx.doi.org/10.1145/2806777.2806849

[66] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. SYNC or ASYNC: Time to Fuse for Distributed Graph-Parallel Computation. In *PPoPP '15*. ACM, 194–204. https://dx.doi.org/10.1145/2688500.2688508

[67] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware Graph-structured Analytics. In *PPoPP '15*. ACM, 183–193. https://dx.doi.org/10.1145/2688500.2688507

[68] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. 2016. Exploring the Hidden Dimension in Graph Processing. In *OSDI '16*. USENIX, 285–300. https://www.usenix.org/node/199311

[69] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *FAST '15*. USENIX, 45–58. https://www.usenix.org/conference/fast15/technical-sessions/presentation/zheng

[70] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program Locality Analysis using Reuse Distance. *ACM Transactions on Programming Languages and Systems* 31, 6 (August 2009), 20:1–20:39. https://dx.doi.org/10.1145/1552309.1552310

[71] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *ATC '15*. USENIX, 375–386. https://www.usenix.org/node/190490

A Artifact Description

A.1 Abstract

This artifact contains all of the source code for Grazelle and the datasets used to evaluate it in the PPoPP 2018 paper *Making Pull-Based Graph Processing Performant*. Validating our results requires compiling and running Grazelle on these datasets. Our Makefile-based build system simplifies the task of replicating our results by pre-setting several experiment settings based on the figure numbers in the paper.

Grazelle requires an x86-64-based processor with support for AVX2 instructions (i.e. Intel Haswell or later), and NUMA tests require multiple sockets. We recommend at least 256GB DRAM per socket. Grazelle runs on Ubuntu 14.04 or later with glibc, libnuma (package "libnuma-dev"), and pthreads. For building, Grazelle requires gcc 4.8.4 or later and either as 2.24 or nasm 2.10.09 or later, as well as the "make" tool.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** PageRank, Connected Components, Breadth-First Search
- **Program:** C and assembly code
- **Compilation:** gcc 4.8.5, as 2.24 or nasm 2.10.09
- **Binary:** Not included; provided source code compiles to an executable targeting x86-64 CPUs that support AVX2 instructions
- **Data set:** Publicly available datasets listed in the paper, converted to Grazelle's binary graph format
- **Run-time environment:** Ubuntu 14.04 or later with glibc, libnuma (package "libnuma-dev"), and pthreads
- **Hardware:** x86-64-based processor with support for AVX2 instructions (i.e. Intel Haswell or later); NUMA tests require multiple sockets; recommended 256GB DRAM per socket
- **Execution:** Run the Grazelle binary with appropriate command-line flags
- **Output:** Execution statistics such as time, vector packing efficiency, and so on are printed to standard output
- **Publicly available:** Yes

A.3 Description

A.3.1 How Delivered

Our artifact is publicly available on GitHub. It can be accessed at https://github.com/stanford-mast/Grazelle-PPoPP18.

A.3.2 Hardware Dependencies

Grazelle requires an x86-64-based CPU with support for AVX2 instructions, such as Intel CPUs of Haswell generation or later. NUMA scaling experiments require multiple CPU sockets. We recommend 256GB DRAM per socket.

A.3.3 Software Dependencies

Ubuntu 14.04 or later with glibc, libnuma (package "libnuma-dev"), and pthreads; make; gcc 4.8.4 or later and either as 2.24 or nasm 2.10.09 or later.

A.3.4 Data Sets

All datasets are publicly-available graphs that have been converted to the binary format Grazelle expects for its input. Directions for obtaining them are included with the published artifact.

A.4 Installation

Ensure that libnuma is installed and available for linking with the `-lnuma` linker option. On Ubuntu, libnuma may be distributed as the "libnuma-dev" package.

Download the Grazelle source code and extract it to a directory. To verify the results in the paper, use the documented make targets in the section that follows. To perform a custom set of tests, type `make help` for all available options.

A.5 Experiment Workflow

First build Grazelle, then run it using a valid set of command-line options.

A.5.1 Building Grazelle

By default the build system uses the "as" assembler. If assembler issues are encountered, it is possible that the installed version of "as" is too old. In that case, "nasm" [2] can be downloaded and used instead by placing the binary into a location covered by the `PATH` environment variable and switching the "AS" variable value to "nasm" in the Makefile.

We have grouped experimental configuration flags into Makefile targets named to correspond to the figures as numbered in the paper. Simply type `make` followed by one of the following named targets to configure Grazelle for the experiment that is described.

- **fig567-trad, fig567-tradna, fig567-sa**: Configures Grazelle to run the PageRank Scheduler Awareness experiments (Figures 5, 6, and 7). Suffixes "-trad", "-tradna", and "-sa" produce traditional, traditional-nonatomic, and scheduler-aware versions respectively.
- **fig8a-trad, fig8a-tradna, fig8a-sa, fig8b-trad, fig8b-tradna, fig8b-sa**: Configures Grazelle to run the Connected Components Scheduler Awareness experiments (Figure 8). "8a" targets produce write-intense versions, "8b" targets produce standard versions, and suffixes "-trad", "-tradna", and "-sa" produce traditional, traditional-nonatomic, and scheduler-aware versions respectively.
- **fig9**: Configures Grazelle to output vector packing efficiency results for vectors of length 4, 8, and 16. The results of this experiment are shown in Figure 9.

[2] NASM is available at http://www.nasm.us.

- **fig10a-edgepull-base, fig10a-edgepull-vec, fig10a-edgepush-base, fig10a-edgepush-vec, fig10a-vertex-base, fig10a-vertex-vec**: Configures Grazelle to run the per-phase vectorization performance tests (Figure 10a); "edgepull", "edgepush", and "vertex" respectively identify the phase of execution, and "base" and "vec" respectively identify the baseline and vectorized implementations.
- **fig10b-pr-base, fig10b-pr-vec, fig10b-cc-base, fig10b-cc-vec, fig10b-bfs-base, fig10b-bfs-vec**: Configures Grazelle to run the end-to-end application vectorization performance tests (Figure 10b); "pr", "cc", and "bfs" identify the application, and "base" and "vec" respectively identify the baseline and vectorized implementations.
- **fig11-pull, fig11-push, fig12, fig13**: Configures Grazelle for performance comparisons with other frameworks (Figures 11, 12, and 13).

A.5.2 Running Grazelle

Once Grazelle is built, it can be run by typing its executable name and supplying appropriate command-line flags. Documentation is available by using `-h` as the command-line flag; however, for convenience the most common flags are documented here.

- `-i [graph-path]`: Obligatory graph filename. Both the "-push" and "-pull" files must be located in the same directory, and the name of the graph should be specified without the "-push" or "-pull" suffix. Grazelle adds these suffixes automatically to the path specified to this command-line option.
- `-u [numa-nodes]`: Comma-delimited list of NUMA nodes Grazelle should use to run the graph application. For example, `-u 0,2` specifies that nodes 0 and 2 should be used. By default only the first node in the system is used.
- `-n [num-threads]`: Total number of threads that should be used for running the graph application. By default Grazelle uses all available threads on the configured NUMA node(s).
- `-N [num-iterations]`: Number of iterations of PageRank to run (ignored for the other applications). Defaults to 1.
- `-s [sched-granularity]`: Scheduling granularity to use, expressed as number of edge vectors per unit of work. Default behavior is to create $32N$ units of work, where N is the number of threads.
- `-o [output-file]`: If specified, causes Grazelle to write output produced by the running application to the specified file. For PageRank this is the final rank of each vertex, for Connected Components this is the component identifier of each vertex, and for Breadth-First Search this is the parent of each vertex.

A.6 Evaluation and Expected Results

Grazelle's output takes the form of execution statistics printed to standard output. Of particular interest is the running time, which is the result we recorded for many of the graphs in the paper. PageRank output includes a "PageRank Sum" field, which is a simple correctness check that should always show a value very close to 1.0.

For Connected Components and Breadth-First Search the number of iterations is controlled automatically by the application. For PageRank, it is a good idea to use the `-N` option to set the number of iterations to a number high enough to get steady-state behavior. We suggest the iteration counts shown in Table 2.

Table 2. Suggested PageRank iteration counts.

Graph	fig10a-vertex-*	All Others
cit-Patents	1024	1024
dimacs-usa	256	256
livejournal	1024	256
twitter-2010	64	16
friendster	64	16
uk-2007	32	16

A.7 Notes

Each invocation of the Grazelle executable produces a single data point. Reproducing figures generally requires data points obtained from multiple invocations that are then compared. For example, replicating Figure 10 requires comparing corresponding data points obtained using baseline and vectorized configurations, and replicating Figure 7 requires sweeping the `-n` command-line parameter.

We used the "perf" tool to generate Figure 5b. We are not able to supply a script to automate the process of generating that graph, as it involved manually looking at traces to capture time percentages spent in specific functions.

Reproducing Figures 11, 12, and 13 requires comparing performance results produced by Grazelle with those produced by other frameworks. Instructions and resources needed to obtain, build, and run these other frameworks are included in the published artifact.